Package 'spatstat.random'

July 27, 2026

Version 3.5-1

Date 2026-07-26

Title Random Generation Functionality for the 'spatstat' Family

Maintainer Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

Depends R (>= 3.5.0), spatstat.data (>= 3.1-9), spatstat.univar (>= 3.2-0), spatstat.geom (>= 3.8-2), stats, utils, methods, grDevices

Imports Matrix, spatstat.utils (>= 3.2-4), spatstat.sparse (>= 3.2-0)

Suggests spatial, spatstat.linnet (>= 3.5), spatstat.explore (>= 3.8), spatstat.model (>= 3.7), spatstat (>= 3.6), gsl

Description Functionality for random generation of spatial data in the 'spatstat' family of packages. Generates random spatial patterns of points according to many simple rules (complete spatial randomness,
Poisson, binomial, random grid, systematic, cell), randomised alteration of patterns (thinning, random shift, jittering), simulated realisations of random point processes including simple sequential inhibition, Matern inhibition models, Neyman-Scott cluster processes (using direct, Brix-Kendall, or hybrid algorithms),
log-Gaussian Cox processes, product shot noise cluster processes
and Gibbs point processes (using Metropolis-Hastings birth-death-shift algorithm, alternating Gibbs sampler, or coupling-from-the-past perfect simulation).
Also generates random spatial patterns of line segments,
random tessellations, and random images (random noise, random mosaics).
Excludes random generation on a linear network,
which is covered by the separate package 'spatstat.linnet'.

License GPL (>= 2)

URL http://spatstat.org/

NeedsCompilation yes

ByteCompile true

BugReports https://github.com/spatstat/spatstat.random/issues

Author Adrian Baddeley [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0001-9499-8382>),

Rolf Turner [aut, cph] (ORCID: <https://orcid.org/0000-0001-5521-5218>),
Ege Rubak [aut, cph] (ORCID: <https://orcid.org/0000-0002-6675-533X>),
Tilman Davies [aut, cph] (ORCID:
<https://orcid.org/0000-0003-0565-1825>),
Kasper Klitgaard Berthelsen [ctb, cph],
David Bryant [ctb, cph],
Ya-Mei Chang [ctb, cph],
Ute Hahn [ctb],
Abdollah Jalilian [ctb],
Dominic Schuhmacher [ctb, cph],
Rasmus Plenge Waagepetersen [ctb, cph]

Repository CRAN

Date/Publication 2026-07-27 07:20:02 UTC

Contents

spatstat.random-package 4
as.owin.rmhmodel 10
clusterfield 12
clusterkernel 14
clusterprocess 15
clusterradius 16
default.expand 17
default.rmhcontrol 19
dmixpois 20
domain.rmhmodel 21
dpakes 23
expand.owin 24
gauss.hermite 25
is.stationary 26
methods.clusterprocess 27
quadratresample 29
rags 30
ragsAreaInter 31
ragsMultiHard 33
rCauchy 34
rcell 38
rcellnumber 40
rclusterBKBC 41
rDGS 44
rdiffuse 46
rDiggleGratton 47
reach 49
recipEnzpois 51
rGaussPoisson 52
rGRFgauss 53
rHardcore 55

rjitter.psp . . . 57
rknn . . . 58
rlabel . . . 59
rLGCP . . . 61
rMatClust . . . 64
rMaternI . . . 68
rMaternII . . . 69
rmh . . . 71
rmh.default . . . 72
rmhcontrol . . . 83
rmhexpand . . . 87
rmhmodel . . . 89
rmhmodel.default . . . 90
rmhmodel.list . . . 97
rmhstart . . . 100
rMosaicField . . . 101
rMosaicSet . . . 103
rmpoint . . . 104
rmpoispp . . . 108
rNeymanScott . . . 111
rnoise . . . 114
rPenttinen . . . 116
rpoint . . . 117
rpoint3 . . . 120
rpoisDirichletTess . . . 121
rpoisline . . . 123
rpoislinetess . . . 124
rpoispp . . . 125
rpoispp3 . . . 127
rpoisppOnLines . . . 129
rpoisppx . . . 131
rPoissonCluster . . . 132
rpoistrunc . . . 134
rPSNCP . . . 136
rshift . . . 138
rshift.ppp . . . 139
rshift.psp . . . 142
rshift.splitppp . . . 144
rSSI . . . 145
rstrat . . . 147
rStrauss . . . 149
rStraussHard . . . 151
rtemper . . . 152
rthin . . . 154
rthinclumps . . . 155
rThomas . . . 157
runifdisc . . . 161
runifpoint . . . 162

runifpoint3 . . . 164
runifpointOnLines . . . 165
runifpointx . . . 167
rUnround . . . 168
rVarGamma . . . 169
update.rmhcontrol . . . 173
will.expand . . . 174
Window.rmhmodel . . . 175

Index **177**

`spatstat.random-package`

The spatstat.random Package

Description

The **spatstat.random** package belongs to the **spatstat** family of packages. It contains the functionality for generating random spatial patterns and simulation of random point processes.

Details

spatstat is a family of R packages for the statistical analysis of spatial data. Its main focus is the analysis of spatial patterns of points in two-dimensional space.

This sub-package **spatstat.random** contains the functions that perform random generation of spatial patterns and simulation of random point processes:

- generation of random spatial patterns of points according to many simple rules (completely random patterns, random grids, systematic random points, stratified random points, simple sequential inhibition, cell process);
- randomised alteration of spatial patterns (thinning, random shifting, jittering, resampling, relabelling, diffusion);
- direct simulation of random point processes (Poisson process, binomial process, cell process, simple sequential inhibition, Matérn inhibition models, log-Gaussian Cox processes;
- simulation of Neyman-Scott cluster processes (truncated direct algorithm, Brix-Kendall and hybrid algorithms) and product shot noise cluster processes;
- simulation of Gibbs point processes (Metropolis-Hastings birth-death-shift algorithm, perfect simulation/ dominated coupling from the past, alternating Gibbs sampler).

Some other types of spatial object are also supported:

- generation of random patterns of points in 3 dimensions;
- generation of random spatial patterns of line segments;
- generation of random tessellations;
- generation of random images (random noise, random mosaics);
- generation of realisations of mixed Poisson or truncated Poisson random variables.

There are some **exceptions**:

- generation of quasi-random patterns is provided in the **spatstat.geom** package;
- generation of determinantal point processes is provided in the **spatstat.model** package;
- generation of point patterns on a linear network is provided in the **spatstat.linnet** package;
- generation of a real-valued random variable from a kernel density estimate is provided in the **spatstat.univar** package;
- simulation of a Markov chain using a sparse representation of the transition matrix is provided in the **spatstat.sparse** package.

Structure of the spatstat family

The **spatstat** family of packages is designed to support a complete statistical analysis of spatial data. It supports

- creation, manipulation and plotting of point patterns;
- exploratory data analysis;
- spatial random sampling;
- simulation of point process models;
- parametric model-fitting;
- non-parametric smoothing and regression;
- formal inference (hypothesis tests, confidence intervals);
- model diagnostics.

The orginal **spatstat** package grew to be very large. It has now been divided into several **sub-packages**:

- **spatstat.utils** containing basic utilities
- **spatstat.sparse** containing linear algebra utilities
- **spatstat.data** containing datasets
- **spatstat.univar** containing functions for estimating probability distributions of random variables
- **spatstat.geom** containing geometrical objects and geometrical operations
- **spatstat.random** containing functionality for simulation and random generation
- **spatstat.explore** containing the main functionality for exploratory data analysis and nonparametric analysis
- **spatstat.model** containing the main functionality for parametric modelling and formal inference for spatial data
- **spatstat.linnet** containing functions for spatial data on a linear network
- **spatstat**, which simply loads the other sub-packages listed above, and provides documentation.

When you install **spatstat**, these sub-packages are also installed. Then if you load the **spatstat** package by typing `library(spatstat)`, the other sub-packages listed above will automatically be loaded or imported.

For an overview of all the functions available in the sub-packages of **spatstat**, see the help file for `"spatstat-package"` in the **spatstat** package.

Additionally there are several **extension packages:**

- **spatstat.gui** for interactive graphics
- **spatstat.local** for local likelihood (including geographically weighted regression)
- **spatstat.Knet** for additional, computationally efficient code for linear networks
- **spatstat.sphere** (under development) for spatial data on a sphere, including spatial data on the earth's surface

The extension packages must be installed separately and loaded explicitly if needed. They also have separate documentation.

Functionality in spatstat.random

Following is a list of the functionality provided in the **spatstat.random** package only.

To simulate a random point pattern:

`runifpoint`	generate n independent uniform random points
`rpoint`	generate n independent random points
`rmpoint`	generate n independent multitype random points
`rpoispp`	simulate the (in)homogeneous Poisson point process
`rmpoispp`	simulate the (in)homogeneous multitype Poisson point process
`runifdisc`	generate n independent uniform random points in disc
`rstrat`	stratified random sample of points
`rMaternI`	simulate the Matérn Model I inhibition process
`rMaternII`	simulate the Matérn Model II inhibition process
`rSSI`	simulate Simple Sequential Inhibition process
`rStrauss`	simulate Strauss process (perfect simulation)
`rHardcore`	simulate Hard Core process (perfect simulation)
`rStraussHard`	simulate Strauss-hard core process (perfect simulation)
`rDiggleGratton`	simulate Diggle-Gratton process (perfect simulation)
`rDGS`	simulate Diggle-Gates-Stibbard process (perfect simulation)
`rPenttinen`	simulate Penttinen process (perfect simulation)
`rNeymanScott`	simulate a general Neyman-Scott process
`rPoissonCluster`	simulate a general Poisson cluster process
`rMatClust`	simulate the Matérn Cluster process
`rThomas`	simulate the Thomas process
`rGaussPoisson`	simulate the Gauss-Poisson cluster process
`rCauchy`	simulate Neyman-Scott Cauchy cluster process
`rVarGamma`	simulate Neyman-Scott Variance Gamma cluster process
`rthin`	random thinning
`rcell`	simulate the Baddeley-Silverman cell process
`rmh`	simulate Gibbs point process using Metropolis-Hastings

rags	alternating Gibbs sampler for multitype processes
ragsAreaInter	alternating Gibbs sample for area-interaction process
ragsMultiHard	alternating Gibbs sample for multitype hardcore process
runifpointOnLines	generate n random points along specified line segments
rpoisppOnLines	generate Poisson random points along specified line segments

To randomly change an existing point pattern:

rshift	random shifting of points
rthin	random thinning
rlabel	random (re)labelling of a multitype point pattern
quadratresample	block resampling

See also rjitter and rexplode in the **spatstat.geom** package.

Random pixel images:

An object of class "im" represents a pixel image.

rnoise	random pixel noise

Line segment patterns

An object of class "psp" represents a pattern of straight line segments.

rpoisline	generate a realisation of the Poisson line process inside a window

Tessellations

An object of class "tess" represents a tessellation.

rpoislinetess	generate tessellation using Poisson line process
rpoisDirichletTess	generate Dirichlet tessellation of Poisson point process

Three-dimensional point patterns

An object of class "pp3" represents a three-dimensional point pattern in a rectangular box. The box is represented by an object of class "box3".

runifpoint3	generate uniform random points in 3-D
rpoispp3	generate Poisson random points in 3-D

Multi-dimensional space-time point patterns

An object of class "ppx" represents a point pattern in multi-dimensional space and/or time.

runifpointx	generate uniform random points
rpoisppx	generate Poisson random points

Probability Distributions

rknn	theoretical distribution of nearest neighbour distance
dmixpois	mixed Poisson distribution

Simulation

There are many ways to generate a random point pattern, line segment pattern, pixel image or tessellation in **spatstat**.

Random point patterns:

runifpoint	generate n independent uniform random points
rpoint	generate n independent random points
rmpoint	generate n independent multitype random points
rpoispp	simulate the (in)homogeneous Poisson point process
rmpoispp	simulate the (in)homogeneous multitype Poisson point process
runifdisc	generate n independent uniform random points in disc
rstrat	stratified random sample of points
rMaternI	simulate the Matérn Model I inhibition process
rMaternII	simulate the Matérn Model II inhibition process
rSSI	simulate Simple Sequential Inhibition process
rHardcore	simulate hard core process (perfect simulation)
rStrauss	simulate Strauss process (perfect simulation)
rStraussHard	simulate Strauss-hard core process (perfect simulation)
rDiggleGratton	simulate Diggle-Gratton process (perfect simulation)
rDGS	simulate Diggle-Gates-Stibbard process (perfect simulation)
rPenttinen	simulate Penttinen process (perfect simulation)
rNeymanScott	simulate a general Neyman-Scott process
rMatClust	simulate the Matérn Cluster process
rThomas	simulate the Thomas process
rLGCP	simulate the log-Gaussian Cox process
rGaussPoisson	simulate the Gauss-Poisson cluster process
rCauchy	simulate Neyman-Scott process with Cauchy clusters
rVarGamma	simulate Neyman-Scott process with Variance Gamma clusters
rcell	simulate the Baddeley-Silverman cell process
runifpointOnLines	generate n random points along specified line segments
rpoisppOnLines	generate Poisson random points along specified line segments

Resampling a point pattern:

quadratresample	block resampling
rshift	random shifting of (subsets of) points
rthin	random thinning

Other random patterns:

rpoisline	simulate the Poisson line process within a window
rpoislinetess	generate random tessellation using Poisson line process
rMosaicSet	generate random set by selecting some tiles of a tessellation

rMosaicField	generate random pixel image by assigning random values in each tile of a tessellation

Resampling and randomisation procedures

You can build your own tests based on randomisation and resampling using the following capabilities:

quadratresample	block resampling
rshift	random shifting of (subsets of) points
rthin	random thinning

Licence

This library and its documentation are usable under the terms of the "GNU General Public License", a copy of which is distributed with the package.

Acknowledgements

Kasper Klitgaard Berthelsen, Ya-Mei Chang, Tilman Davies, Ute Hahn, Abdollah Jalilian, Dominic Schuhmacher and Rasmus Waagepetersen made substantial contributions of code.

For comments, corrections, bug alerts and suggestions, we thank Monsuru Adepeju, Corey Anderson, Ang Qi Wei, Ryan Arellano, Jens Åström, Robert Aue, Marcel Austenfeld, Sandro Azaele, Guy Bayegnak, Colin Beale, Melanie Bell, Thomas Bendtsen, Ricardo Bernhardt, Andrew Bevan, Brad Biggerstaff, Anders Bilgrau, Leanne Bischof, Christophe Biscio, Roger Bivand, Jose M. Blanco Moreno, Florent Bonneu, Jordan Brown, Ian Buller, Julian Burgos, Simon Byers, Ya-Mei Chang, Jianbao Chen, Igor Chernayavsky, Y.C. Chin, Bjarke Christensen, Lucía Cobo Sanchez, Jean-Francois Coeurjolly, Kim Colyvas, Hadrien Commenges, Rochelle Constantine, Robin Corria Ainslie, Richard Cotton, Marcelino de la Cruz, Peter Dalgaard, Mario D'Antuono, Sourav Das, Peter Diggle, Patrick Donnelly, Ian Dryden, Stephen Eglen, Ahmed El-Gabbas, Belarmain Fandohan, Olivier Flores, David Ford, Peter Forbes, Shane Frank, Janet Franklin, Funwi-Gabga Neba, Oscar Garcia, Agnes Gault, Jonas Geldmann, Marc Genton, Shaaban Ghalandarayeshi, Jason Goldstick, Pavel Grabarnik, C. Graf, Ute Hahn, Andrew Hardegen, Martin Bøgsted Hansen, Martin Hazelton, Juha Heikkinen, Mandy Hering, Markus Herrmann, Maximilian Hesselbarth, Paul Hewson, Hamidreza Heydarian, Kurt Hornik, Philipp Hunziker, Jack Hywood, Ross Ihaka, Čenk Içös, Aruna Jammalamadaka, Robert John-Chandran, Devin Johnson, Mahdieh Khanmohammadi, Bob Klaver, Lily Kozmian-Ledward, Peter Kovesi, Mike Kuhn, Jeff Laake, Robert Lamb, Frédéric Lavancier, Tom Lawrence, Tomas Lazauskas, Jonathan Lee, George Leser, Angela Li, Li Haitao, George Limitsios, Andrew Lister, Nestor Luambua, Ben Madin, Martin Maechler, Kiran Marchikanti, Jeff Marcus, Robert Mark, Peter McCullagh, Monia Mahling, Jorge Mateu Mahiques, Ulf Mehlig, Frederico Mestre, Sebastian Wastl Meyer, Mi Xiangcheng, Lore De Middeleer, Robin Milne, Enrique Miranda, Jesper Møller, Annie Mollié, Ines Moncada, Mehdi Moradi, Virginia Morera Pujol, Erika Mudrak, Gopalan Nair, Nader Najari, Nicoletta Nava, Linda Stougaard Nielsen, Felipe Nunes, Jens Randel Nyengaard, Jens Oehlschlägel, Thierry Onkelinx, Sean O'Riordan, Evgeni Parilov, Jeff Picka, Nicolas Picard, Tim Pollington, Mike Porter, Sergiy Protsiv, Adrian Raftery, Ben Ramage, Pablo Ramon, Xavier Raynaud, Nicholas Read, Matt Reiter, Ian Renner, Tom Richardson, Brian Ripley, Ted Rosenbaum, Barry Rowlingson, Jason Rudokas, Tyler Rudolph, John Rudge, Christopher Ryan, Farzaneh Safavimanesh, Aila Särkkä, Cody Schank, Katja Schladitz, Sebastian Schutte, Bryan Scott, Olivia Semboli, François Sémécurbe, Vadim Shcherbakov, Shen Guochun, Shi Peijian, Harold-Jeffrey Ship, Tammy L Silva, Ida-Maria Sintorn, Yong Song, Malte Spiess, Mark Stevenson,

Kaspar Stucki, Jan Sulavik, Michael Sumner, P. Surovy, Ben Taylor, Thordis Linda Thorarinsdottir, Leigh Torres, Berwin Turlach, Torben Tvedebrink, Kevin Ummer, Medha Uppala, Malissa Usher, Andrew van Burgel, Tobias Verbeke, Mikko Vihtakari, Alexendre Villers, Fabrice Vinatier, Maximilian Vogtland, Sasha Voss, Sven Wagner, Hao Wang, H. Wendrock, Jan Wild, Carl G. Witthoft, Selene Wong, Maxime Woringer, Luke Yates, Mike Zamboni and Achim Zeileis.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

`as.owin.rmhmodel` *Convert Data To Class owin*

Description

Converts data specifying an observation window in any of several formats, into an object of class `"owin"`.

Usage

```
 ## S3 method for class 'rmhmodel'
as.owin(W, ..., fatal=FALSE)
```

Arguments

`W`	Data specifying an observation window, in any of several formats described under *Details* below.
`fatal`	Logical value determining what to do if the data cannot be converted to an observation window. See Details.
`...`	Ignored.

Details

The class `"owin"` is a way of specifying the observation window for a point pattern. See `owin.object` for an overview.

The generic function `as.owin` converts data in any of several formats into an object of class `"owin"` for use by the **spatstat** package. The function `as.owin` is generic, with methods for different classes of objects, and a default method.

The argument `W` may be

- an object of class `"owin"`
- a structure with entries `xrange`, `yrange` specifying the x and y dimensions of a rectangle
- a structure with entries named `xmin`, `xmax`, `ymin`, `ymax` (in any order) specifying the x and y dimensions of a rectangle. This will accept objects of class `bbox` in the `sf` package.

- a numeric vector of length 4 (interpreted as `(xmin, xmax, ymin, ymax)` in that order) specifying the x and y dimensions of a rectangle
- a structure with entries named `xl`, `xu`, `yl`, `yu` (in any order) specifying the x and y dimensions of a rectangle as `(xmin, xmax) = (xl, xu)` and `(ymin, ymax) = (yl, yu)`. This will accept objects of class `spp` used in the Venables and Ripley **spatial** package.
- an object of class `"ppp"` representing a point pattern. In this case, the object's `window` structure will be extracted.
- an object of class `"psp"` representing a line segment pattern. In this case, the object's `window` structure will be extracted.
- an object of class `"tess"` representing a tessellation. In this case, the object's `window` structure will be extracted.
- an object of class `"quad"` representing a quadrature scheme. In this case, the window of the `data` component will be extracted.
- an object of class `"im"` representing a pixel image. In this case, a window of type `"mask"` will be returned, with the same pixel raster coordinates as the image. An image pixel value of `NA`, signifying that the pixel lies outside the window, is transformed into the logical value `FALSE`, which is the corresponding convention for window masks.
- an object of class `"ppm"`, `"kppm"`, `"slrm"` or `"dppm"` representing a fitted point process model. In this case, if `from="data"` (the default), `as.owin` extracts the original point pattern data to which the model was fitted, and returns the observation window of this point pattern. If `from="covariates"` then `as.owin` extracts the covariate images to which the model was fitted, and returns a binary mask window that specifies the pixel locations.
- an object of class `"lpp"` representing a point pattern on a linear network. In this case, `as.owin` extracts the linear network and returns a window containing this network.
- an object of class `"lppm"` representing a fitted point process model on a linear network. In this case, `as.owin` extracts the linear network and returns a window containing this network.
- A `data.frame` with exactly three columns. Each row of the data frame corresponds to one pixel. Each row contains the x and y coordinates of a pixel, and a logical value indicating whether the pixel lies inside the window.
- A `data.frame` with exactly two columns. Each row of the data frame contains the x and y coordinates of a pixel that lies inside the window.
- an object of class `"distfun"`, `"nnfun"` or `"funxy"` representing a function of spatial location, defined on a spatial domain. The spatial domain of the function will be extracted.
- an object of class `"rmhmodel"` representing a point process model that can be simulated using `rmh`. The window (spatial domain) of the model will be extracted. The window may be `NULL` in some circumstances (indicating that the simulation window has not yet been determined). This is not treated as an error, because the argument `fatal` defaults to `FALSE` for this method.
- an object of class `"layered"` representing a list of spatial objects. See `layered`. In this case, `as.owin` will be applied to each of the objects in the list, and the union of these windows will be returned.
- an object of another suitable class from another package. For full details, see `vignette('shapefiles')`.

If the argument `W` is not in one of these formats and cannot be converted to a window, then an error will be generated (if `fatal=TRUE`) or a value of `NULL` will be returned (if `fatal=FALSE`).

When `W` is a data frame, the argument `step` can be used to specify the pixel grid spacing; otherwise, the spacing will be guessed from the data.

Value

An object of class "owin" (see owin.object) specifying an observation window.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

as.owin, as.owin.ppm, as.owin.lpp.

owin.object, owin.

Additional methods for as.owin may be provided by other packages outside the **spatstat** family.

Examples

```
m <- rmhmodel(cif='poisson', par=list(beta=1), w=square(2))
as.owin(m)
```

---

clusterfield *Field of clusters*

Description

Calculate the superposition of cluster kernels at the location of a point pattern.

Usage

```
clusterfield(model, locations = NULL, ...)

## S3 method for class 'character'
clusterfield(model, locations = NULL, ...)

## S3 method for class 'function'
clusterfield(model, locations = NULL, ..., mu = NULL)
```

Arguments

model	Cluster model. Either a fitted cluster model (object of class "kppm"), a character string specifying the type of cluster model, or a function defining the cluster kernel. See Details.
locations	A point pattern giving the locations of the kernels. Defaults to the centroid of the observation window for the "kppm" method and to the center of a unit square otherwise.
...	Additional arguments passed to density.ppp or the cluster kernel. See Details.
mu	Mean number of offspring per cluster. A single number or a pixel image.

Details

The function clusterfield is generic, with methods for "character" and "function" (described here) and a method for "kppm" (described in clusterfield.kppm).

The calculations are performed by density.ppp and ... arguments are passed thereto for control over the pixel resolution etc. (These arguments are then passed on to pixellate.ppp and as.mask.)

For the method clusterfield.function, the given kernel function should accept vectors of x and y coordinates as its first two arguments. Any additional arguments may be passed through the ....

The method clusterfield.function also accepts the optional parameter mu (defaulting to 1) specifying the mean number of points per cluster (as a numeric) or the inhomogeneous reference cluster intensity (as an "im" object or a function(x,y)). The interpretation of mu is as explained in the simulation functions referenced in the See Also section below.

For the method clusterfield.character, the argument model must be one of the following character strings: model="Thomas" for the Thomas process, model="MatClust" for the Matérn cluster process, model="Cauchy" for the Neyman-Scott cluster process with Cauchy kernel, or model="VarGamma" for the Neyman-Scott cluster process with Variance Gamma kernel. For all these models the parameter scale is required and passed through ... as well as the parameter nu when model="VarGamma". This method calls clusterfield.function so the parameter mu may also be passed through ... and will be interpreted as explained above.

Value

A pixel image (object of class "im").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

clusterfield.kppm.

density.ppp and kppm.

Simulation algorithms for cluster models: rCauchy rMatClust rThomas rVarGamma

Examples

```
# method for functions
kernel <- function(x,y,scal) {
    r <- sqrt(x^2 + y^2)
    ifelse(r > 0,
           dgamma(r, shape=5, scale=scal)/(2 * pi * r),
           0)
}
X <- runifpoint(10)
clusterfield(kernel, X, scal=0.05)
```

clusterkernel *Extract Cluster Offspring Kernel*

Description

Given a cluster point process model, this command returns the probability density of the cluster offspring.

Usage

```
clusterkernel(model, ...)
## S3 method for class 'character'
clusterkernel(model, ...)
```

Arguments

model	Cluster model. Either a fitted cluster or Cox model (object of class "kppm"), or a character string specifying the type of cluster model.
...	Parameter values for the model, when model is a character string.

Details

Given a specification of a cluster point process model, this command returns a function(x,y) giving the two-dimensional probability density of the cluster offspring points assuming a cluster parent located at the origin.

The function clusterkernel is generic, with methods for class "character" (described here) and "kppm" (described in clusterkernel.kppm).

Value

A function in the R language with arguments x, y, ...

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

clusterkernel.kppm), clusterfield, kppm.

Examples

```
f <- clusterkernel("Thomas", kappa=10, scale=0.5)
f(0.1, 0.2)
```

`clusterprocess` *Cluster Point Process Model*

Description

Creates an object representing a cluster point process model with the specified parameters. Typically used for simulations or calculations about such a model.

Usage

```
clusterprocess(name = "Thomas", ..., mu, kappa, scale)
```

Arguments

`name`	Name of the cluster process. One of `"Thomas"`, `"MatClust"`, `"VarGamma"` or `"Cauchy"`.
`...`	Other arguments needed for the model.
`mu`	Mean cluster size. A single number, or a pixel image.
`kappa`	Parent intensity. A single number.
`scale`	Cluster scale parameter of the model.

Details

This function creates an object representing a Neyman-Scott-Cox cluster process model with the specified parameter values. The object belongs to the class `"clusterprocess"`.

This is different from the model-fitting function `kppm` which fits such a model to a point pattern dataset.

There are methods for simulating a `"clusterprocess"`, calculating its moments, and other purposes, listed under *See Also*.

Value

Object of class `"clusterprocess"`.

Author(s)

Adrian Baddeley `<Adrian.Baddeley@curtin.edu.au>`

See Also

Methods for `simulate`, `predict`, `intensity`, `print`, `clusterradius` and `reach` are supplied in package **spatstat.random** and documented under `methods.clusterprocess`.

The functions `Kmodel`, `pcfmodel`, `psib`, `persist`, `clusterstrength` and `varcount` in package **spatstat.model** can also be applied to a `"clusterprocess"`.

Examples

```
m <- clusterprocess("Thomas", kappa=10, mu=5, scale=0.1)
simulate(m, win=square(2))
```

`clusterradius` *Compute or Extract Effective Range of Cluster Kernel*

Description

Given a cluster point process model, this command returns a value beyond which the the probability density of the cluster offspring is neglible.

Usage

```
clusterradius(model, ...)

## S3 method for class 'character'
clusterradius(model, ..., thresh = NULL, precision = FALSE)
```

Arguments

`model`	Cluster model. Either a fitted cluster or Cox model (object of class `"kppm"`), or a character string specifying the type of cluster model.
`...`	Parameter values for the model, when `model` is a character string.
`thresh`	Numerical threshold relative to the cluster kernel value at the origin (parent location) determining when the cluster kernel will be considered neglible. A sensible default is provided.
`precision`	Logical. If `precision=TRUE` the precision of the calculated range is returned as an attribute to the range. See details.

Details

Given a cluster model this function by default returns the effective range of the model with the given parameters as used in spatstat. For the Matérn cluster model (see e.g. `rMatClust`) this is simply the finite radius of the offsring density given by the paramter `scale` irrespective of other options given to this function. The remaining models in spatstat have infinite theoretical range, and an effective finite value is given as follows: For the Thomas model (see e.g. `rThomas` the default is `4*scale` where scale is the scale or standard deviation parameter of the model. If `thresh` is given the value is instead found as described for the other models below.

For the Cauchy model (see e.g. `rCauchy`) and the Variance Gamma (Bessel) model (see e.g. `rVarGamma`) the value of `thresh` defaults to 0.001, and then this is used to compute the range numerically as follows. If $k(x, y) = k_0(r)$ with $r = \sqrt{(x^2 + y^2)}$ denotes the isotropic cluster kernel then $f(r) = 2\pi r k_0(r)$ is the density function of the offspring distance from the parent. The range is determined as the value of r where $f(r)$ falls below `thresh` times $k_0(r)$.

If `precision=TRUE` the precision related to the chosen range is returned as an attribute. Here the precision is defined as the polar integral of the kernel from distance 0 to the calculated range. Ideally this should be close to the value 1 which would be obtained for the true theretical infinite range.

Value

A positive numeric.

Additionally, the precision related to this range value is returned as an attribute "prec", if precision=TRUE.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

clusterkernel, kppm, rMatClust, rThomas, rCauchy, rVarGamma, rNeymanScott.

Examples

```
clusterradius("Thomas", scale = .1)
clusterradius("Thomas", scale = .1, thresh = 0.001)
clusterradius("VarGamma", scale = .1, nu = 2, precision = TRUE)
```

---

default.expand *Default Expansion Rule for Simulation of Model*

Description

Defines the default expansion window or expansion rule for simulation of a point process model.

Usage

```
default.expand(object, m=2, epsilon=1e-6, w=Window(object))
```

Arguments

object	A point process model (object of class "ppm" or "rmhmodel").
m	A single numeric value. The window will be expanded by a distance m * reach(object) along each side.
epsilon	Threshold argument passed to reach to determine reach(object).
w	Optional. The un-expanded window in which the model is defined. The resulting simulated point patterns will lie in this window.

Details

This function computes a default value for the expansion rule (the argument expand in rmhcontrol) given a fitted point process model object. This default is used by rmh, simulate.ppm, envelope, qqplot.ppm, and other functions.

Suppose we wish to generate simulated realisations of a fitted point process model inside a window w. It is advisable to first simulate the pattern on a larger window, and then clip it to the original window w. This avoids edge effects in the simulation. It is called *expansion* of the simulation window.

Accordingly, for the Metropolis-Hastings simulation algorithm rmh, the algorithm control parameters specified by rmhcontrol include an argument expand that determines the expansion of the simulation window.

The function default.expand determines the default expansion rule for a fitted point process model object.

If the model is Poisson, then no expansion is necessary. No expansion is performed by default, and default.expand returns a rule representing no expansion. The simulation window is the original window w = Window(object).

If the model depends on external covariates (i.e.\ covariates other than the Cartesian covariates x and y and the marks) then no expansion is feasible, in general, because the spatial domain of the covariates is not guaranteed to be large enough. default.expand returns a rule representing no expansion. The simulation window is the original window w = Window(object).

If the model depends on the Cartesian covariates x and y, it would be feasible to expand the simulation window, and this was the default for **spatstat** version 1.24-1 and earlier. However this sometimes produces artefacts (such as an empty point pattern) or memory overflow, because the fitted trend, extrapolated outside the original window of the data, may become very large. In **spatstat** version 1.24-2 and later, the default rule is *not* to expand if the model depends on x or y. Again default.expand returns a rule representing no expansion.

Otherwise, expansion will occur. The original window w = Window(object) is expanded by a distance m * rr, where rr is the interaction range of the model, computed by reach. If w is a rectangle then each edge of w is displaced outward by distance m * rr. If w is not a rectangle then w is dilated by distance m * rr using dilation.

Value

A window expansion rule (object of class "rmhexpand").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

rmhexpand, rmhcontrol, rmh, envelope, qqplot.ppm

Examples

```
if(require(spatstat.model)) {
  fit <- ppm(cells ~1, Strauss(0.07), nd=20)
```

```
  default.expand(fit)
}
  mod <- rmhmodel(cif="strauss", par=list(beta=100, gamma=0.5, r=0.07))
  default.expand(mod)
```

`default.rmhcontrol` *Set Default Control Parameters for Metropolis-Hastings Algorithm.*

Description

For a Gibbs point process model (either a fitted model, or a model specified by its parameters), this command sets appropriate default values of the parameters controlling the iterative behaviour of the Metropolis-Hastings algorithm.

Usage

```
default.rmhcontrol(model, w=NULL)
```

Arguments

`model`	A fitted point process model (object of class `"ppm"`) or a description of a Gibbs point process model (object of class `"rmhmodel"`).
`w`	Optional. Window for the resulting simulated patterns.

Details

This function sets the values of the parameters controlling the iterative behaviour of the Metropolis-Hastings simulation algorithm. It uses default values that would be appropriate for the fitted point process model `model`.

The expansion parameter `expand` is set to `default.expand(model, w)`.

All other parameters revert to their defaults given in `rmhcontrol.default`.

See `rmhcontrol` for the full list of control parameters. To override default parameters, use `update.rmhcontrol`.

Value

An object of class `"rmhcontrol"`. See `rmhcontrol`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

`rmhcontrol`, `update.rmhcontrol`, `ppm`, `default.expand`

Examples

```
if(require(spatstat.model)) {
  fit <- ppm(cells, ~1, Strauss(0.1))
  default.rmhcontrol(fit)
  default.rmhcontrol(fit, w=square(2))
}
   m <- rmhmodel(cif='strauss',
                 par=list(beta=100, gamma=0.5, r=0.1),
                 w=unit.square())
  default.rmhcontrol(m)
  default.rmhcontrol(m, w=square(2))
```

`dmixpois` *Mixed Poisson Distribution*

Description

Density, distribution function, quantile function and random generation for a mixture of Poisson distributions.

Usage

```
dmixpois(x, mu, sd, invlink = exp, GHorder = 5)
pmixpois(q, mu, sd, invlink = exp, lower.tail = TRUE, GHorder = 5)
qmixpois(p, mu, sd, invlink = exp, lower.tail = TRUE, GHorder = 5)
rmixpois(n, mu, sd, invlink = exp)
```

Arguments

`x`	vector of (non-negative integer) quantiles.
`q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of random values to return.
`mu`	Mean of the linear predictor. A single numeric value.
`sd`	Standard deviation of the linear predictor. A single numeric value.
`invlink`	Inverse link function. A function in the R language, used to transform the linear predictor into the parameter `lambda` of the Poisson distribution.
`lower.tail`	Logical. If `TRUE` (the default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
`GHorder`	Number of quadrature points in the Gauss-Hermite quadrature approximation. A small positive integer.

Details

These functions are analogous to dpois ppois, qpois and rpois except that they apply to a mixture of Poisson distributions.

In effect, the Poisson mean parameter lambda is randomised by setting lambda = invlink(Z) where Z has a Gaussian $N(\mu, \sigma^2)$ distribution. The default is invlink=exp which means that lambda is lognormal. Set invlink=I to assume that lambda is approximately Normal.

For dmixpois, pmixpois and qmixpois, the probability distribution is approximated using Gauss-Hermite quadrature. For rmixpois, the deviates are simulated exactly.

Value

Numeric vector: dmixpois gives probability masses, ppois gives cumulative probabilities, qpois gives (non-negative integer) quantiles, and rpois generates (non-negative integer) random deviates.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

, Rolf Turner <rolfturner@posteo.net>

and Ege Rubak <rubak@math.aau.dk>

See Also

dpois, gauss.hermite.

Examples

```
dmixpois(7, 10, 1, invlink = I)
dpois(7, 10)

pmixpois(7, log(10), 0.2)
ppois(7, 10)

qmixpois(0.95, log(10), 0.2)
qpois(0.95, 10)

x <- rmixpois(100, log(10), log(1.2))
mean(x)
var(x)
```

---

domain.rmhmodel *Extract the Domain of any Spatial Object*

Description

Given a spatial object such as a point pattern, in any number of dimensions, this function extracts the spatial domain in which the object is defined.

Usage

```
 ## S3 method for class 'rmhmodel'
domain(X, ...)
```

Arguments

X	A spatial object such as a point pattern (in any number of dimensions), line segment pattern or pixel image.
...	Extra arguments. They are ignored by all the methods listed here.

Details

The function domain is generic.

For a spatial object X in any number of dimensions, domain(X) extracts the spatial domain in which X is defined.

For a two-dimensional object X, typically domain(X) is the same as Window(X).

Exceptions occur for methods related to linear networks.

Value

A spatial object representing the domain of X. Typically a window (object of class "owin"), a three-dimensional box ("box3"), a multidimensional box ("boxx") or a linear network ("linnet").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

domain, domain.quadratcount, domain.ppm, domain.quadrattest, domain.lpp. Window, Frame.

Examples

```
domain(rmhmodel(cif='poisson', par=list(beta=1), w=square(2)))
```

dpakes *Pakes distribution*

Description

Probability density, cumulative distribution function, quantile function, and random generation for the Pakes distribution.

Usage

```
dpakes(x, zeta)
ppakes(q, zeta)
qpakes(p, zeta)
rpakes(n, zeta)
```

Arguments

x, q	Numeric vector of quantiles.
p	Numeric vector of probabilities
n	Number of observations.
zeta	Mean of distribution. A single, non-negative, numeric value.

Details

These functions concern the probability distribution of the random variable

$$X = \sum_{n=1}^{\infty} \prod_{j=1}^{n} U_j^{1/\zeta}$$

where $U_1, U_2, \ldots$ are independent random variables uniformly distributed on $[0, 1]$ and ζ is a parameter.

This distribution arises in many contexts. For example, for a homogeneous Poisson point process in two-dimensional space with intensity λ, the standard Gaussian kernel estimator of intensity with bandwidth σ, evaluated at any fixed location u, has the same distribution as $(\lambda/\zeta)X$ where $\zeta = 2\pi\lambda\sigma^2$.

Following the usual convention, dpakes computes the probability density, ppakes the cumulative distribution function, and qpakes the quantile function, and rpakes generates random variates with this distribution.

The computation is based on a recursive integral equation for the cumulative distribution function, due to Professor Tony Pakes, presented in Baddeley, Moller and Pakes (2008). The solution uses the fact that the random variable satisfies the distributional equivalence

$$X \equiv U^{1/\zeta}(1 + X)$$

where U is uniformly distributed on $[0, 1]$ and independent of X.

Value

A numeric vector.

Author(s)

Adrian Baddeley.

References

Baddeley, A., Moller, J. and Pakes, A.G. (2008) Properties of residuals for spatial point processes, *Annals of the Institute of Statistical Mathematics* **60**, 627–649.

Examples

```
curve(dpakes(x, 1.5), to=4)
rpakes(3, 1.5)
```

---

expand.owin *Apply Expansion Rule*

Description

Applies an expansion rule to a window.

Usage

```
expand.owin(W, ...)
```

Arguments

W	A window.
...	Arguments passed to rmhexpand to determine an expansion rule.

Details

The argument W should be a window (an object of class "owin").

This command applies the expansion rule specified by the arguments ... to the window W, yielding another window.

The arguments ... are passed to rmhexpand to determine the expansion rule.

For other transformations of the scale, location and orientation of a window, see shift, affine and rotate.

Value

A window (object of class "owin").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

See Also

rmhexpand about expansion rules.

shift, rotate, affine for other types of manipulation.

Examples

```
expand.owin(square(1), 9)
expand.owin(square(1), distance=0.5)
expand.owin(letterR, length=2)
expand.owin(letterR, distance=0.1)
```

---

gauss.hermite *Gauss-Hermite Quadrature Approximation to Expectation for Normal Distribution*

Description

Calculates an approximation to the expected value of any function of a normally-distributed random variable, using Gauss-Hermite quadrature.

Usage

```
gauss.hermite(f, mu = 0, sd = 1, ..., order = 5)
```

Arguments

f	The function whose moment should be approximated.
mu	Mean of the normal distribution.
sd	Standard deviation of the normal distribution.
...	Additional arguments passed to f.
order	Number of quadrature points in the Gauss-Hermite quadrature approximation. A small positive integer.

Details

This algorithm calculates the approximate expected value of f(Z) when Z is a normally-distributed random variable with mean mu and standard deviation sd. The expected value is an integral with respect to the Gaussian density; this integral is approximated using Gauss-Hermite quadrature.

The argument f should be a function in the R language whose first argument is the variable Z. Additional arguments may be passed through .... The value returned by f may be a single numeric value, a vector, or a matrix. The values returned by f for different values of Z must have compatible dimensions.

The result is a weighted average of several values of f.

Value

Numeric value, vector or matrix.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

, Rolf Turner <rolfturner@posteo.net>

and Ege Rubak <rubak@math.aau.dk>.

Examples

```
gauss.hermite(function(x) x^2, 3, 1)
```

---

is.stationary *Recognise Stationary and Poisson Point Process Models*

Description

Given a point process model (either a model that has been fitted to data, or a model specified by its parameters), determine whether the model is a stationary point process, and whether it is a Poisson point process.

Usage

```
is.stationary(x)
## S3 method for class 'rmhmodel'
is.stationary(x)

is.poisson(x)
## S3 method for class 'rmhmodel'
is.poisson(x)
```

Arguments

x	A fitted spatial point process model (object of class "ppm", "kppm", "lppm", "dppm" or "slrm") or a specification of a Gibbs point process model (object of class "rmhmodel") or a similar object.

Details

The argument x represents a fitted spatial point process model or a similar object.

is.stationary(x) returns TRUE if x represents a stationary point process, and FALSE if not.

is.poisson(x) returns TRUE if x represents a Poisson point process, and FALSE if not.

The functions is.stationary and is.poisson are generic, with methods for the classes "ppm" (Gibbs point process models), "kppm" (cluster or Cox point process models), "slrm" (spatial logistic regression models) and "rmhmodel" (model specifications for the Metropolis-Hastings algorithm). Additionally is.stationary has a method for classes "detpointprocfamily" and "dppm"

(both determinantal point processes) and is.poisson has a method for class "interact" (interaction structures for Gibbs models).

is.poisson.kppm will return FALSE, unless the model x is degenerate: either x has zero intensity so that its realisations are empty with probability 1, or it is a log-Gaussian Cox process where the log intensity has zero variance.

is.poisson.slrm will always return TRUE, by convention.

Value

A logical value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

is.marked to determine whether a model is a marked point process.

Examples

```
m <- rmhmodel(cif='strauss', par=list(beta=10, gamma=0.1, r=1))
is.stationary(m)
is.poisson(m)
is.poisson(rmhmodel(cif='strauss', par=list(beta=10, gamma=1, r=1)))
```

---

methods.clusterprocess

Methods for Cluster Models

Description

Methods for the class "clusterprocess" of cluster point process models with specified values of the parameters.

Usage

```
 ## S3 method for class 'clusterprocess'
intensity(X, ...)

 ## S3 method for class 'clusterprocess'
predict(object, ...,
                  locations, type = "intensity", ngrid = NULL)

 ## S3 method for class 'clusterprocess'
print(x, ...)
```

```
 ## S3 method for class 'clusterprocess'
clusterradius(model,...,thresh=NULL, precision=FALSE)

 ## S3 method for class 'clusterprocess'
reach(x, ..., epsilon)

 ## S3 method for class 'clusterprocess'
simulate(object, nsim=1, ..., win=unit.square(), window=win)
```

Arguments

model, object, x, X	Object of class "clusterprocess".
...	Arguments passed to other methods.
locations	Locations where prediction should be performed. A window or a point pattern.
type	Currently must equal "intensity".
ngrid	Pixel grid dimensions for prediction, if locations is a rectangle or polygon.
thresh, epsilon	Tolerance thresholds
precision	Logical value stipulating whether the precision should also be returned.
win, window	Window (object of class "owin") in which the simulated pattern should be generated.
nsim	Number of simulated patterns to be generated.

Details

An object of class "clusterprocess" represents a Neyman-Scott-Cox cluster process model with specified parameter values.

This page documents methods for printing, simulating and predicting such models, supplied by the package **spatstat.random**. Other methods are supplied in other packages.

Value

Same as for other methods.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

See Also

clusterprocess

Examples

```
m <- clusterprocess("Thomas", kappa=10, mu=5, scale=0.1)
m2 <- clusterprocess("VarGamma", kappa=10, mu=10, scale=0.1, nu=0.7)
m
m2
intensity(m)
reach(m)
clusterradius(m)
Z <- predict(m, locations=square(2))
X <- simulate(m, win=square(2))
```

---

quadratresample *Resample a Point Pattern by Resampling Quadrats*

Description

Given a point pattern dataset, create a resampled point pattern by dividing the window into rectangular quadrats and randomly resampling the list of quadrats.

Usage

```
quadratresample(X, nx, ny=nx, ...,
                replace = FALSE, nsamples = 1,
                verbose = (nsamples > 1))
```

Arguments

X	A point pattern dataset (object of class "ppp").
nx, ny	Numbers of quadrats in the x and y directions.
...	Ignored.
replace	Logical value. Specifies whether quadrats should be sampled with or without replacement.
nsamples	Number of randomised point patterns to be generated.
verbose	Logical value indicating whether to print progress reports.

Details

This command implements a very simple bootstrap resampling procedure for spatial point patterns X.

The dataset X must be a point pattern (object of class "ppp") and its observation window must be a rectangle.

The window is first divided into N = nx * ny rectangular tiles (quadrats) of equal size and shape. To generate one resampled point pattern, a random sample of N quadrats is selected from the list of N quadrats, with replacement (if replace=TRUE) or without replacement (if replace=FALSE). The ith quadrat in the original dataset is then replaced by the ith sampled quadrat, after the latter

is shifted so that it occupies the correct spatial position. The quadrats are then reconstituted into a point pattern inside the same window as X.

If replace=FALSE, this procedure effectively involves a random permutation of the quadrats. The resulting resampled point pattern has the same number of points as X. If replace=TRUE, the number of points in the resampled point pattern is random.

Value

A point pattern (if nsamples = 1) or a list of point patterns (if nsamples > 1).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

quadrats, quadratcount.

See varblock to estimate the variance of a summary statistic by block resampling.

Examples

```
quadratresample(bei, 6, 3)
```

---

rags *Alternating Gibbs Sampler for Multitype Point Processes*

Description

Simulate a realisation of a point process model using the alternating Gibbs sampler.

Usage

```
rags(model, ..., ncycles = 100)
```

Arguments

model	Data specifying some kind of point process model.
...	Additional arguments passed to other code.
ncycles	Number of cycles of the alternating Gibbs sampler that should be performed.

Details

The Alternating Gibbs Sampler for a multitype point process is an iterative simulation procedure. Each step of the sampler updates the pattern of points of a particular type `i`, by drawing a realisation from the conditional distribution of points of type `i` given the points of all other types. Successive steps of the sampler update the points of type 1, then type 2, type 3, and so on.

This is an experimental implementation which currently works only for multitype hard core processes (see `MultiHard`) in which there is no interaction between points of the same type.

The argument `model` should be an object describing a point process model. At the moment, the only permitted format for `model` is of the form `list(beta, hradii)` where `beta` gives the first order trend and `hradii` is the matrix of interaction radii. See `ragsMultiHard` for full details.

Value

A point pattern (object of class `"ppp"`).

Author(s)

Adrian Baddeley `<Adrian.Baddeley@curtin.edu.au>`

See Also

`ragsMultiHard`, `ragsAreaInter`

Examples

```
mo <- list(beta=c(30, 20),
           hradii = 0.05 * matrix(c(0,1,1,0), 2, 2))
rags(mo, ncycles=10)
```

`ragsAreaInter` *Alternating Gibbs Sampler for Area-Interaction Process*

Description

Generate a realisation of the area-interaction process using the alternating Gibbs sampler. Applies only when the interaction parameter eta is greater than 1.

Usage

```
ragsAreaInter(beta, eta, r, ...,
              win = NULL, bmax = NULL, periodic = FALSE, ncycles = 100)
```

Arguments

beta	First order trend. A number, a pixel image (object of class "im"), or a function(x,y).
eta	Interaction parameter (canonical form) as described in the help for AreaInter. A number greater than 1.
r	Disc radius in the model. A number greater than 1.
...	Additional arguments for beta if it is a function.
win	Simulation window. An object of class "owin". (Ignored if beta is a pixel image.)
bmax	Optional. The maximum possible value of beta, or a number larger than this.
periodic	Logical value indicating whether to treat opposite sides of the simulation window as being the same, so that points close to one side may interact with points close to the opposite side. Feasible only when the window is a rectangle.
ncycles	Number of cycles of the alternating Gibbs sampler to be performed.

Details

This function generates a simulated realisation of the area-interaction process (see AreaInter) using the alternating Gibbs sampler (see rags).

It exploits a mathematical relationship between the (unmarked) area-interaction process and the two-type hard core process (Baddeley and Van Lieshout, 1995; Widom and Rowlinson, 1970). This relationship only holds when the interaction parameter eta is greater than 1 so that the area-interaction process is clustered.

The parameters beta,eta are the canonical parameters described in the help for AreaInter. The first order trend beta may be a constant, a function, or a pixel image.

The simulation window is determined by beta if it is a pixel image, and otherwise by the argument win (the default is the unit square).

Value

A point pattern (object of class "ppp").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

References

Baddeley, A.J. and Van Lieshout, M.N.M. (1995). Area-interaction point processes. *Annals of the Institute of Statistical Mathematics* **47** (1995) 601–619.

Widom, B. and Rowlinson, J.S. (1970). New model for the study of liquid-vapor phase transitions. *The Journal of Chemical Physics* **52** (1970) 1670–1684.

See Also

rags, ragsMultiHard

AreaInter

Examples

```
plot(ragsAreaInter(100, 2, 0.07, ncycles=15))
```

`ragsMultiHard` *Alternating Gibbs Sampler for Multitype Hard Core Process*

Description

Generate a realisation of the multitype hard core point process using the alternating Gibbs sampler.

Usage

```
ragsMultiHard(beta, hradii, ..., types=NULL, bmax = NULL,
              periodic=FALSE, ncycles = 100)
```

Arguments

`beta`	First order trend. A numeric vector, a pixel image, a function, a list of functions, or a list of pixel images.
`hradii`	Matrix of hard core radii between each pair of types. Diagonal entries should be `0` or `NA`.
`types`	Vector of all possible types for the multitype point pattern.
`...`	Arguments passed to `rmpoispp` when generating random points.
`bmax`	Optional upper bound on `beta`.
`periodic`	Logical value indicating whether to measure distances in the periodic sense, so that opposite sides of the (rectangular) window are treated as identical.
`ncycles`	Number of cycles of the sampler to be performed.

Details

The Alternating Gibbs Sampler for a multitype point process is an iterative simulation procedure. Each step of the sampler updates the pattern of points of a particular type `i`, by drawing a realisation from the conditional distribution of points of type `i` given the points of all other types. Successive steps of the sampler update the points of type 1, then type 2, type 3, and so on.

This is an experimental implementation which currently works only for multitype hard core processes (see `MultiHard`) in which there is no interaction between points of the same type, and for the area-interaction process (see `ragsAreaInter`).

The argument `beta` gives the first order trend for each possible type of point. It may be a single number, a numeric vector, a `function(x,y)`, a pixel image, a list of functions, a `function(x,y,m)`, or a list of pixel images.

The argument `hradii` is the matrix of hard core radii between each pair of possible types of points. Two points of types `i` and `j` respectively are forbidden to lie closer than a distance `hradii[i,j]` apart. The diagonal of this matrix must contain `NA` or `0` values, indicating that there is no hard core constraint applying between points of the same type.

Value

A point pattern (object of class "ppp").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

See Also

rags, ragsAreaInter

Examples

```
b <- c(30,20)
h <- 0.05 * matrix(c(0,1,1,0), 2, 2)
ragsMultiHard(b, h, ncycles=10)
ragsMultiHard(b, h, ncycles=5, periodic=TRUE)
```

---

rCauchy *Simulate Neyman-Scott Point Process with Cauchy cluster kernel*

Description

Generate a random point pattern, a simulated realisation of the Neyman-Scott process with Cauchy cluster kernel.

Usage

```
rCauchy(kappa, scale, mu, win = square(1),
          nsim=1, drop=TRUE,
          ...,
          n.cond = NULL, w.cond = NULL,
          algorithm=c("BKBC", "naive"),
          nonempty=TRUE,
          thresh = 0.001, poisthresh=1e-6,
          expand = NULL,
          saveparents=FALSE, saveLambda=FALSE,
          kappamax=NULL, mumax=NULL, LambdaOnly=FALSE)
```

Arguments

kappa	Intensity of the Poisson process of cluster centres. A single positive number, a function, or a pixel image.
scale	Scale parameter for cluster kernel. Determines the size of clusters. A single positive number, in the same units as the spatial coordinates.
mu	Mean number of points per cluster (a single positive number) or reference intensity for the cluster points (a function or a pixel image).

win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
...	Passed to clusterfield to control the image resolution when saveLambda=TRUE, and to clusterradius when expand is missing or NULL.
n.cond	Optional. Integer specifying a fixed number of points. See the section on *Conditional Simulation*.
w.cond	Optional. Conditioning region. A window (object of class "owin") specifying the region which must contain exactly n.cond points. See the section on *Conditional Simulation*.
algorithm	String (partially matched) specifying the simulation algorithm. See Details.
nonempty	Logical. If TRUE (the default), a more efficient algorithm is used, in which parents are generated conditionally on having at least one offspring point. If FALSE, parents are generated even if they have no offspring. Both choices are valid; the default is recommended unless you need to simulate all the parent points for some other purpose.
thresh	Threshold relative to the cluster kernel value at the origin (parent location) determining when the cluster kernel will be treated as zero for simulation purposes. Will be overridden by argument expand if that is given.
poisthresh	Numerical threshold below which the model will be treated as a Poisson process. See Details.
expand	Window expansion distance. A single number. The distance by which the original window will be expanded in order to generate parent points. Has a sensible default, determined by calling clusterradius with the numeric threshold value given in thresh.
saveparents	Logical value indicating whether to save the locations of the parent points as an attribute.
saveLambda	Logical. If TRUE then the random intensity corresponding to the simulated parent points will also be calculated and saved, and returns as an attribute of the point pattern.
kappamax	Optional. Numerical value which is an upper bound for the values of kappa, when kappa is a pixel image or a function.
mumax	Optional. Numerical value which is an upper bound for the values of mu, when mu is a pixel image or a function.
LambdaOnly	Logical value specifying whether to return only the random intensity, rather than the point pattern.

Details

This algorithm generates a realisation of the Neyman-Scott process with Cauchy cluster kernel, inside the window win.

The process is constructed by first generating a Poisson point process of "parent" points with intensity kappa. Then each parent point is replaced by a random cluster of points, the number of points in each cluster being random with a Poisson (mu) distribution, and the points being placed independently and uniformly according to a Cauchy kernel.

Note that, for correct simulation of the model, the parent points are not restricted to lie inside the window win; the parent process is effectively the uniform Poisson process on the infinite plane.

The algorithm can also generate spatially inhomogeneous versions of the cluster process:

- The parent points can be spatially inhomogeneous. If the argument kappa is a function(x,y) or a pixel image (object of class "im"), then it is taken as specifying the intensity function of an inhomogeneous Poisson process that generates the parent points.
- The offspring points can be inhomogeneous. If the argument mu is a function(x,y) or a pixel image (object of class "im"), then it is interpreted as the reference density for offspring points, in the sense of Waagepetersen (2006).

When the parents are homogeneous (kappa is a single number) and the offspring are inhomogeneous (mu is a function or pixel image), the model can be fitted to data using kppm.

If the pair correlation function of the model is very close to that of a Poisson process, deviating by less than poisthresh, then the model is approximately a Poisson process, and will be simulated as a Poisson process with intensity kappa * mu, using rpoispp. This avoids computations that would otherwise require huge amounts of memory.

Value

A point pattern (object of class "ppp") or a list of point patterns.

Additionally, some intermediate results of the simulation are returned as attributes of this point pattern (see rNeymanScott). Furthermore, the simulated intensity function is returned as an attribute "Lambda", if saveLambda=TRUE.

If LambdaOnly=TRUE the result is a pixel image (object of class "im") or a list of pixel images.

Simulation Algorithm

Two simulation algorithms are implemented.

- The *naive* algorithm generates the cluster process by directly following the description given above. First the window win is expanded by a distance equal to expand. Then the parent points are generated in the expanded window according to a Poisson process with intensity kappa. Then each parent point is replaced by a finite cluster of offspring points as described above. The naive algorithm is used if algorithm="naive" or if nonempty=FALSE.
- The *BKBC* algorithm, proposed by Baddeley and Chang (2023), is a modification of the algorithm of Brix and Kendall (2002). Parents are generated in the infinite plane, subject to the condition that they have at least one offspring point inside the window win. The BKBC algorithm is used when algorithm="BKBC" (the default) and nonempty=TRUE (the default).

The naive algorithm becomes very slow when scale is large, while the BKBC algorithm is uniformly fast (Baddeley and Chang, 2023).

If saveparents=TRUE, then the simulated point pattern will have an attribute "parents" containing the coordinates of the parent points, and an attribute "parentid" mapping each offspring point to its parent.

If nonempty=TRUE (the default), then parents are generated subject to the condition that they have at least one offspring point in the window win. nonempty=FALSE, then parents without offspring will be included; this option is not available in the *BKBC* algorithm.

Note that if kappa is a pixel image, its domain must be larger than the window win. This is because an offspring point inside win could have its parent point lying outside win. In order to allow this, the naive simulation algorithm first expands the original window win by a distance equal to expand and generates the Poisson process of parent points on this larger window. If kappa is a pixel image, its domain must contain this larger window.

If the pair correlation function of the model is very close to that of a Poisson process, with maximum deviation less than poisthresh, then the model is approximately a Poisson process. This is detected by the naive algorithm which then simulates a Poisson process with intensity kappa * mu, using rpoispp. This avoids computations that would otherwise require huge amounts of memory.

Conditional Simulation

If n.cond is specified, it should be a single integer. Simulation will be conditional on the event that the pattern contains exactly n.cond points (or contains exactly n.cond points inside the region w.cond if it is given).

Conditional simulation uses the rejection algorithm described in Section 6.2 of Moller, Syversveen and Waagepetersen (1998). There is a maximum number of proposals which will be attempted. Consequently the return value may contain fewer than nsim point patterns.

The current algorithm for conditional simulation ignores the argument saveparents and does not save the parent points.

Fitting cluster models to data

The Cauchy cluster model with homogeneous parents (i.e. where kappa is a single number) where the offspring are either homogeneous or inhomogeneous (mu is a single number, a function or pixel image) can be fitted to point pattern data using kppm, or fitted to the inhomogeneous K function using cauchy.estK or cauchy.estpcf.

Currently **spatstat** does not support fitting the Cauchy cluster process model with inhomogeneous parents.

A Cauchy cluster process model fitted by kppm can be simulated automatically using simulate.kppm (which invokes rCauchy to perform the simulation).

Author(s)

Original algorithm by Abdollah Jalilian and Rasmus Waagepetersen. Adapted for **spatstat** by Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>. Brix-Kendall-Baddeley-Chang algorithm implemented by Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Ya-Mei Chang <yamei628@gmail.com>.

References

Baddeley, A. and Chang, Y.-M. (2023) Robust algorithms for simulating cluster point processes. *Journal of Statistical Computation and Simulation* **93**, 1950–1975.

Brix, A. and Kendall, W.S. (2002) Simulation of cluster point processes without edge effects. *Advances in Applied Probability* **34**, 267–280.

Ghorbani, M. (2013) Cauchy cluster process. *Metrika* **76**, 697-706.

Jalilian, A., Guan, Y. and Waagepetersen, R. (2013) Decomposition of variance for spatial Cox processes. *Scandinavian Journal of Statistics* **40**, 119-137.

Møller, J., Syversveen, A. and Waagepetersen, R. (1998) Log Gaussian Cox Processes. *Scandinavian Journal of Statistics* **25**, 451–482.

Waagepetersen, R. (2007) An estimating function approach to inference for inhomogeneous Neyman-Scott processes. *Biometrics* **63**, 252–258.

See Also

`rpoispp`, `rMatClust`, `rThomas`, `rVarGamma`, `rNeymanScott`, `rGaussPoisson`.

For fitting the model, see `kppm`, `clusterfit`.

Examples

```
# homogeneous
X <- rCauchy(30, 0.01, 5)
# inhomogeneous
ff <- function(x,y){ exp(2 - 3 * abs(x)) }
Z <- as.im(ff, W= owin())
Y <- rCauchy(50, 0.01, Z)
YY <- rCauchy(ff, 0.01, 5)
```

`rcell` *Simulate Baddeley-Silverman Cell Process*

Description

Generates a random point pattern, a simulated realisation of the Baddeley-Silverman cell process model.

Usage

```
rcell(win=square(1), nx=NULL, ny=nx, ..., dx=NULL, dy=dx,
      N=10, nsim=1, drop=TRUE)
```

Arguments

win	A window. An object of class owin, or data in any format acceptable to as.owin().
nx	Number of columns of cells in the window. Incompatible with dx.
ny	Number of rows of cells in the window. Incompatible with dy.
...	Ignored.
dx	Width of the cells. Incompatible with nx.
dy	Height of the cells. Incompatible with ny.
N	Integer. Distributional parameter: the maximum number of random points in each cell. Passed to rcellnumber.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a simulated realisation of the "cell process" (Baddeley and Silverman, 1984), a random point process with the same second-order properties as the uniform Poisson process. In particular, the K function of this process is identical to the K function of the uniform Poisson process (aka Complete Spatial Randomness). The same holds for the pair correlation function and all other second-order properties. The cell process is a counterexample to the claim that the K function completely characterises a point pattern.

A cell process is generated by dividing space into equal rectangular tiles. In each tile, a random number of random points is placed. By default, there are either 0, 1 or 10 points, with probabilities $1/10$, $8/9$ and $1/90$ respectively. The points within a tile are independent and uniformly distributed in that tile, and the numbers of points in different tiles are independent random integers.

The tile width is determined either by the number of columns nx or by the horizontal spacing dx. The tile height is determined either by the number of rows ny or by the vertical spacing dy. The cell process is then generated in these tiles. The random numbers of points are generated by rcellnumber.

Some of the resulting random points may lie outside the window win: if they do, they are deleted. The result is a point pattern inside the window win.

Value

A point pattern (an object of class "ppp") if nsim=1, or a list of point patterns if nsim > 1.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

References

Baddeley, A.J. and Silverman, B.W. (1984) A cautionary example on the use of second-order methods for analyzing point patterns. *Biometrics* **40**, 1089-1094.

See Also

rcellnumber, rstrat, rsyst, runifpoint, Kest

Examples

```
X <- rcell(nx=15)
plot(X)
if(require(spatstat.explore)) {
   plot(Kest(X))
}
```

---

rcellnumber *Generate Random Numbers of Points for Cell Process*

Description

Generates random integers for the Baddeley-Silverman counterexample.

Usage

```
rcellnumber(n, N = 10, mu=1)
```

Arguments

n	Number of random integers to be generated.
N	Distributional parameter: the largest possible value (when mu <= 1). An integer greater than 1.
mu	Mean of the distribution (equals the variance). Any positive real number.

Details

If mu = 1 (the default), this function generates random integers which have mean and variance equal to 1, but which do not have a Poisson distribution. The random integers take the values 0, 1 and N with probabilities $1/N$, $(N-2)/(N-1)$ and $1/(N(N-1))$ respectively. See Baddeley and Silverman (1984).

If mu is another positive number, the random integers will have mean and variance equal to mu. They are obtained by generating the one-dimensional counterpart of the cell process and counting the number of points in the interval from 0 to mu. The maximum possible value of each random integer is N * ceiling(mu).

Value

An integer vector of length n.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

References

Baddeley, A.J. and Silverman, B.W. (1984) A cautionary example on the use of second-order methods for analyzing point patterns. *Biometrics* **40**, 1089-1094.

See Also

rcell

Examples

```
rcellnumber(30, 3)
```

`rclusterBKBC` *Simulate Cluster Process using Brix-Kendall Algorithm or Modifications*

Description

Generates simulated realisations of a stationary Neyman-Scott cluster point process, using the Brix-Kendall (2002) algorithm or various modifications proposed by Baddeley and Chang (2023). For advanced research use.

Usage

```
rclusterBKBC(clusters="Thomas",
   kappa, mu, scale,
   ...,
   W = unit.square(),
   nsim = 1, drop = TRUE,
   best = FALSE,
   external = c("BK", "superBK", "border"),
   internal = c("dominating", "naive"),
   inflate = 1,
   psmall = 1e-04,
   use.inverse=TRUE,
   use.special=TRUE,
   integralmethod=c("quadrature", "trapezoid"),
   verbose = TRUE, warn=TRUE)
```

Arguments

`clusters`	Character string (partially matched) specifying the cluster process. Current options include `"Thomas"`, `"MatClust"`, `"Cauchy"` and `"VarGamma"`.
`kappa`	Intensity of the parent process. A nonnegative number.
`mu`	Mean number of offspring per parent. A nonnegative number.
`scale`	Cluster scale. Interpretation depends on the model.

...	Additional arguments controlling the shape of the cluster kernel, if any.
W	Window in which the simulation should be generated. An object of class "owin".
nsim	The number of simulated point patterns to be generated. A positive integer.
drop	Logical value. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
best	Logical value. If best=TRUE, the code will choose the fastest algorithm. If best=FALSE (the default), the algorithm will be specified by the other arguments external and internal. See Details.
external	Algorithm to be used to generate parent points which lie outside the bounding window. See Details.
internal	Algorithm to be used to generate parent points which lie inside the bounding window. See Details.
inflate	Numerical value determining the position of the bounding window. See Details.
psmall	Threshold of small probability for use in the algorithm.
use.inverse	Logical value specifying whether to compute the inverse function analytically, if possible (use.inverse=TRUE, the default) or by numerical root-finding (use.inverse=FALSE). This is mainly for checking validity of code.
use.special	Logical value specifying whether to use efficient special code (if available) to generate the simulations (use.special=TRUE, the default) or to use generic code (use.special=FALSE). This is mainly for checking validity of code.
integralmethod	Character string (partially matched) specifying how to perform numerical computation of integrals when required. This argument is passed to indefinteg. The default integralmethod="quadrature" is accurate but can be slow. Faster, but possibly less accurate, integration can be performed by setting integralmethod="trapezoid".
verbose	Logical value specifying whether to print detailed information about the simulation algorithm during execution.
warn	Logical value specifying whether to issue a warning if the number of random proposal points is very large.

Details

This function is intended for advanced research use. It implements the algorithm of Brix and Kendall (2002) for generating simulated realisations of a stationary Neyman-Scott process, and various modifications of this algorithm proposed in Baddeley and Chang (2023). It is an alternative to rNeymanScott.

The function supports the following models:

- clusters="Thomas": the (modified) Thomas cluster process which can also be simulated by rThomas.
- clusters="MatClust": the Matérn cluster process which can also be simulated by rMatClust.
- clusters="Cauchy": the Cauchy cluster process which can also be simulated by rCauchy.
- clusters="VarGamma": the variance-gamma cluster process which can also be simulated by rVarGamma.
- any other Poisson cluster process models that may be recognised by kppm.

By default, the code executes the original Brix-Kendall algorithm described in Sections 2.3 and 3.1 of Brix and Kendall (2002).

Modifications of this algorithm, proposed in Baddeley and Chang (2023), can be selected using the arguments `external` and `internal`, or `best`.

If `best=TRUE`, the code will choose the algorithm that would run fastest with the given parameters. If `best=FALSE` (the default), the choice of algorithm is determined by the arguments `external` and `internal`.

First the window `W` is enclosed in a disc `D` and Monte Carlo proposal densities are defined with reference to `D` as described in Brix and Kendall (2002). Then `D` is inflated by the scale factor `inflate` to produce a larger disc `E` (by default `inflate=1` implying `E=D`). Then the parent points of the clusters are generated, possibly using different mechanisms inside and outside `E`.

The argument `external` determines the algorithm for generating parent points outside `E`.

- If `external="BK"` (the default), proposed parents outside `E` will be generated from a dominating point process as described in Section 3.1 of Brix and Kendall (2002). These points will be thinned to obtain the correct intensity of parent points. For each accepted parent, offspring points are generated inside `D`, subject to the condition that the parent has at least one offspring inside `D`. Offspring points are subsequently clipped to the true window `W`.
- If `external="superBK"`, proposed parents will initially be generated from a process that dominates the dominating point process as described in Baddeley and Chang (2023). These proposals will then be thinned to obtain the correct intensity of the dominating process, then thinned again to obtain the correct intensity of parent points. This procedure reduces computation time when `scale` is large. For each accepted parent, offspring points are generated inside `D`, subject to the condition that the parent has at least one offspring inside `D`. Offspring points are subsequently clipped to the true window `W`.
- If `external="border"` then proposed parents will be generated with uniform intensity in a border region surrounding the disc `D`. For each proposed parent, offspring points are generated in the entire plane according to the cluster offspring distribution, without any restriction. Offspring points are subsequently clipped to the true window `W`. This is the technique currently used in `rNeymanScott`.

The argument `internal` determines the algorithm for generating proposed parent points inside `E`.

- If `internal="dominating"`, parent points in `E` are generated according to the dominating point process described in Sections 2.3 and 3.1 of Brix and Kendall (2002), and then thinned to obtain the correct intensity of parent points. For each accepted parent, offspring points are generated inside `D`, subject to the condition that the parent has at least one offspring inside `D`. Offspring points are subsequently clipped to the true window `W`.
- If `internal="naive"`, parent points in `E` are generated with uniform intensity inside `E` and are not thinned. For each proposed parent, offspring points are generated in the entire plane according to the cluster offspring distribution, without any restriction. Offspring points are subsequently clipped to the true window `W`. This is the technique currently used in `rNeymanScott`.

If `warn=TRUE`, then a warning will be issued if the number of random proposal points (proposed parents and proposed offspring) is very large. The threshold is `spatstat.options("huge.npoints")`. This warning has no consequences, but it helps to trap a number of common problems.

Value

A point pattern, or a list of point patterns.

If nsim=1 and drop=TRUE, the result is a point pattern (an object of class "ppp").

Otherwise, the result is a list of nsim point patterns, and also belongs to the class "solist".

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Ya-Mei Chang <yamei628@gmail.com>.

References

Baddeley, A. and Chang, Y.-M. (2023) Robust algorithms for simulating cluster point processes. *Journal of Statistical Computation and Simulation* **93**, 1950–1975.

Brix, A. and Kendall, W.S. (2002) Simulation of cluster point processes without edge effects. *Advances in Applied Probability* **34**, 267–280.

See Also

rNeymanScott, rMatClust, rThomas, rCauchy, rVarGamma

Examples

```
Y <- rclusterBKBC("Thomas", 10,5,0.2)
Y
Z <- rclusterBKBC("VarGamma", 10,5,0.2,
        nu=-1/4,
        internal="naive", external="super",
        verbose=FALSE)
```

---

rDGS *Perfect Simulation of the Diggle-Gates-Stibbard Process*

Description

Generate a random pattern of points, a simulated realisation of the Diggle-Gates-Stibbard process, using a perfect simulation algorithm.

Usage

```
rDGS(beta, rho, W = owin(), expand=TRUE, nsim=1, drop=TRUE)
```

Arguments

beta	intensity parameter (a positive number).
rho	interaction range (a non-negative number).
W	window (object of class "owin") in which to generate the random pattern.
expand	Logical. If FALSE, simulation is performed in the window W, which must be rectangular. If TRUE (the default), simulation is performed on a larger window, and the result is clipped to the original window W. Alternatively expand can be an object of class "rmhexpand" (see rmhexpand) determining the expansion method.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a realisation of the Diggle-Gates-Stibbard point process in the window W using a 'perfect simulation' algorithm.

Diggle, Gates and Stibbard (1987) proposed a pairwise interaction point process in which each pair of points separated by a distance d contributes a factor $e(d)$ to the probability density, where

$$e(d) = \sin^2\left(\frac{\pi d}{2\rho}\right)$$

for $d < \rho$, and $e(d)$ is equal to 1 for $d \geq \rho$.

The simulation algorithm used to generate the point pattern is 'dominated coupling from the past' as implemented by Berthelsen and Møller (2002, 2003). This is a 'perfect simulation' or 'exact simulation' algorithm, so called because the output of the algorithm is guaranteed to have the correct probability distribution exactly (unlike the Metropolis-Hastings algorithm used in rmh, whose output is only approximately correct).

There is a tiny chance that the algorithm will run out of space before it has terminated. If this occurs, an error message will be generated.

Value

If nsim = 1, a point pattern (object of class "ppp"). If nsim > 1, a list of point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, based on original code for the Strauss process by Kasper Klitgaard Berthelsen.

References

Berthelsen, K.K. and Møller, J. (2002) A primer on perfect simulation for spatial point processes. *Bulletin of the Brazilian Mathematical Society* 33, 351-367.

Berthelsen, K.K. and Møller, J. (2003) Likelihood and non-parametric Bayesian MCMC inference for spatial point processes based on perfect simulation and path sampling. *Scandinavian Journal of Statistics* 30, 549-564.

Diggle, P.J., Gates, D.J., and Stibbard, A. (1987) A nonparametric estimator for pairwise-interaction point processes. Biometrika **74**, 763 – 770. *Scandinavian Journal of Statistics* **21**, 359–373.

Møller, J. and Waagepetersen, R. (2003). *Statistical Inference and Simulation for Spatial Point Processes.* Chapman and Hall/CRC.

See Also

rmh, DiggleGatesStibbard.

rStrauss, rHardcore, rStraussHard, rDiggleGratton, rPenttinen.

Examples

```
X <- rDGS(50, 0.05)
Z <- rDGS(50, 0.03, nsim=2)
```

---

rdiffuse *Perturb the Points in a Point Pattern According to a Diffusion Process*

Description

Given a spatial point pattern inside a window, allow each point of the pattern to undergo random spatial diffusion inside the window for a specified amount of time, and return the final position of each point.

Usage

```
rdiffuse(X, sigma, ...)

## S3 method for class 'ppp'
rdiffuse(X, sigma, ..., nsim=1, drop=TRUE,
    connect = 8, method = c("C", "interpreted"), unround = TRUE)
```

Arguments

X	Point pattern (object of class "ppp") specifying the initial position of each point.
sigma	Equivalent standard deviation of the final position of each point, if the window were unbounded. A single positive number, or a pixel image.
...	Arguments passed to as.mask controlling the spatial resolution.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
connect	Pixel grid connectivity (4 or 8).

method	For testing purposes only. Character string (partially matched) specifying whether to use a C language implementation or an interpreted R implementation.
unround	Logical value specifying whether the final positions of the points should be altered slightly by adding a small random error to the coordinates using rUnround. This reverses the effect of the discretisation. If unround=FALSE, all the final positions are exactly at the centre of a pixel.

Details

The spatial locations of the points of X are first discretised onto a pixel grid, with resolution specified by the arguments .... Each point then executes a random walk on the pixel grid, independently of other points. The final location of each point is returned.

Value

A point pattern or a list of point patterns. Each point pattern has the same window as X and the same number of points as X.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

densityHeat.ppp

Examples

```
plot(solist(original=cells, diffused=rdiffuse(cells, 0.05)))
```

---

rDiggleGratton *Perfect Simulation of the Diggle-Gratton Process*

Description

Generate a random pattern of points, a simulated realisation of the Diggle-Gratton process, using a perfect simulation algorithm.

Usage

```
rDiggleGratton(beta, delta, rho, kappa=1, W = owin(),
               expand=TRUE, nsim=1, drop=TRUE)
```

Arguments

beta	intensity parameter (a positive number).
delta	hard core distance (a non-negative number).
rho	interaction range (a number greater than delta).
kappa	interaction exponent (a non-negative number).
W	window (object of class "owin") in which to generate the random pattern. Currently this must be a rectangular window.
expand	Logical. If FALSE, simulation is performed in the window W, which must be rectangular. If TRUE (the default), simulation is performed on a larger window, and the result is clipped to the original window W. Alternatively expand can be an object of class "rmhexpand" (see rmhexpand) determining the expansion method.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a realisation of the Diggle-Gratton point process in the window W using a 'perfect simulation' algorithm.

Diggle and Gratton (1984, pages 208-210) introduced the pairwise interaction point process with pair potential $h(t)$ of the form

$$h(t) = \left(\frac{t-\delta}{\rho-\delta}\right)^{\kappa} \qquad \text{if } \delta \le t \le \rho$$

with $h(t) = 0$ for $t < \delta$ and $h(t) = 1$ for $t > \rho$. Here δ, ρ and κ are parameters.

Note that we use the symbol κ where Diggle and Gratton (1984) use β, since in **spatstat** we reserve the symbol β for an intensity parameter.

The parameters must all be nonnegative, and must satisfy $\delta \le \rho$.

The simulation algorithm used to generate the point pattern is 'dominated coupling from the past' as implemented by Berthelsen and Møller (2002, 2003). This is a 'perfect simulation' or 'exact simulation' algorithm, so called because the output of the algorithm is guaranteed to have the correct probability distribution exactly (unlike the Metropolis-Hastings algorithm used in rmh, whose output is only approximately correct).

There is a tiny chance that the algorithm will run out of space before it has terminated. If this occurs, an error message will be generated.

Value

If nsim = 1, a point pattern (object of class "ppp"). If nsim > 1, a list of point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

based on original code for the Strauss process by Kasper Klitgaard Berthelsen.

References

Berthelsen, K.K. and Møller, J. (2002) A primer on perfect simulation for spatial point processes. *Bulletin of the Brazilian Mathematical Society* 33, 351-367.

Berthelsen, K.K. and Møller, J. (2003) Likelihood and non-parametric Bayesian MCMC inference for spatial point processes based on perfect simulation and path sampling. *Scandinavian Journal of Statistics* 30, 549-564.

Diggle, P.J. and Gratton, R.J. (1984) Monte Carlo methods of inference for implicit statistical models. *Journal of the Royal Statistical Society, series B* **46**, 193 – 212.

Møller, J. and Waagepetersen, R. (2003). *Statistical Inference and Simulation for Spatial Point Processes.* Chapman and Hall/CRC.

See Also

rmh, rStrauss, rHardcore, rStraussHard, rDGS, rPenttinen.

For fitting the model, see DiggleGratton.

Examples

```
X <- rDiggleGratton(50, 0.02, 0.07)
Z <- rDiggleGratton(50, 0.02, 0.07, 2, nsim=2)
```

---

reach *Interaction Distance of a Point Process Model*

Description

Computes the interaction distance of a point process model.

Usage

```
reach(x, ...)

## S3 method for class 'rmhmodel'
reach(x, ...)
```

Arguments

x	Either a fitted point process model (object of class "ppm"), an interpoint interaction (object of class "interact"), a fitted interpoint interaction (object of class "fii") or a point process model for simulation (object of class "rmhmodel").
...	Other arguments are ignored.

Details

The function reach computes the 'interaction distance' or 'interaction range' of a point process model.

The definition of the interaction distance depends on the type of point process model. This help page explains the interaction distance for a Gibbs point process. For other kinds of models, see reach.kppm and reach.dppm.

For a Gibbs point process model, the interaction distance is the shortest distance D such that any two points in the process which are separated by a distance greater than D do not interact with each other.

For example, the interaction range of a Strauss process (see Strauss or rStrauss) with parameters β, γ, r is equal to r, unless $\gamma = 1$ in which case the model is Poisson and the interaction range is 0. The interaction range of a Poisson process is zero. The interaction range of the Ord threshold process (see OrdThresh) is infinite, since two points *may* interact at any distance apart.

The function reach is generic, with methods for the case where x is

- a fitted point process model (object of class "ppm", usually obtained from the model-fitting function ppm);
- an interpoint interaction structure (object of class "interact")
- a fitted interpoint interaction (object of class "fii")
- a point process model for simulation (object of class "rmhmodel"), usually obtained from rmhmodel.

Value

The interaction distance, or NA if this cannot be computed from the information given.

Other types of models

Methods for reach are also defined for point process models of class "kppm" and "dppm". Their technical definition is different from this one. See reach.kppm and reach.dppm.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

reach.ppm

rmhmodel

See reach.kppm and reach.dppm for other types of point process models.

Examples

```
reach(rmhmodel(cif='poisson', par=list(beta=100)))
# returns 0

reach(rmhmodel(cif='strauss', par=list(beta=100, gamma=0.1, r=7)))
```

```
# returns 7

reach(rmhmodel(cif='sftcr', par=list(beta=100, sigma=1, kappa=0.7)))
# returns Inf

reach(rmhmodel(cif='multihard',
               par=list(beta=c(10,10), hradii=matrix(c(1,3,3,1),2,2))))
# returns 3
```

`recipEnzpois` *First Reciprocal Moment of the Truncated Poisson Distribution*

Description

Computes the first reciprocal moment (first negative moment) of the truncated Poisson distribution (the Poisson distribution conditioned to have a nonzero value).

Usage

```
recipEnzpois(mu, exact=TRUE)
```

Arguments

`mu`	The mean of the original Poisson distribution. A single positive numeric value, or a vector of positive numbers.
`exact`	Logical value specifying whether to use the exact analytic formula if possible.

Details

This function calculates the expected value of $1/N$ given $N > 0$, where N is a Poisson random variable with mean μ.

If the library **gsl** is loaded, and if `exact=TRUE` (the default), then the calculation uses the exact analytic formula

$$\nu = \frac{e^{-\mu}}{1 - e^{-\mu}} \left(Ei(\mu) - \log \mu - \gamma \right)$$

(see e.g. Grab and Savage, 1954) where ν is the desired reciprocal moment, and

$$Ei(x) = \int_{-\infty}^{x} te^{-t} dt$$

is the first exponential integral, and $\gamma \approx 0.577$ is the Euler-Mascheroni constant.

If **gsl** is not loaded, or if `exact=FALSE` is specified, the value is computed approximately (and more slowly) by summing over the possible values of N up to a finite limit.

Value

A single numerical value or a numeric vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

References

Grab, E.L. and Savage, I.R. (1954) Tables of the expected value of 1/X for positive Bernoulli and Poisson variables. *Journal of the American Statistical Association* **49**, 169–177.

See Also

rpoisnonzero

Examples

```
if(require(gsl)) {
  v <- recipEnzpois(10)
  print(v)
}
recipEnzpois(10, exact=FALSE)
```

---

rGaussPoisson *Simulate Gauss-Poisson Process*

Description

Generate a random point pattern, a simulated realisation of the Gauss-Poisson Process.

Usage

```
rGaussPoisson(kappa, r, p2, win = owin(c(0,1),c(0,1)),
              ..., nsim=1, drop=TRUE)
```

Arguments

kappa	Intensity of the Poisson process of cluster centres. A single positive number, a function, or a pixel image.
r	Diameter of each cluster that consists of exactly 2 points.
p2	Probability that a cluster contains exactly 2 points.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin.
...	Ignored.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This algorithm generates a realisation of the Gauss-Poisson point process inside the window `win`. The process is constructed by first generating a Poisson point process of parent points with intensity `kappa`. Then each parent point is either retained (with probability `1 - p2`) or replaced by a pair of points at a fixed distance `r` apart (with probability `p2`). In the case of clusters of 2 points, the line joining the two points has uniform random orientation.

In this implementation, parent points are not restricted to lie in the window; the parent process is effectively the uniform Poisson process on the infinite plane.

Value

A point pattern (an object of class `"ppp"`) if `nsim=1`, or a list of point patterns if `nsim > 1`.

Additionally, some intermediate results of the simulation are returned as attributes of the point pattern. See `rNeymanScott`.

Author(s)

Adrian Baddeley `<Adrian.Baddeley@curtin.edu.au>` and Rolf Turner `<rolfturner@posteo.net>`

See Also

`rpoispp`, `rThomas`, `rMatClust`, `rNeymanScott`

Examples

```
pp <- rGaussPoisson(30, 0.07, 0.5)
```

`rGRFgauss` *Simulate a Gaussian Random Field*

Description

Generate a realisation of a Gaussian Random Field with given first and second moments.

Usage

```
rGRFgauss(W = owin(), mu = 0, var = 1, scale,
          ..., nsim = 1, drop = TRUE)
rGRFexpo(W = owin(), mu = 0, var = 1, scale,
          ..., nsim = 1, drop = TRUE)
rGRFstable(W = owin(), mu = 0, var = 1, scale, alpha,
           ..., nsim = 1, drop = TRUE)
rGRFgencauchy(W = owin(), mu = 0, var = 1, scale, alpha, beta,
           ..., nsim = 1, drop = TRUE)
rGRFmatern(W = owin(), mu = 0, var = 1, scale, nu,
           ..., nsim = 1, drop = TRUE)
```

Arguments

W	Window (object of class "owin") in which to generate the simulated random field.
mu	The mean of the random field. A single numeric value, or a function(x, y, ...), or a pixel image (object of class "im").
var	Variance of the random field. A single positive number.
scale	Spatial scale parameter h. A single positive number. See Details.
alpha, beta, nu	Additional parameters for specific models. See Details.
...	Arguments passed to as.mask to determine the pixel resolution.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

These functions generate simulated realisations of a Gaussian random field.

The mean $E[Z(u)]$ of the Gaussian random field value $Z(u)$ at any location u is specified by the argument mu, which may be a constant, a function(x,y,...), or a pixel image.

The variance $V[Z(u)]$ of the Gaussian random field value is specified by the argument var, which should be a single positive numerical value.

The correlation $C(u-v) = C(Z(u), Z(v))$ between the values at two locations u and v depends on the distance $r = \|u-v\|$ as follows:

rGRFexpo the exponential covariance function

$$C(r) = \sigma^2 \exp(-r/h)$$

where σ^2 is the variance parameter var, and h is the scale parameter scale.

rGRFgauss the Gaussian covariance function

$$C(r) = \sigma^2 \exp(-(r/h)^2)$$

where σ^2 is the variance parameter var, and h is the scale parameter scale.

rGRFstable the stable covariance function

$$C(r) = \sigma^2 \exp(-(r/h)^\alpha)$$

where σ^2 is the variance parameter var, h is the scale parameter scale, and α is the shape parameter alpha.

rGRFgencauchy the generalised Cauchy covariance function

$$C(r) = \sigma^2 (1 + (x/h)^\alpha)^{-\beta/\alpha}$$

where σ^2 is the variance parameter var, h is the scale parameter scale, and α and β are the shape parameters alpha and beta.

`rGRFmatern` the Whittle-Matérn covariance function

$$C(r) = \sigma^2 \frac{1}{2^{\nu-1}\Gamma(\nu)} (\sqrt{2\nu}\, r/h)^\nu K_\nu(\sqrt{2\nu}\, r/h)$$

where σ^2 is the variance parameter `var`, h is the scale parameter `scale`, and ν is the shape parameter `nu`.

The algorithm generates `nsim` simulated realisations of the random field using the circulant embedding technique (Davies and Bryant, 2013).

Value

If `nsim=1` and `drop=TRUE`, a pixel image (object of class `"im"`). Otherwise, a list of pixel images.

Author(s)

Tilman Davies <Tilman.Davies@otago.ac.nz> and David Bryant. Modified by Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

References

Davies, T.M. and Bryant, D. (2013) On Circulant Embedding for Gaussian Random Fields in R. *Journal of Statistical Software* 55, issue 9 doi:10.18637/jss.v055.i09

See Also

`rLGCP`

Examples

```
plot(rGRFgauss(scale=0.1))
```

---

rHardcore *Perfect Simulation of the Hardcore Process*

Description

Generate a random pattern of points, a simulated realisation of the Hardcore process, using a perfect simulation algorithm.

Usage

```
rHardcore(beta, R = 0, W = owin(), expand=TRUE, nsim=1, drop=TRUE)
```

Arguments

beta	intensity parameter (a positive number).
R	hard core distance (a non-negative number).
W	window (object of class "owin") in which to generate the random pattern. Currently this must be a rectangular window.
expand	Logical. If FALSE, simulation is performed in the window W, which must be rectangular. If TRUE (the default), simulation is performed on a larger window, and the result is clipped to the original window W. Alternatively expand can be an object of class "rmhexpand" (see rmhexpand) determining the expansion method.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a realisation of the Hardcore point process in the window W using a 'perfect simulation' algorithm.

The Hardcore process is a model for strong spatial inhibition. Two points of the process are forbidden to lie closer than R units apart. The Hardcore process is the special case of the Strauss process (see rStrauss) with interaction parameter γ equal to zero.

The simulation algorithm used to generate the point pattern is 'dominated coupling from the past' as implemented by Berthelsen and Møller (2002, 2003). This is a 'perfect simulation' or 'exact simulation' algorithm, so called because the output of the algorithm is guaranteed to have the correct probability distribution exactly (unlike the Metropolis-Hastings algorithm used in rmh, whose output is only approximately correct).

There is a tiny chance that the algorithm will run out of space before it has terminated. If this occurs, an error message will be generated.

Value

If nsim = 1, a point pattern (object of class "ppp"). If nsim > 1, a list of point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, based on original code for the Strauss process by Kasper Klitgaard Berthelsen.

References

Berthelsen, K.K. and Møller, J. (2002) A primer on perfect simulation for spatial point processes. *Bulletin of the Brazilian Mathematical Society* 33, 351-367.

Berthelsen, K.K. and Møller, J. (2003) Likelihood and non-parametric Bayesian MCMC inference for spatial point processes based on perfect simulation and path sampling. *Scandinavian Journal of Statistics* 30, 549-564.

Møller, J. and Waagepetersen, R. (2003). *Statistical Inference and Simulation for Spatial Point Processes.* Chapman and Hall/CRC.

See Also

rmh, rStrauss, rStraussHard, rDiggleGratton. rDGS, rPenttinen.

For fitting the model, see Hardcore.

Examples

```
X <- rHardcore(0.05,1.5,square(50))
```

---

rjitter.psp *Random Perturbation of Line Segment Pattern*

Description

Randomly pertubs a spatial pattern of line segments by applying independent random displacements to the segment endpoints.

Usage

```
## S3 method for class 'psp'
rjitter(X, radius, ..., clip=TRUE, nsim=1, drop=TRUE)
```

Arguments

X	A point pattern on a linear network (object of class "psp").
radius	Scale of perturbations. A positive numerical value. Each point will be displaced by a random distance, with maximum displacement equal to this value.
...	Ignored.
clip	Logical value specifying what to do if segments cross the boundary of the window. See Details.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a spatial pattern of line segments (class "psp") rather than a list of length 1 containing this pattern.

Details

The function rjitter is generic. This function is the method for the class "psp" of line segment patterns.

Each of the endpoints of each segment in X will be subjected to an independent random displacement. The displacement vectors are uniformly distributed in a circle of radius radius. Marks attached to the segments will be unchanged.

If clip=TRUE (the default), segment endpoints are permitted to move to locations slightly outside the window of X, and the resulting segments will be clipped to the window. If clip=FALSE, segment endpoints are conditioned to fall inside the window.

If nsim=1 and drop=TRUE, the result is another spatial pattern of line segments (object of class "psp"). Otherwise, the result is a list of nsim line segment patterns.

Value

A spatial pattern of line segments (object of class "psp") or a list of such patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

rjitter for point patterns in two dimensions.

Examples

```
E <- edges(letterR)
Window(E) <- owin(c(1.9, 4.1), c(0.5, 3.5))
plot(rjitter(E, 0.1))
```

---

rknn *Theoretical Distribution of Nearest Neighbour Distance*

Description

Density, distribution function, quantile function and random generation for the random distance to the kth nearest neighbour in a Poisson point process in d dimensions.

Usage

```
dknn(x, k = 1, d = 2, lambda = 1)
pknn(q, k = 1, d = 2, lambda = 1)
qknn(p, k = 1, d = 2, lambda = 1)
rknn(n, k = 1, d = 2, lambda = 1)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations to be generated.
k	order of neighbour.
d	dimension of space.
lambda	intensity of Poisson point process.

Details

In a Poisson point process in d-dimensional space, let the random variable R be the distance from a fixed point to the k-th nearest random point, or the distance from a random point to the k-th nearest other random point.

Then R^d has a Gamma distribution with shape parameter k and rate $\lambda * \alpha$ where α is a constant (equal to the volume of the unit ball in d-dimensional space). See e.g. Cressie (1991, page 61).

These functions support calculation and simulation for the distribution of R.

Value

A numeric vector: dknn returns the probability density, pknn returns cumulative probabilities (distribution function), qknn returns quantiles, and rknn generates random deviates.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

References

Cressie, N.A.C. (1991) *Statistics for spatial data*. John Wiley and Sons, 1991.

Examples

```
x <- seq(0, 5, length=20)
densities <- dknn(x, k=3, d=2)
cdfvalues <- pknn(x, k=3, d=2)
randomvalues <- rknn(100, k=3, d=2)
deciles <- qknn((1:9)/10, k=3, d=2)
```

---

rlabel *Random Re-Labelling of Point Pattern*

Description

Randomly allocates marks to a point pattern, or permutes the existing marks, or resamples from the existing marks.

Usage

```
rlabel(X, labels=marks(X), permute=TRUE, group=NULL, ..., nsim=1, drop=TRUE)
```

Arguments

X	Point pattern (object of class "ppp", "lpp", "pp3" or "ppx") or line segment pattern (object of class "psp").
labels	Vector of values from which the new marks will be drawn at random. Defaults to the vector of existing marks.
permute	Logical value indicating whether to generate new marks by randomly permuting labels or by drawing a random sample with replacement.
group	Optional. A factor, or other data dividing the points into groups. Random relabelling will be performed separately within each group. See Details.
...	Additional arguments passed to cut.ppp to determine the grouping factor, when group is given.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This very simple function allocates random marks to an existing point pattern X. It is useful for hypothesis testing purposes. (The function can also be applied to line segment patterns.)

In the simplest case, the command rlabel(X) yields a point pattern obtained from X by randomly permuting the marks of the points.

If permute=TRUE, then labels should be a vector of length equal to the number of points in X. The result of rlabel will be a point pattern with locations given by X and marks given by a random permutation of labels (i.e. a random sample without replacement).

If permute=FALSE, then labels may be a vector of any length. The result of rlabel will be a point pattern with locations given by X and marks given by a random sample from labels (with replacement).

The argument group specifies that the points are divided into several different groups, and that the random labelling shall be performed separately on each group. The arguments group and ... are passed to cut.ppp to determine the grouping. Thus group could be a factor, or the name of a column of marks in X, or a tessellation, or a factor-valued pixel image, etc.

Value

If nsim = 1 and drop=TRUE, a marked point pattern (of the same class as X). If nsim > 1, a list of point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

marks<- to assign arbitrary marks.

Examples

```
amacrine

# Randomly permute the marks "on" and "off"
# Result always has 142 "off" and 152 "on"
Y <- rlabel(amacrine)

# randomly allocate marks "on" and "off"
# with probabilities p(off) = 0.48, p(on) = 0.52
Y <- rlabel(amacrine, permute=FALSE)

# randomly allocate marks "A" and "B" with equal probability
Y <- rlabel(cells, labels=factor(c("A", "B")), permute=FALSE)

# divide the window into tiles and
# randomly permute the marks within each tile
Z <- rlabel(amacrine, group=quadrats(Window(amacrine), 4, 3))
```

---

rLGCP *Simulate Log-Gaussian Cox Process*

Description

Generate a random point pattern, a realisation of the log-Gaussian Cox process.

Usage

```
rLGCP(model=c("exponential", "gauss", "stable", "gencauchy", "matern"),
      mu = 0, param = NULL,
      ...,
      win=NULL, saveLambda=TRUE, nsim=1, drop=TRUE,
      n.cond=NULL, w.cond=NULL)
```

Arguments

model	character string (partially matched) giving the name of a covariance model for the Gaussian random field.
mu	mean function of the Gaussian random field. Either a single number, a function(x,y, ...) or a pixel image (object of class "im").
param	List of parameters for the covariance. Standard arguments are var and scale.
...	Additional parameters for the covariance, or arguments passed to as.mask to determine the pixel resolution.
win	Window in which to simulate the pattern. An object of class "owin".
saveLambda	Logical. If TRUE (the default) then the simulated random intensity will also be saved, and returns as an attribute of the point pattern.
nsim	Number of simulated realisations to be generated.

drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
n.cond	Optional. Integer specifying a fixed number of points. See the section on *Conditional Simulation*.
w.cond	Optional. Conditioning region. A window (object of class "owin") specifying the region which must contain exactly n.cond points. See the section on *Conditional Simulation*.

Details

This function generates a realisation of a log-Gaussian Cox process (LGCP). This is a Cox point process in which the logarithm of the random intensity is a Gaussian random field with mean function μ and covariance function $c(r)$. Conditional on the random intensity, the point process is a Poisson process with this intensity.

The string model specifies the covariance function of the Gaussian random field, and the parameters of the covariance are determined by param and ....

All models recognise the parameters var for the variance at distance zero, and scale for the scale parameter. Some models require additional parameters which are listed below.

The available models are as follows:

model="exponential": the exponential covariance function

$$C(r) = \sigma^2 \exp(-r/h)$$

where σ^2 is the variance parameter var, and h is the scale parameter scale.

model="gauss": the Gaussian covariance function

$$C(r) = \sigma^2 \exp(-(r/h)^2)$$

where σ^2 is the variance parameter var, and h is the scale parameter scale.

model="stable": the stable covariance function

$$C(r) = \sigma^2 \exp(-(r/h)^\alpha)$$

where σ^2 is the variance parameter var, h is the scale parameter scale, and α is the shape parameter alpha. The parameter alpha must be given, either as a stand-alone argument, or as an entry in the list param.

model="gencauchy": the generalised Cauchy covariance function

$$C(r) = \sigma^2 (1 + (x/h)^\alpha)^{-\beta/\alpha}$$

where σ^2 is the variance parameter var, h is the scale parameter scale, and α and β are the shape parameters alpha and beta. The parameters alpha and beta must be given, either as stand-alone arguments, or as entries in the list param.

model="matern": the Whittle-Matérn covariance function

$$C(r) = \sigma^2 \frac{1}{2^{\nu-1}\Gamma(\nu)} (\sqrt{2\nu}\, r/h)^\nu K_\nu(\sqrt{2\nu}\, r/h)$$

where σ^2 is the variance parameter var, h is the scale parameter scale, and ν is the shape parameter nu. The parameter nu must be given, either as a stand-alone argument, or as an entry in the list param.

The algorithm uses the circulant embedding technique to generate values of a Gaussian random field, with the specified mean function mu and the covariance specified by the arguments model and param, on the points of a regular grid. The exponential of this random field is taken as the intensity of a Poisson point process, and a realisation of the Poisson process is then generated by the function rpoispp in the **spatstat.random** package.

If the simulation window win is missing or NULL, then it defaults to Window(mu) if mu is a pixel image, and it defaults to the unit square otherwise.

The LGCP model can be fitted to data using kppm.

Value

A point pattern (object of class "ppp") or a list of point patterns.

Additionally, the simulated intensity function for each point pattern is returned as an attribute "Lambda" of the point pattern, if saveLambda=TRUE.

Conditional Simulation

If n.cond is specified, it should be a single integer. Simulation will be conditional on the event that the pattern contains exactly n.cond points (or contains exactly n.cond points inside the region w.cond if it is given).

Conditional simulation uses the rejection algorithm described in Section 6.2 of Moller, Syversveen and Waagepetersen (1998). There is a maximum number of proposals which will be attempted. Consequently the return value may contain fewer than nsim point patterns.

Warning: new implementation

The simulation algorithm for rLGCP has been completely re-written in **spatstat.random** version 3.2-0 to avoid depending on the package **RandomFields** which is now defunct (and is sadly missed).

It is no longer possible to replicate results that were obtained using rLGCP in previous versions of **spatstat.random**.

The current code is a new implementation and should be considered vulnerable to new bugs.

Author(s)

Abdollah Jalilian and Rasmus Waagepetersen. Modified by Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

References

Møller, J., Syversveen, A. and Waagepetersen, R. (1998) Log Gaussian Cox Processes. *Scandinavian Journal of Statistics* **25**, 451–482.

See Also

rpoispp, rMatClust, rGaussPoisson, rNeymanScott.

For fitting the model, see kppm, lgcp.estK.

For generating Gaussian Random Fields, see rGRFgauss.

Examples

```
online <- interactive()

# homogeneous LGCP with exponential covariance function
X <- rLGCP("exp", 3, var=0.2, scale=.1)

# inhomogeneous LGCP with Gaussian covariance function
m <- as.im(function(x, y){5 - 1.5 * (x - 0.5)^2 + 2 * (y - 0.5)^2}, W=owin())
X <- rLGCP("gauss", m, var=0.15, scale =0.1)

if(online) {
 plot(attr(X, "Lambda"))
 points(X)
}

# inhomogeneous LGCP with Matern covariance function
X <- rLGCP("matern", function(x, y){ 1 - 0.4 * x},
           var=2, scale=0.7, nu=0.5,
           win = owin(c(0, 10), c(0, 10)))
if(online) plot(X)
```

---

rMatClust *Simulate Matern Cluster Process*

Description

Generate a random point pattern, a simulated realisation of the Matérn Cluster Process.

Usage

```
rMatClust(kappa, scale, mu, win = square(1),
          nsim=1, drop=TRUE, ...,
          n.cond=NULL, w.cond=NULL,
          algorithm=c("BKBC", "naive"),
          nonempty=TRUE,
          poisthresh=1e-6, saveparents=FALSE, saveLambda=FALSE,
          kappamax=NULL, mumax=NULL, LambdaOnly=FALSE)
```

Arguments

kappa	Intensity of the Poisson process of cluster centres. A single positive number, a function, or a pixel image.
scale	Radius of the clusters. A single positive number.
mu	Mean number of points per cluster (a single positive number) or reference intensity for the cluster points (a function or a pixel image).
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin.

nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
...	Passed to clusterfield to control the image resolution when saveLambda=TRUE.
n.cond	Optional. Integer specifying a fixed number of points. See the section on *Conditional Simulation*.
w.cond	Optional. Conditioning region. A window (object of class "owin") specifying the region which must contain exactly n.cond points. See the section on *Conditional Simulation*.
algorithm	String (partially matched) specifying the simulation algorithm. See Details.
nonempty	Logical. If TRUE (the default), a more efficient algorithm is used, in which parents are generated conditionally on having at least one offspring point in the window. If FALSE, parents are generated even if they have no offspring in the window. The default is recommended unless you need to simulate all the parent points for some other purpose.
poisthresh	Numerical threshold below which the model will be treated as a Poisson process. See Details.
saveparents	Logical value indicating whether to save the locations of the parent points as an attribute.
saveLambda	Logical. If TRUE then the random intensity corresponding to the simulated parent points will also be calculated and saved, and returns as an attribute of the point pattern.
kappamax	Optional. Numerical value which is an upper bound for the values of kappa, when kappa is a pixel image or a function.
mumax	Optional. Numerical value which is an upper bound for the values of mu, when mu is a pixel image or a function.
LambdaOnly	Logical value specifying whether to return only the random intensity, rather than the point pattern.

Details

This algorithm generates a realisation of Matérn's cluster process, a special case of the Neyman-Scott process, inside the window win.

In the simplest case, where kappa and mu are single numbers, the cluster process is formed by first generating a uniform Poisson point process of "parent" points with intensity kappa. Then each parent point is replaced by a random cluster of "offspring" points, the number of points per cluster being Poisson (mu) distributed, and their positions being placed and uniformly inside a disc of radius scale centred on the parent point. The resulting point pattern is a realisation of the classical "stationary Matérn cluster process". This point process has intensity kappa * mu.

The algorithm can also generate spatially inhomogeneous versions of the Matérn cluster process:

- The parent points can be spatially inhomogeneous. If the argument kappa is a function(x,y) or a pixel image (object of class "im"), then it is taken as specifying the intensity function of an inhomogeneous Poisson process that generates the parent points.

- The offspring points can be inhomogeneous. If the argument mu is a function(x,y) or a pixel image (object of class "im"), then it is interpreted as the reference density for offspring points, in the sense of Waagepetersen (2007). For a given parent point, the offspring constitute a Poisson process with intensity function equal to mu/(pi * scale^2) inside the disc of radius scale centred on the parent point, and zero intensity outside this disc. Equivalently we first generate, for each parent point, a Poisson (M) random number of offspring (where M is the maximum value of mu) placed independently and uniformly in the disc of radius scale centred on the parent location, and then randomly thin the offspring points, with retention probability mu/M.
- Both the parent points and the offspring points can be inhomogeneous, as described above.

The intensity of the Matérn cluster process is kappa * mu if either kappa or mu is a single number. In the general case the intensity is an integral involving kappa, mu and scale.

Value

A point pattern (object of class "ppp") or a list of point patterns.

Additionally, some intermediate results of the simulation are returned as attributes of this point pattern (see rNeymanScott). Furthermore, the simulated intensity function is returned as an attribute "Lambda", if saveLambda=TRUE.

If LambdaOnly=TRUE the result is a pixel image (object of class "im") or a list of pixel images.

Simulation Algorithm

Two simulation algorithms are implemented.

- The *naive* algorithm generates the cluster process by directly following the description given above. First the window win is expanded by a distance equal to scale. Then the parent points are generated in the expanded window according to a Poisson process with intensity kappa. Then each parent point is replaced by a finite cluster of offspring points as described above. The naive algorithm is used if algorithm="naive" or if nonempty=FALSE.
- The *BKBC* algorithm, proposed by Baddeley and Chang (2023), is a modification of the algorithm of Brix and Kendall (2002). Parents are generated in the infinite plane, subject to the condition that they have at least one offspring point inside the window win. The BKBC algorithm is used when algorithm="BKBC" (the default) and nonempty=TRUE (the default).

The naive algorithm becomes very slow when scale is large, while the BKBC algorithm is uniformly fast (Baddeley and Chang, 2023).

If saveparents=TRUE, then the simulated point pattern will have an attribute "parents" containing the coordinates of the parent points, and an attribute "parentid" mapping each offspring point to its parent.

If nonempty=TRUE (the default), then parents are generated subject to the condition that they have at least one offspring point in the window win. nonempty=FALSE, then parents without offspring will be included; this option is not available in the *BKBC* algorithm.

Note that if kappa is a pixel image, its domain must be larger than the window win. This is because an offspring point inside win could have its parent point lying outside win. In order to allow this, the naive simulation algorithm first expands the original window win by a distance equal to scale

and generates the Poisson process of parent points on this larger window. If kappa is a pixel image, its domain must contain this larger window.

If the pair correlation function of the model is very close to that of a Poisson process, with maximum deviation less than poisthresh, then the model is approximately a Poisson process. This is detected by the naive algorithm which then simulates a Poisson process with intensity kappa * mu, using rpoispp. This avoids computations that would otherwise require huge amounts of memory.

Fitting cluster models to data

The Matérn cluster process model with homogeneous parents (i.e. where kappa is a single number) where the offspring are either homogeneous or inhomogeneous (mu is a single number, a function or pixel image) can be fitted to point pattern data using kppm, or fitted to the inhomogeneous K function using matclust.estK or matclust.estpcf.

Currently **spatstat** does not support fitting the Matérn cluster process model with inhomogeneous parents.

A fitted Matérn cluster process model can be simulated automatically using simulate.kppm (which invokes rMatClust to perform the simulation).

Conditional Simulation

If n.cond is specified, it should be a single integer. Simulation will be conditional on the event that the pattern contains exactly n.cond points (or contains exactly n.cond points inside the region w.cond if it is given).

Conditional simulation uses the rejection algorithm described in Section 6.2 of Moller, Syversveen and Waagepetersen (1998). There is a maximum number of proposals which will be attempted. Consequently the return value may contain fewer than nsim point patterns.

The current algorithm for conditional simulation ignores the argument saveparents and does not save the parent points.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Ya-Mei Chang <yamei628@gmail.com> and Rolf Turner <rolfturner@posteo.net>.

References

Baddeley, A. and Chang, Y.-M. (2023) Robust algorithms for simulating cluster point processes. *Journal of Statistical Computation and Simulation* **93**, 1950–1975.

Brix, A. and Kendall, W.S. (2002) Simulation of cluster point processes without edge effects. *Advances in Applied Probability* **34**, 267–280.

Matérn, B. (1960) *Spatial Variation*. Meddelanden från Statens Skogsforskningsinstitut, volume 59, number 5. Statens Skogsforskningsinstitut, Sweden.

Matérn, B. (1986) *Spatial Variation*. Lecture Notes in Statistics 36, Springer-Verlag, New York.

Møller, J., Syversveen, A. and Waagepetersen, R. (1998) Log Gaussian Cox Processes. *Scandinavian Journal of Statistics* **25**, 451–482.

Waagepetersen, R. (2007) An estimating function approach to inference for inhomogeneous Neyman-Scott processes. *Biometrics* **63**, 252–258.

See Also

rpoispp, rThomas, rCauchy, rVarGamma, rNeymanScott, rGaussPoisson.

For fitting the model, see kppm, clusterfit.

Examples

```
# homogeneous
X <- rMatClust(10, 0.05, 4)
# inhomogeneous
ff <- function(x,y){ 4 * exp(2 * abs(x) - 1) }
Z <- as.im(ff, owin())
Y <- rMatClust(10, 0.05, Z)
YY <- rMatClust(ff, 0.05, 3)
```

---

rMaternI *Simulate Matern Model I*

Description

Generate a random point pattern, a simulated realisation of the Matérn Model I inhibition process model.

Usage

```
rMaternI(kappa, r, win = owin(c(0,1),c(0,1)), stationary=TRUE, ...,
         nsim=1, drop=TRUE)
```

Arguments

kappa	Intensity of the Poisson process of proposal points. A single positive number.
r	Inhibition distance.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin. Alternatively a higher-dimensional box of class "box3" or "boxx".
stationary	Logical. Whether to start with a stationary process of proposal points (stationary=TRUE) or to generate the proposal points only inside the window (stationary=FALSE).
...	Ignored.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This algorithm generates one or more realisations of Matérn's Model I inhibition process inside the window `win`.

The process is constructed by first generating a uniform Poisson point process of "proposal" points with intensity `kappa`. If `stationary = TRUE` (the default), the proposal points are generated in a window larger than `win` that effectively means the proposals are stationary. If `stationary=FALSE` then the proposal points are only generated inside the window `win`.

A proposal point is then deleted if it lies within `r` units' distance of another proposal point. Otherwise it is retained.

The retained points constitute Matérn's Model I.

Value

A point pattern if `nsim=1`, or a list of point patterns if `nsim > 1`. Each point pattern is normally an object of class `"ppp"`, but may be of class `"pp3"` or `"ppx"` depending on the window.

Author(s)

Ute Hahn, Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

`rMaternII` for Matérn's Model II.

`rSSI`.

`rpoispp`, `rMatClust`

Examples

```
X <- rMaternI(20, 0.05)
Y <- rMaternI(20, 0.05, stationary=FALSE)
```

`rMaternII` *Simulate Matern Model II*

Description

Generate a random point pattern, a simulated realisation of the Matérn Model II inhibition process.

Usage

```
rMaternII(kappa, r, win = owin(c(0,1),c(0,1)), stationary=TRUE, ...,
          nsim=1, drop=TRUE)
```

Arguments

kappa	Intensity of the Poisson process of proposal points. A single positive number.
r	Inhibition distance.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin. Alternatively a higher-dimensional box of class "box3" or "boxx".
stationary	Logical. Whether to start with a stationary process of proposal points (stationary=TRUE) or to generate the proposal points only inside the window (stationary=FALSE).
...	Ignored.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This algorithm generates one or more realisations of Matérn's Model II inhibition process inside the window win.

The process is constructed by first generating a uniform Poisson point process of "proposal" points with intensity kappa. If stationary = TRUE (the default), the proposal points are generated in a window larger than win that effectively means the proposals are stationary. If stationary=FALSE then the proposal points are only generated inside the window win.

Then each proposal point is marked by an "arrival time", a number uniformly distributed in $[0, 1]$ independently of other variables.

A proposal point is deleted if it lies within r units' distance of another proposal point *that has an earlier arrival time*. Otherwise it is retained. The retained points constitute Matérn's Model II.

The difference between Matérn's Model I and II is the italicised statement above. Model II has a higher intensity for the same parameter values.

Value

A point pattern if nsim=1, or a list of point patterns if nsim > 1. Each point pattern is normally an object of class "ppp", but may be of class "pp3" or "ppx" depending on the window.

Author(s)

Ute Hahn, Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

rMaternI for Matérn's Model I.

rSSI.

rpoispp, rMatClust,

Examples

```
X <- rMaternII(20, 0.05)
Y <- rMaternII(20, 0.05, stationary=FALSE)
```

---

rmh *Simulate point patterns using the Metropolis-Hastings algorithm.*

Description

Generic function for running the Metropolis-Hastings algorithm to produce simulated realisations of a point process model.

Usage

```
rmh(model, ...)
```

Arguments

model	The point process model to be simulated.
...	Further arguments controlling the simulation.

Details

The Metropolis-Hastings algorithm can be used to generate simulated realisations from a wide range of spatial point processes. For caveats, see below.

The function rmh is generic; it has methods rmh.ppm (for objects of class "ppm") and rmh.default (the default). The actual implementation of the Metropolis-Hastings algorithm is contained in rmh.default. For details of its use, see rmh.ppm or rmh.default.

[If the model is a Poisson process, then Metropolis-Hastings is not used; the Poisson model is generated directly using rpoispp or rmpoispp.]

In brief, the Metropolis-Hastings algorithm is a Markov Chain, whose states are spatial point patterns, and whose limiting distribution is the desired point process. After running the algorithm for a very large number of iterations, we may regard the state of the algorithm as a realisation from the desired point process.

However, there are difficulties in deciding whether the algorithm has run for "long enough". The convergence of the algorithm may indeed be extremely slow. No guarantees of convergence are given!

While it is fashionable to decry the Metropolis-Hastings algorithm for its poor convergence and other properties, it has the advantage of being easy to implement for a wide range of models.

Value

A point pattern, in the form of an object of class "ppp". See rmh.default for details.

Warning

As of version 1.22-1 of spatstat a subtle change was made to rmh.default(). We had noticed that the results produced were sometimes not "scalable" in that two models, differing in effect only by the units in which distances are measured and starting from the same seed, gave different results. This was traced to an idiosyncracy of floating point arithmetic. The code of rmh.default() has been changed so that the results produced by rmh are now scalable. The downside of this is that code which users previously ran may now give results which are different from what they formerly were.

In order to recover former behaviour (so that previous results can be reproduced) set spatstat.options(scalable=FALSE). See the last example in the help for rmh.default.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

See Also

rmh.default

Examples

```
# See examples in rmh.default and rmh.ppm
```

---

rmh.default *Simulate Point Process Models using the Metropolis-Hastings Algorithm.*

Description

Generates a random point pattern, simulated from a chosen point process model, using the Metropolis-Hastings algorithm.

Usage

```
   ## Default S3 method:
rmh(model, start=NULL,
   control=default.rmhcontrol(model),
   ...,
   nsim=1, drop=TRUE, saveinfo=TRUE,
   verbose=TRUE, snoop=FALSE)
```

Arguments

model	Data specifying the point process model that is to be simulated.
start	Data determining the initial state of the algorithm.
control	Data controlling the iterative behaviour and termination of the algorithm.
...	Further arguments passed to rmhcontrol or to trend functions in model.
nsim	Number of simulated point patterns that should be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a single point pattern.
saveinfo	Logical value indicating whether to save auxiliary information.
verbose	Logical value indicating whether to print progress reports.
snoop	Logical. If TRUE, activate the visual debugger.

Details

This function generates simulated realisations from any of a range of spatial point processes, using the Metropolis-Hastings algorithm. It is the default method for the generic function rmh.

This function executes a Metropolis-Hastings algorithm with birth, death and shift proposals as described in Geyer and Møller (1994).

The argument model specifies the point process model to be simulated. It is either a list, or an object of class "rmhmodel", with the following components:

cif A character string specifying the choice of interpoint interaction for the point process.

par Parameter values for the conditional intensity function.

w (Optional) window in which the pattern is to be generated. An object of class "owin", or data acceptable to as.owin.

trend Data specifying the spatial trend in the model, if it has a trend. This may be a function, a pixel image (of class "im"), (or a list of functions or images if the model is multitype).

If the trend is a function or functions, any auxiliary arguments ... to rmh.default will be passed to these functions, which should be of the form function(x, y, ...).

types List of possible types, for a multitype point process.

For full details of these parameters, see rmhmodel.default.

The argument start determines the initial state of the Metropolis-Hastings algorithm. It is either NULL, or an object of class "rmhstart", or a list with the following components:

n.start Number of points in the initial point pattern. A single integer, or a vector of integers giving the numbers of points of each type in a multitype point pattern. Incompatible with x.start.

x.start Initial point pattern configuration. Incompatible with n.start.

x.start may be a point pattern (an object of class "ppp"), or data which can be coerced to this class by as.ppp, or an object with components x and y, or a two-column matrix. In the last two cases, the window for the pattern is determined by model$w. In the first two cases, if model$w is also present, then the final simulated pattern will be clipped to the window model$w.

For full details of these parameters, see rmhstart.

The third argument control controls the simulation procedure (including *conditional simulation*), iterative behaviour, and termination of the Metropolis-Hastings algorithm. It is either NULL, or a list, or an object of class "rmhcontrol", with components:

p The probability of proposing a "shift" (as opposed to a birth or death) in the Metropolis-Hastings algorithm.

q The conditional probability of proposing a death (rather than a birth) given that birth/death has been chosen over shift.

nrep The number of repetitions or iterations to be made by the Metropolis-Hastings algorithm. It should be large.

expand Either a numerical expansion factor, or a window (object of class "owin"). Indicates that the process is to be simulated on a larger domain than the original data window w, then clipped to w when the algorithm has finished.

The default is to expand the simulation window if the model is stationary and non-Poisson (i.e. it has no trend and the interaction is not Poisson) and not to expand in all other cases.

If the model has a trend, then in order for expansion to be feasible, the trend must be given either as a function, or an image whose bounding box is large enough to contain the expanded window.

periodic A logical scalar; if periodic is TRUE we simulate a process on the torus formed by identifying opposite edges of a rectangular window.

ptypes A vector of probabilities (summing to 1) to be used in assigning a random type to a new point.

fixall A logical scalar specifying whether to condition on the number of points of each type.

nverb An integer specifying how often "progress reports" (which consist simply of the number of repetitions completed) should be printed out. If nverb is left at 0, the default, the simulation proceeds silently.

x.cond If this argument is present, then *conditional simulation* will be performed, and x.cond specifies the conditioning points and the type of conditioning.

nsave,nburn If these values are specified, then intermediate states of the simulation algorithm will be saved every nsave iterations, after an initial burn-in period of nburn iterations.

track Logical flag indicating whether to save the transition history of the simulations.

For full details of these parameters, see rmhcontrol. The control parameters can also be given in the ... arguments.

Value

A point pattern (an object of class "ppp", see ppp.object) or a list of point patterns.

The returned value has an attribute info containing modified versions of the arguments model, start, and control which together specify the exact simulation procedure. The info attribute can be printed (and is printed automatically by summary.ppp). For computational efficiency, the info attribute can be omitted by setting saveinfo=FALSE.

The value of .Random.seed at the start of the simulations is also saved and returned as an attribute seed.

If the argument track=TRUE was given (see rmhcontrol), the transition history of the algorithm is saved, and returned as an attribute history. The transition history is a data frame containing a factor proposaltype identifying the proposal type (Birth, Death or Shift) and a logical vector accepted indicating whether the proposal was accepted. The data frame also has columns numerator, denominator which give the numerator and denominator of the Hastings ratio for the proposal.

If the argument nsave was given (see rmhcontrol), the return value has an attribute saved which is a list of point patterns, containing the intermediate states of the algorithm.

Conditional Simulation

There are several kinds of conditional simulation.

- Simulation *conditional upon the number of points*, that is, holding the number of points fixed. To do this, set control$p (the probability of a shift) equal to 1. The number of points is then determined by the starting state, which may be specified either by setting start$n.start to be a scalar, or by setting the initial pattern start$x.start.
- In the case of multitype processes, it is possible to simulate the model *conditionally upon the number of points of each type*, i.e. holding the number of points of each type to be fixed. To do this, set control$p equal to 1 and control$fixall to be TRUE. The number of points is then determined by the starting state, which may be specified either by setting start$n.start to be an integer vector, or by setting the initial pattern start$x.start.
- Simulation *conditional on the configuration observed in a sub-window*, that is, requiring that, inside a specified sub-window V, the simulated pattern should agree with a specified point pattern y.To do this, set control$x.cond to equal the specified point pattern y, making sure that it is an object of class "ppp" and that the window Window(control$x.cond) is the conditioning window V.
- Simulation *conditional on the presence of specified points*, that is, requiring that the simulated pattern should include a specified set of points. This is simulation from the Palm distribution of the point process given a pattern y. To do this, set control$x.cond to be a data.frame containing the coordinates (and marks, if appropriate) of the specified points.

For further information, see rmhcontrol.

Note that, when we simulate conditionally on the number of points, or conditionally on the number of points of each type, no expansion of the window is possible.

Visual Debugger

If snoop = TRUE, an interactive debugger is activated. On the current plot device, the debugger displays the current state of the Metropolis-Hastings algorithm together with the proposed transition to the next state. Clicking on this graphical display (using the left mouse button) will re-centre the display at the clicked location. Surrounding this graphical display is an array of boxes representing different actions. Clicking on one of the action boxes (using the left mouse button) will cause the action to be performed. Debugger actions include:

- Zooming in or out
- Panning (shifting the field of view) left, right, up or down
- Jumping to the next iteration

- Skipping 10, 100, 1000, 10000 or 100000 iterations
- Jumping to the next Birth proposal (etc)
- Changing the fate of the proposal (i.e. changing whether the proposal is accepted or rejected)
- Dumping the current state and proposal to a file
- Printing detailed information at the terminal
- Exiting the debugger (so that the simulation algorithm continues without further interruption).

Right-clicking the mouse will also cause the debugger to exit.

Warnings

There is never a guarantee that the Metropolis-Hastings algorithm has converged to its limiting distribution.

If `start$x.start` is specified then `expand` is set equal to 1 and simulation takes place in `Window(x.start)`. Any specified value for `expand` is simply ignored.

The presence of both a component `w` of `model` and a non-null value for `Window(x.start)` makes sense ONLY if `w` is contained in `Window(x.start)`.

For multitype processes make sure that, even if there is to be no trend corresponding to a particular type, there is still a component (a NULL component) for that type, in the list.

Other models

In theory, any finite point process model can be simulated using the Metropolis-Hastings algorithm, provided the conditional intensity is uniformly bounded.

In practice, the list of point process models that can be simulated using `rmh.default` is limited to those that have been implemented in the package's internal C code. More options will be added in the future.

Note that the `lookup` conditional intensity function permits the simulation (in theory, to any desired degree of approximation) of any pairwise interaction process for which the interaction depends only on the distance between the pair of points.

Reproducible simulations

If the user wants the simulation to be exactly reproducible (e.g. for a figure in a journal article, where it is useful to have the figure consistent from draft to draft) then the state of the random number generator should be set before calling `rmh.default`. This can be done either by calling `set.seed` or by assigning a value to `.Random.seed`. In the examples below, we use `set.seed`.

If a simulation has been performed and the user now wants to repeat it exactly, the random seed should be extracted from the simulated point pattern `X` by `seed <- attr(x, "seed")`, then assigned to the system random nunber state by `.Random.seed <- seed` before calling `rmh.default`.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

References

Baddeley, A. and Turner, R. (2000) Practical maximum pseudolikelihood for spatial point patterns. *Australian and New Zealand Journal of Statistics* **42**, 283 – 322.

Diggle, P. J. (2003) *Statistical Analysis of Spatial Point Patterns* (2nd ed.) Arnold, London.

Diggle, P.J. and Gratton, R.J. (1984) Monte Carlo methods of inference for implicit statistical models. *Journal of the Royal Statistical Society, series B* **46**, 193 – 212.

Diggle, P.J., Gates, D.J., and Stibbard, A. (1987) A nonparametric estimator for pairwise-interaction point processes. Biometrika **74**, 763 – 770.

Geyer, C.J. and Møller, J. (1994) Simulation procedures and likelihood inference for spatial point processes. *Scandinavian Journal of Statistics* **21**, 359–373.

Geyer, C.J. (1999) Likelihood Inference for Spatial Point Processes. Chapter 3 in O.E. Barndorff-Nielsen, W.S. Kendall and M.N.M. Van Lieshout (eds) *Stochastic Geometry: Likelihood and Computation*, Chapman and Hall / CRC, Monographs on Statistics and Applied Probability, number 80. Pages 79–140.

See Also

rmh, rmh.ppm, rStrauss, ppp, ppm

Interactions: AreaInter, BadGey, DiggleGatesStibbard, DiggleGratton, Fiksel, Geyer, Hardcore, Hybrid, LennardJones, MultiStrauss, MultiStraussHard, PairPiece, Penttinen, Poisson, Softcore, Strauss, StraussHard and Triplets.

Examples

```
if(online <- interactive()) {
  nr   <- 1e5
  nv  <- 5000
  ns <- 200
} else {
  nr  <- 20
  nv <- 5
  ns <- 20
  oldopt <- spatstat.options()
  spatstat.options(expand=1.05)
}
set.seed(961018)

# Strauss process.
mod01 <- list(cif="strauss",par=list(beta=2,gamma=0.2,r=0.7),
              w=c(0,10,0,10))
X1.strauss <- rmh(model=mod01,start=list(n.start=ns),
                  control=list(nrep=nr,nverb=nv))

if(online) plot(X1.strauss)

# Strauss process, conditioning on n = 42:
X2.strauss <- rmh(model=mod01,start=list(n.start=42),
                  control=list(p=1,nrep=nr,nverb=nv))
```

```
# Tracking algorithm progress:
# (a) saving intermediate states:
X <- rmh(model=mod01,start=list(n.start=ns),
         control=list(nrep=nr, nsave=nr/5, nburn=nr/2))
Saved <- attr(X, "saved")
plot(Saved)

# (b) inspecting transition history:
X <- rmh(model=mod01,start=list(n.start=ns),
         control=list(nrep=nr, track=TRUE))
History <- attr(X, "history")
head(History)

# Hard core process:
mod02 <- list(cif="hardcore",par=list(beta=2,hc=0.7),w=c(0,10,0,10))
X3.hardcore <- rmh(model=mod02,start=list(n.start=ns),
                   control=list(nrep=nr,nverb=nv))

if(online) plot(X3.hardcore)

# Strauss process equal to pure hardcore:
mod02s <- list(cif="strauss",par=list(beta=2,gamma=0,r=0.7),w=c(0,10,0,10))
X3.strauss <- rmh(model=mod02s,start=list(n.start=ns),
                   control=list(nrep=nr,nverb=nv))

# Strauss process in a polygonal window.
x     <- c(0.55,0.68,0.75,0.58,0.39,0.37,0.19,0.26,0.42)
y     <- c(0.20,0.27,0.68,0.99,0.80,0.61,0.45,0.28,0.33)
mod03 <- list(cif="strauss",par=list(beta=2000,gamma=0.6,r=0.07),
              w=owin(poly=list(x=x,y=y)))
X4.strauss <- rmh(model=mod03,start=list(n.start=ns),
                   control=list(nrep=nr,nverb=nv))
if(online) plot(X4.strauss)

# Strauss process in a polygonal window, conditioning on n = 80.
X5.strauss <- rmh(model=mod03,start=list(n.start=ns),
                   control=list(p=1,nrep=nr,nverb=nv))

# Strauss process, starting off from X4.strauss, but with the
# polygonal window replace by a rectangular one.  At the end,
# the generated pattern is clipped to the original polygonal window.
xxx <- X4.strauss
Window(xxx) <- as.owin(c(0,1,0,1))
X6.strauss <- rmh(model=mod03,start=list(x.start=xxx),
                   control=list(nrep=nr,nverb=nv))

# Strauss with hardcore:
mod04 <- list(cif="straush",par=list(beta=2,gamma=0.2,r=0.7,hc=0.3),
              w=c(0,10,0,10))
X1.straush <- rmh(model=mod04,start=list(n.start=ns),
                   control=list(nrep=nr,nverb=nv))

# Another Strauss with hardcore (with a perhaps surprising result):
```

```
mod05 <- list(cif="straush",par=list(beta=80,gamma=0.36,r=45,hc=2.5),
             w=c(0,250,0,250))
X2.straush <- rmh(model=mod05,start=list(n.start=ns),
                  control=list(nrep=nr,nverb=nv))

# Pure hardcore (identical to X3.strauss).
mod06 <- list(cif="straush",par=list(beta=2,gamma=1,r=1,hc=0.7),
             w=c(0,10,0,10))
X3.straush <- rmh(model=mod06,start=list(n.start=ns),
                  control=list(nrep=nr,nverb=nv))

# Soft core:
w    <- c(0,10,0,10)
mod07 <- list(cif="sftcr",par=list(beta=0.8,sigma=0.1,kappa=0.5),
             w=c(0,10,0,10))
X.sftcr <- rmh(model=mod07,start=list(n.start=ns),
               control=list(nrep=nr,nverb=nv))
if(online) plot(X.sftcr)

# Area-interaction process:
mod42 <- rmhmodel(cif="areaint",par=list(beta=2,eta=1.6,r=0.7),
              w=c(0,10,0,10))
X.area <- rmh(model=mod42,start=list(n.start=ns),
              control=list(nrep=nr,nverb=nv))
if(online) plot(X.area)

# Triplets process
modtrip <- list(cif="triplets",par=list(beta=2,gamma=0.2,r=0.7),
                w=c(0,10,0,10))
X.triplets <- rmh(model=modtrip,
                  start=list(n.start=ns),
                  control=list(nrep=nr,nverb=nv))
if(online) plot(X.triplets)

# Multitype Strauss:
beta <- c(0.027,0.008)
gmma <- matrix(c(0.43,0.98,0.98,0.36),2,2)
r    <- matrix(c(45,45,45,45),2,2)
mod08 <- list(cif="straussm",par=list(beta=beta,gamma=gmma,radii=r),
             w=c(0,250,0,250))
X1.straussm <- rmh(model=mod08,start=list(n.start=ns),
                   control=list(ptypes=c(0.75,0.25),nrep=nr,nverb=nv))
if(online) plot(X1.straussm)

# Multitype Strauss conditioning upon the total number
# of points being 80:
X2.straussm <- rmh(model=mod08,start=list(n.start=ns),
                   control=list(p=1,ptypes=c(0.75,0.25),nrep=nr,
                                nverb=nv))

# Conditioning upon the number of points of type 1 being 60
# and the number of points of type 2 being 20:
X3.straussm <- rmh(model=mod08,start=list(n.start=c(60,20)),
```

```
                  control=list(fixall=TRUE,p=1,ptypes=c(0.75,0.25),
                               nrep=nr,nverb=nv))

# Multitype Strauss hardcore:
rhc  <- matrix(c(9.1,5.0,5.0,2.5),2,2)
mod09 <- list(cif="straushm",par=list(beta=beta,gamma=gmma,
             iradii=r,hradii=rhc),w=c(0,250,0,250))
X.straushm <- rmh(model=mod09,start=list(n.start=ns),
                  control=list(ptypes=c(0.75,0.25),nrep=nr,nverb=nv))

# Multitype Strauss hardcore with trends for each type:
beta  <- c(0.27,0.08)
tr3   <- function(x,y){x <- x/250; y <- y/250;
    exp((6*x + 5*y - 18*x^2 + 12*x*y - 9*y^2)/6)
                       }
                       # log quadratic trend
tr4   <- function(x,y){x <- x/250; y <- y/250;
                       exp(-0.6*x+0.5*y)}
                      # log linear trend
mod10 <- list(cif="straushm",par=list(beta=beta,gamma=gmma,
              iradii=r,hradii=rhc),w=c(0,250,0,250),
              trend=list(tr3,tr4))
X1.straushm.trend <- rmh(model=mod10,start=list(n.start=ns),
                         control=list(ptypes=c(0.75,0.25),
                         nrep=nr,nverb=nv))
if(online) plot(X1.straushm.trend)

# Multitype Strauss hardcore with trends for each type, given as images:
bigwin <- square(250)
i1 <- as.im(tr3, bigwin)
i2 <- as.im(tr4, bigwin)
mod11 <- list(cif="straushm",par=list(beta=beta,gamma=gmma,
              iradii=r,hradii=rhc),w=bigwin,
              trend=list(i1,i2))
X2.straushm.trend <- rmh(model=mod11,start=list(n.start=ns),
                         control=list(ptypes=c(0.75,0.25),expand=1,
                         nrep=nr,nverb=nv))

# Diggle, Gates, and Stibbard:
mod12 <- list(cif="dgs",par=list(beta=3600,rho=0.08),w=c(0,1,0,1))
X.dgs <- rmh(model=mod12,start=list(n.start=ns),
             control=list(nrep=nr,nverb=nv))
if(online) plot(X.dgs)

# Diggle-Gratton:
mod13 <- list(cif="diggra",
              par=list(beta=1800,kappa=3,delta=0.02,rho=0.04),
              w=square(1))
X.diggra <- rmh(model=mod13,start=list(n.start=ns),
                control=list(nrep=nr,nverb=nv))
if(online) plot(X.diggra)

# Fiksel:
```

```
modFik <- list(cif="fiksel",
              par=list(beta=180,r=0.15,hc=0.07,kappa=2,a= -1.0),
              w=square(1))
X.fiksel <- rmh(model=modFik,start=list(n.start=ns),
                control=list(nrep=nr,nverb=nv))
if(online) plot(X.fiksel)

# Geyer:
mod14 <- list(cif="geyer",par=list(beta=1.25,gamma=1.6,r=0.2,sat=4.5),
              w=c(0,10,0,10))
X1.geyer <- rmh(model=mod14,start=list(n.start=ns),
                control=list(nrep=nr,nverb=nv))
if(online) plot(X1.geyer)

# Geyer; same as a Strauss process with parameters
# (beta=2.25,gamma=0.16,r=0.7):

mod15 <- list(cif="geyer",par=list(beta=2.25,gamma=0.4,r=0.7,sat=10000),
              w=c(0,10,0,10))
X2.geyer <- rmh(model=mod15,start=list(n.start=ns),
                control=list(nrep=nr,nverb=nv))

mod16 <- list(cif="geyer",par=list(beta=8.1,gamma=2.2,r=0.08,sat=3))

X3.geyer <- rmh(model=mod16,start=list(x.start=redwood),
                control=list(periodic=TRUE,nrep=nr,nverb=nv))

# Geyer, starting from the redwood data set, simulating
# on a torus, and conditioning on n:
X4.geyer <- rmh(model=mod16,start=list(x.start=redwood),
                control=list(p=1,periodic=TRUE,nrep=nr,nverb=nv))

# Lookup (interaction function h_2 from page 76, Diggle (2003)):
   r <- seq(from=0,to=0.2,length=101)[-1] # Drop 0.
   h <- 20*(r-0.05)
   h[r<0.05] <- 0
   h[r>0.10] <- 1
   mod17 <- list(cif="lookup",par=list(beta=4000,h=h,r=r),w=c(0,1,0,1))
   X.lookup <- rmh(model=mod17,start=list(n.start=ns),
                   control=list(nrep=nr,nverb=nv))
   if(online) plot(X.lookup)

# Strauss with trend
tr <- function(x,y){x <- x/250; y <- y/250;
    exp((6*x + 5*y - 18*x^2 + 12*x*y - 9*y^2)/6)
                    }
beta <- 0.3
gmma <- 0.5
r    <- 45
modStr <- list(cif="strauss",par=list(beta=beta,gamma=gmma,r=r),
              w=square(250), trend=tr)
X1.strauss.trend <- rmh(model=modStr,start=list(n.start=ns),
                        control=list(nrep=nr,nverb=nv))
```

```
# Baddeley-Geyer
r <- seq(0,0.2,length=8)[-1]
gmma <- c(0.5,0.6,0.7,0.8,0.7,0.6,0.5)
mod18 <- list(cif="badgey",par=list(beta=4000, gamma=gmma,r=r,sat=5),
             w=square(1))
X1.badgey <- rmh(model=mod18,start=list(n.start=ns),
                 control=list(nrep=nr,nverb=nv))
mod19 <- list(cif="badgey",
             par=list(beta=4000, gamma=gmma,r=r,sat=1e4),
             w=square(1))
set.seed(1329)
X2.badgey <- rmh(model=mod18,start=list(n.start=ns),
                 control=list(nrep=nr,nverb=nv))

# Check:
h <- ((prod(gmma)/cumprod(c(1,gmma)))[-8])^2
hs <- stepfun(r,c(h,1))
mod20 <- list(cif="lookup",par=list(beta=4000,h=hs),w=square(1))
set.seed(1329)
X.check <- rmh(model=mod20,start=list(n.start=ns),
                  control=list(nrep=nr,nverb=nv))
# X2.badgey and X.check will be identical.

mod21 <- list(cif="badgey",par=list(beta=300,gamma=c(1,0.4,1),
             r=c(0.035,0.07,0.14),sat=5), w=square(1))
X3.badgey <- rmh(model=mod21,start=list(n.start=ns),
                 control=list(nrep=nr,nverb=nv))
# Same result as Geyer model with beta=300, gamma=0.4, r=0.07,
# sat = 5 (if seeds and control parameters are the same)

# Or more simply:
mod22 <- list(cif="badgey",
             par=list(beta=300,gamma=0.4,r=0.07, sat=5),
             w=square(1))
X4.badgey <- rmh(model=mod22,start=list(n.start=ns),
                 control=list(nrep=nr,nverb=nv))
# Same again --- i.e. the BadGey model includes the Geyer model.


# Illustrating scalability.
if(FALSE) {
 M1 <- rmhmodel(cif="strauss",par=list(beta=60,gamma=0.5,r=0.04),w=owin())
 set.seed(496)
 X1 <- rmh(model=M1,start=list(n.start=300))
 M2 <- rmhmodel(cif="strauss",par=list(beta=0.6,gamma=0.5,r=0.4),
           w=owin(c(0,10),c(0,10)))
 set.seed(496)
 X2  <- rmh(model=M2,start=list(n.start=300))
 chk <- affine(X1,mat=diag(c(10,10)))
 all.equal(chk,X2,check.attributes=FALSE)
 # Under the default spatstat options the foregoing all.equal()
 # will yield TRUE.  Setting spatstat.options(scalable=FALSE) and
 # re-running the code will reveal differences between X1 and X2.
```

```
}

if(!online) spatstat.options(oldopt)
```

`rmhcontrol` *Set Control Parameters for Metropolis-Hastings Algorithm.*

Description

Sets up a list of parameters controlling the iterative behaviour of the Metropolis-Hastings algorithm.

Usage

```
rmhcontrol(...)

## Default S3 method:
rmhcontrol(..., p=0.9, q=0.5, nrep=5e5,
                    expand=NULL, periodic=NULL, ptypes=NULL,
                    x.cond=NULL, fixall=FALSE, nverb=0,
                    nsave=NULL, nburn=nsave, track=FALSE,
                    pstage=c("block", "start"))
```

Arguments

`...`	Arguments passed to methods.
`p`	Probability of proposing a shift (as against a birth/death).
`q`	Conditional probability of proposing a death given that a birth or death will be proposed.
`nrep`	Total number of steps (proposals) of Metropolis-Hastings algorithm that should be run.
`expand`	Simulation window or expansion rule. Either a window (object of class `"owin"`) or a numerical expansion factor, specifying that simulations are to be performed in a domain other than the original data window, then clipped to the original data window. This argument is passed to `rmhexpand`. A numerical expansion factor can be in several formats: see `rmhexpand`.
`periodic`	Logical value (or `NULL`) indicating whether to simulate "periodically", i.e. identifying opposite edges of the rectangular simulation window. A `NULL` value means "undecided."
`ptypes`	For multitype point processes, the distribution of the mark attached to a new random point (when a birth is proposed)
`x.cond`	Conditioning points for conditional simulation.
`fixall`	(Logical) for multitype point processes, whether to fix the number of points of each type.
`nverb`	Progress reports will be printed every `nverb` iterations

nsave, nburn	If these values are specified, then intermediate states of the simulation algorithm will be saved every nsave iterations, after an initial burn-in period of nburn iterations.
track	Logical flag indicating whether to save the transition history of the simulations.
pstage	Character string specifying when to generate proposal points. Either "start" or "block".

Details

The Metropolis-Hastings algorithm, implemented as rmh, generates simulated realisations of point process models. The function rmhcontrol sets up a list of parameters which control the iterative behaviour and termination of the Metropolis-Hastings algorithm, for use in a subsequent call to rmh. It also checks that the parameters are valid.

(A separate function rmhstart determines the initial state of the algorithm, and rmhmodel determines the model to be simulated.)

The parameters are as follows:

p The probability of proposing a "shift" (as opposed to a birth or death) in the Metropolis-Hastings algorithm.

If $p = 1$ then the algorithm only alters existing points, so the number of points never changes, i.e. we are simulating conditionally upon the number of points. The number of points is determined by the initial state (specified by rmhstart).

If $p = 1$ and fixall=TRUE and the model is a multitype point process model, then the algorithm only shifts the locations of existing points and does not alter their marks (types). This is equivalent to simulating conditionally upon the number of points of each type. These numbers are again specified by the initial state.

If $p = 1$ then no expansion of the simulation window is allowed (see expand below).

The default value of p can be changed by setting the parameter rmh.p in spatstat.options.

q The conditional probability of proposing a death (rather than a birth) given that a shift is not proposed. This is of course ignored if p is equal to 1.

The default value of q can be changed by setting the parameter rmh.q in spatstat.options.

nrep The number of repetitions or iterations to be made by the Metropolis-Hastings algorithm. It should be large.

The default value of nrep can be changed by setting the parameter rmh.nrep in spatstat.options.

expand Either a number or a window (object of class "owin"). Indicates that the process is to be simulated on a domain other than the original data window w, then clipped to w when the algorithm has finished. This would often be done in order to approximate the simulation of a stationary process (Geyer, 1999) or more generally a process existing in the whole plane, rather than just in the window w.

If expand is a window object, it is taken as the larger domain in which simulation is performed.

If expand is numeric, it is interpreted as an expansion factor or expansion distance for determining the simulation domain from the data window. It should be a *named* scalar, such as expand=c(area=2), expand=c(distance=0.1), expand=c(length=1.2). See rmhexpand() for more details. If the name is omitted, it defaults to area.

Expansion is not permitted if the number of points has been fixed by setting p = 1 or if the starting configuration has been specified via the argument x.start in rmhstart.

If expand is NULL, this is interpreted to mean "not yet decided". An expansion rule will be determined at a later stage, using appropriate defaults. See rmhexpand.

periodic A logical value (or NULL) determining whether to simulate "periodically". If periodic is TRUE, and if the simulation window is a rectangle, then the simulation algorithm effectively identifies opposite edges of the rectangle. Points near the right-hand edge of the rectangle are deemed to be close to points near the left-hand edge. Periodic simulation usually gives a better approximation to a stationary point process. For periodic simulation, the simulation window must be a rectangle. (The simulation window is determined by expand as described above.)

The value NULL means 'undecided'. The decision is postponed until rmh is called. Depending on the point process model to be simulated, rmh will then set periodic=TRUE if the simulation window is expanded *and* the expanded simulation window is rectangular; otherwise periodic=FALSE.

Note that periodic=TRUE is only permitted when the simulation window (i.e. the expanded window) is rectangular.

ptypes A vector of probabilities (summing to 1) to be used in assigning a random type to a new point. Defaults to a vector each of whose entries is $1/nt$ where nt is the number of types for the process. Convergence of the simulation algorithm should be improved if ptypes is close to the relative frequencies of the types which will result from the simulation.

x.cond If this argument is given, then *conditional simulation* will be performed, and x.cond specifies the location of the fixed points as well as the type of conditioning. It should be either a point pattern (object of class "ppp") or a list(x,y) or a data.frame. See the section on Conditional Simulation.

fixall A logical scalar specifying whether to condition on the number of points of each type. Meaningful only if a marked process is being simulated, and if $p = 1$. A warning message is given if fixall is set equal to TRUE when it is not meaningful.

nverb An integer specifying how often "progress reports" (which consist simply of the number of repetitions completed) should be printed out. If nverb is left at 0, the default, the simulation proceeds silently.

nsave,nburn If these integers are given, then the current state of the simulation algorithm (i.e. the current random point pattern) will be saved every nsave iterations, starting from iteration nburn. (Alternatively nsave can be a vector, specifying different numbers of iterations between each successive save. This vector will be recycled until the end of the simulations.)

track Logical flag indicating whether to save the transition history of the simulations (i.e. information specifying what type of proposal was made, and whether it was accepted or rejected, for each iteration).

pstage Character string specifying the stage of the algorithm at which the randomised proposal points should be generated. If pstage="start" or if nsave=0, the entire sequence of nrep random proposal points is generated at the start of the algorithm. This is the original behaviour of the code, and should be used in order to maintain consistency with older versions of **spatstat**. If pstage="block" and nsave > 0, then a set of nsave random proposal points will be generated before each block of nsave iterations. This is much more efficient. The default is pstage="block".

Value

An object of class "rmhcontrol", which is essentially a list of parameter values for the algorithm.

There is a print method for this class, which prints a sensible description of the parameters chosen.

Conditional Simulation

For a Gibbs point process X, the Metropolis-Hastings algorithm easily accommodates several kinds of conditional simulation:

conditioning on the total number of points: We fix the total number of points $N(X)$ to be equal to n. We simulate from the conditional distribution of X given $N(X) = n$.

conditioning on the number of points of each type: In a multitype point process, where Y_j denotes the process of points of type j, we fix the number $N(Y_j)$ of points of type j to be equal to n_j, for $j = 1, 2, \ldots, m$. We simulate from the conditional distribution of X given $N(Y_j) = n_j$ for $j = 1, 2, \ldots, m$.

conditioning on the realisation in a subwindow: We require that the point process X should, within a specified sub-window V, coincide with a specified point pattern y. We simulate from the conditional distribution of X given $X \cap V = y$.

Palm conditioning: We require that the point process X include a specified list of points y. We simulate from the point process with probability density $g(x) = cf(x \cup y)$ where f is the probability density of the original process X, and c is a normalising constant.

To achieve each of these types of conditioning we do as follows:

conditioning on the total number of points: Set `p=1`. The number of points is determined by the initial state of the simulation: see `rmhstart`.

conditioning on the number of points of each type: Set `p=1` and `fixall=TRUE`. The number of points of each type is determined by the initial state of the simulation: see `rmhstart`.

conditioning on the realisation in a subwindow: Set `x.cond` to be a point pattern (object of class `"ppp"`). Its window `V=Window(x.cond)` becomes the conditioning subwindow V.

Palm conditioning: Set `x.cond` to be a `list(x,y)` or `data.frame` with two columns containing the coordinates of the points, or a `list(x,y,marks)` or `data.frame` with three columns containing the coordinates and marks of the points.

The arguments `x.cond`, `p` and `fixall` can be combined.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

References

Geyer, C.J. (1999) Likelihood Inference for Spatial Point Processes. Chapter 3 in O.E. Barndorff-Nielsen, W.S. Kendall and M.N.M. Van Lieshout (eds) *Stochastic Geometry: Likelihood and Computation*, Chapman and Hall / CRC, Monographs on Statistics and Applied Probability, number 80. Pages 79–140.

See Also

`rmh`, `rmhmodel`, `rmhstart`, `rmhexpand`, `spatstat.options`

Examples

```
# parameters given as named arguments
c1 <- rmhcontrol(p=0.3,periodic=TRUE,nrep=1e6,nverb=1e5)

# parameters given as a list
liz <- list(p=0.9, nrep=1e4)
c2 <- rmhcontrol(liz)

# parameters given in rmhcontrol object
c3 <- rmhcontrol(c1)
```

---

rmhexpand *Specify Simulation Window or Expansion Rule*

Description

Specify a spatial domain in which point process simulations will be performed. Alternatively, specify a rule which will be used to determine the simulation window.

Usage

```
rmhexpand(x = NULL, ..., area = NULL, length = NULL, distance = NULL)
```

Arguments

x	Any kind of data determining the simulation window or the expansion rule. A window (object of class "owin") specifying the simulation window, a numerical value specifying an expansion factor or expansion distance, a list containing one numerical value, an object of class "rmhexpand", or NULL.
...	Ignored.
area	Area expansion factor. Incompatible with other arguments.
length	Length expansion factor. Incompatible with other arguments.
distance	Expansion distance (buffer width). Incompatible with other arguments.

Details

In the Metropolis-Hastings algorithm rmh for simulating spatial point processes, simulations are usually carried out on a spatial domain that is larger than the original window of the point process model, then subsequently clipped to the original window.

The command rmhexpand can be used to specify the simulation window, or to specify a rule which will later be used to determine the simulation window from data.

The arguments are all incompatible: at most one of them should be given.

If the first argument x is given, it may be any of the following:

- a window (object of class "owin") specifying the simulation window.

- an object of class "rmhexpand" specifying the expansion rule.
- a single numerical value, without attributes. This will be interpreted as the value of the argument area.
- either c(area=v) or list(area=v), where v is a single numeric value. This will be interpreted as the value of the argument area.
- either c(length=v) or list(length=v), where v is a single numeric value. This will be interpreted as the value of the argument length.
- either c(distance=v) or list(distance=v), where v is a single numeric value. This will be interpreted as the value of the argument distance.
- NULL, meaning that the expansion rule is not yet determined.

If one of the arguments area, length or distance is given, then the simulation window is determined from the original data window as follows.

area The bounding box of the original data window will be extracted, and the simulation window will be a scalar dilation of this rectangle. The argument area should be a numerical value, greater than or equal to 1. It specifies the area expansion factor, i.e. the ratio of the area of the simulation window to the area of the original point process window's bounding box.

length The bounding box of the original data window will be extracted, and the simulation window will be a scalar dilation of this rectangle. The argument length should be a numerical value, greater than or equal to 1. It specifies the length expansion factor, i.e. the ratio of the width (height) of the simulation window to the width (height) of the original point process window's bounding box.

distance The argument distance should be a numerical value, greater than or equal to 0. It specifies the width of a buffer region around the original data window. If the original data window is a rectangle, then this window is extended by a margin of width equal to distance around all sides of the original rectangle. The result is a rectangle. If the original data window is not a rectangle, then morphological dilation is applied using dilation.owin so that a margin or buffer of width equal to distance is created around all sides of the original window. The result is a non-rectangular window, typically of a different shape.

Value

An object of class "rmhexpand" specifying the expansion rule. There is a print method for this class.

Undetermined expansion

If expand=NULL, this is interpreted to mean that the expansion rule is "not yet decided". Expansion will be decided later, by the simulation algorithm rmh. If the model cannot be expanded (for example if the covariate data in the model are not available on a larger domain) then expansion will not occur. If the model can be expanded, then if the point process model has a finite interaction range r, the default is rmhexpand(distance=2*r), and otherwise rmhexpand(area=2).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

expand.owin to apply the rule to a window.

will.expand to test whether expansion will occur.

rmh, rmhcontrol for background details.

Examples

```
rmhexpand()
rmhexpand(2)
rmhexpand(1)
rmhexpand(length=1.5)
rmhexpand(distance=0.1)
rmhexpand(letterR)
```

---

rmhmodel *Define Point Process Model for Metropolis-Hastings Simulation.*

Description

Builds a description of a point process model for use in simulating the model by the Metropolis-Hastings algorithm.

Usage

```
rmhmodel(...)
```

Arguments

...	Arguments specifying the point process model in some format.

Details

Simulated realisations of many point process models can be generated using the Metropolis-Hastings algorithm rmh. The algorithm requires the model to be specified in a particular format: an object of class "rmhmodel".

The function rmhmodel takes a description of a point process model in some other format, and converts it into an object of class "rmhmodel". It also checks that the parameters of the model are valid.

The function rmhmodel is generic, with methods for

fitted point process models: an object of class "ppm", obtained by a call to the model-fitting function ppm. See rmhmodel.ppm.

lists: a list of parameter values in a certain format. See rmhmodel.list.

default: parameter values specified as separate arguments to .... See rmhmodel.default.

Value

An object of class "rmhmodel", which is essentially a list of parameter values for the model.

There is a print method for this class, which prints a sensible description of the model chosen.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

References

Diggle, P. J. (2003) *Statistical Analysis of Spatial Point Patterns* (2nd ed.) Arnold, London.

Diggle, P.J. and Gratton, R.J. (1984) Monte Carlo methods of inference for implicit statistical models. *Journal of the Royal Statistical Society, series B* **46**, 193 – 212.

Diggle, P.J., Gates, D.J., and Stibbard, A. (1987) A nonparametric estimator for pairwise-interaction point processes. Biometrika **74**, 763 – 770. *Scandinavian Journal of Statistics* **21**, 359–373.

Geyer, C.J. (1999) Likelihood Inference for Spatial Point Processes. Chapter 3 in O.E. Barndorff-Nielsen, W.S. Kendall and M.N.M. Van Lieshout (eds) *Stochastic Geometry: Likelihood and Computation*, Chapman and Hall / CRC, Monographs on Statistics and Applied Probability, number 80. Pages 79–140.

See Also

rmhmodel.ppm, rmhmodel.default, rmhmodel.list, rmh, rmhcontrol, rmhstart, ppm, Strauss, Softcore, StraussHard, Triplets, MultiStrauss, MultiStraussHard, DiggleGratton, PairPiece Penttinen

---

rmhmodel.default *Build Point Process Model for Metropolis-Hastings Simulation.*

Description

Builds a description of a point process model for use in simulating the model by the Metropolis-Hastings algorithm.

Usage

```
  ## Default S3 method:
rmhmodel(...,
         cif=NULL, par=NULL, w=NULL, trend=NULL, types=NULL)
```

Arguments

...	Ignored.
cif	Character string specifying the choice of model
par	Parameters of the model
w	Spatial window in which to simulate
trend	Specification of the trend in the model
types	A vector of factor levels defining the possible marks, for a multitype process.

Details

The generic function rmhmodel takes a description of a point process model in some format, and converts it into an object of class "rmhmodel" so that simulations of the model can be generated using the Metropolis-Hastings algorithm rmh.

This function rmhmodel.default is the default method. It builds a description of the point process model from the simple arguments listed.

The argument cif is a character string specifying the choice of interpoint interaction for the point process. The current options are

'areaint' Area-interaction process.

'badgey' Baddeley-Geyer (hybrid Geyer) process.

'dgs' Diggle, Gates and Stibbard (1987) process

'diggra' Diggle and Gratton (1984) process

'fiksel' Fiksel double exponential process (Fiksel, 1984).

'geyer' Saturation process (Geyer, 1999).

'hardcore' Hard core process

'lennard' Lennard-Jones process

'lookup' General isotropic pairwise interaction process, with the interaction function specified via a "lookup table".

'multihard' Multitype hardcore process

'penttinen' The Penttinen process

'strauss' The Strauss process

'straush' The Strauss process with hard core

'sftcr' The Softcore process

'straussm' The multitype Strauss process

'straushm' Multitype Strauss process with hard core

'triplets' Triplets process (Geyer, 1999).

It is also possible to specify a *hybrid* of these interactions in the sense of Baddeley et al (2013). In this case, cif is a character vector containing names from the list above. For example, cif=c('strauss', 'geyer') would specify a hybrid of the Strauss and Geyer models.

The argument par supplies parameter values appropriate to the conditional intensity function being invoked. For the interactions listed above, these parameters are:

areaint: (Area-interaction process.) A **named** list with components `beta,eta,r` which are respectively the "base" intensity, the scaled interaction parameter and the interaction radius.

badgey: (Baddeley-Geyer process.) A **named** list with components `beta` (the "base" intensity), `gamma` (a vector of non-negative interaction parameters), `r` (a vector of interaction radii, of the same length as `gamma`, in *increasing* order), and `sat` (the saturation parameter(s); this may be a scalar, or a vector of the same length as `gamma` and `r`; all values should be at least 1). Note that because of the presence of "saturation" the `gamma` values are permitted to be larger than 1.

dgs: (Diggle, Gates, and Stibbard process. See Diggle, Gates, and Stibbard (1987)) A **named** list with components `beta` and `rho`. This process has pairwise interaction function equal to

$$e(t) = \sin^2\left(\frac{\pi t}{2\rho}\right)$$

for $t < \rho$, and equal to 1 for $t \geq \rho$.

diggra: (Diggle-Gratton process. See Diggle and Gratton (1984) and Diggle, Gates and Stibbard (1987).) A **named** list with components `beta`, `kappa`, `delta` and `rho`. This process has pairwise interaction function $e(t)$ equal to 0 for $t < \delta$, equal to

$$\left(\frac{t-\delta}{\rho-\delta}\right)^{\kappa}$$

for $\delta \leq t < \rho$, and equal to 1 for $t \geq \rho$. Note that here we use the symbol κ where Diggle, Gates, and Stibbard use β since we reserve the symbol β for an intensity parameter.

fiksel: (Fiksel double exponential process, see Fiksel (1984)) A **named** list with components `beta`, `r`, `hc`, `kappa` and `a`. This process has pairwise interaction function $e(t)$ equal to 0 for $t < hc$, equal to

$$\exp(a\exp(-\kappa t))$$

for $hc \leq t < r$, and equal to 1 for $t \geq r$.

geyer: (Geyer's saturation process. See Geyer (1999).) A **named** list with components `beta`, `gamma`, `r`, and `sat`. The components `beta`, `gamma`, `r` are as for the Strauss model, and `sat` is the "saturation" parameter. The model is Geyer's "saturation" point process model, a modification of the Strauss process in which we effectively impose an upper limit (`sat`) on the number of neighbours which will be counted as close to a given point.

Explicitly, a saturation point process with interaction radius r, saturation threshold s, and parameters β and γ, is the point process in which each point x_i in the pattern X contributes a factor

$$\beta\gamma^{\min(s,t(x_i,X))}$$

to the probability density of the point pattern, where $t(x_i, X)$ denotes the number of "r-close neighbours" of x_i in the pattern X.

If the saturation threshold s is infinite, the Geyer process reduces to a Strauss process with interaction parameter γ^2 rather than γ.

hardcore: (Hard core process.) A **named** list with components `beta` and `hc` where `beta` is the base intensity and `hc` is the hard core distance. This process has pairwise interaction function $e(t)$ equal to 1 if $t > hc$ and 0 if $t <= hc$.

lennard: (Lennard-Jones process.) A **named** list with components `sigma` and `epsilon`, where `sigma` is the characteristic diameter and `epsilon` is the well depth. See `LennardJones` for explanation.

multihard: (Multitype hard core process.) A **named** list with components `beta` and `hradii`, where `beta` is a vector of base intensities for each type of point, and `hradii` is a matrix of hard core radii between each pair of types.

penttinen: (Penttinen process.) A **named** list with components `beta,gamma,r` which are respectively the "base" intensity, the pairwise interaction parameter, and the disc radius. Note that `gamma` must be less than or equal to 1. See `Penttinen` for explanation. (Note that there is also an algorithm for perfect simulation of the Penttinen process, `rPenttinen`)

strauss: (Strauss process.) A **named** list with components `beta,gamma,r` which are respectively the "base" intensity, the pairwise interaction parameter and the interaction radius. Note that `gamma` must be less than or equal to 1. (Note that there is also an algorithm for perfect simulation of the Strauss process, `rStrauss`)

straush: (Strauss process with hardcore.) A **named** list with entries `beta,gamma,r,hc` where `beta`, `gamma`, and `r` are as for the Strauss process, and `hc` is the hardcore radius. Of course `hc` must be less than `r`.

sftcr: (Softcore process.) A **named** list with components `beta,sigma,kappa`. Again `beta` is a "base" intensity. The pairwise interaction between two points $u \neq v$ is

$$\exp\left\{-\left(\frac{\sigma}{||u-v||}\right)^{2/\kappa}\right\}$$

Note that it is necessary that $0 < \kappa < 1$.

straussm: (Multitype Strauss process.) A **named** list with components

- `beta`: A vector of "base" intensities, one for each possible type.
- `gamma`: A **symmetric** matrix of interaction parameters, with γ_{ij} pertaining to the interaction between type i and type j.
- `radii`: A **symmetric** matrix of interaction radii, with entries r_{ij} pertaining to the interaction between type i and type j.

straushm: (Multitype Strauss process with hardcore.) A **named** list with components `beta` and `gamma` as for `straussm` and **two** "radii" components:

- `iradii`: the interaction radii
- `hradii`: the hardcore radii

which are both symmetric matrices of nonnegative numbers. The entries of `hradii` must be less than the corresponding entries of `iradii`.

triplets: (Triplets process.) A **named** list with components `beta,gamma,r` which are respectively the "base" intensity, the triplet interaction parameter and the interaction radius. Note that `gamma` must be less than or equal to 1.

lookup: (Arbitrary pairwise interaction process with isotropic interaction.) A **named** list with components `beta`, `r`, and `h`, or just with components `beta` and `h`.

This model is the pairwise interaction process with an isotropic interaction given by any chosen function H. Each pair of points x_i, x_j in the point pattern contributes a factor $H(d(x_i, x_j))$ to the probability density, where d denotes distance and H is the pair interaction function.

The component beta is a (positive) scalar which determines the "base" intensity of the process.

In this implementation, H must be a step function. It is specified by the user in one of two ways.

- **as a vector of values:** If r is present, then r is assumed to give the locations of jumps in the function H, while the vector h gives the corresponding values of the function.
  Specifically, the interaction function $H(t)$ takes the value h[1] for distances t in the interval [0, r[1]); takes the value h[i] for distances t in the interval [r[i-1], r[i]) where $i = 2, \ldots, n$; and takes the value 1 for $t \geq r[n]$. Here n denotes the length of r.
  The components r and h must be numeric vectors of equal length. The r values must be strictly positive, and sorted in increasing order.
  The entries of h must be non-negative. If any entry of h is greater than 1, then the entry h[1] must be 0 (otherwise the specified process is non-existent).
  Greatest efficiency is achieved if the values of r are equally spaced.
  [**Note:** The usage of r and h has *changed* from the previous usage in **spatstat** versions 1.4-7 to 1.5-1, in which ascending order was not required, and in which the first entry of r had to be 0.]
- **as a stepfun object:** If r is absent, then h must be an object of class "stepfun" specifying a step function. Such objects are created by stepfun.
  The stepfun object h must be right-continuous (which is the default using stepfun.)
  The values of the step function must all be nonnegative. The values must all be less than 1 unless the function is identically zero on some initial interval $[0, r)$. The rightmost value (the value of h(t) for large t) must be equal to 1.
  Greatest efficiency is achieved if the jumps (the "knots" of the step function) are equally spaced.

For a hybrid model, the argument par should be a list, of the same length as cif, such that par[[i]] is a list of the parameters required for the interaction cif[i]. See the Examples.

The optional argument trend determines the spatial trend in the model, if it has one. It should be a function or image (or a list of such, if the model is multitype) to provide the value of the trend at an arbitrary point.

trend given as a function: A trend function may be a function of any number of arguments, but the first two must be the x, y coordinates of a point. Auxiliary arguments may be passed to the trend function at the time of simulation, via the ... argument to rmh.

The function **must** be **vectorized**. That is, it must be capable of accepting vector valued x and y arguments. Put another way, it must be capable of calculating the trend value at a number of points, simultaneously, and should return the **vector** of corresponding trend values.

trend given as an image: An image (see im.object) provides the trend values at a grid of points in the observation window and determines the trend value at other points as the value at the nearest grid point.

Note that the trend or trends must be **non-negative**; no checking is done for this.

The optional argument w specifies the window in which the pattern is to be generated. If specified, it must be in a form which can be coerced to an object of class owin by as.owin.

The optional argument types specifies the possible types in a multitype point process. If the model being simulated is multitype, and types is not specified, then this vector defaults to 1:ntypes where ntypes is the number of types.

Value

An object of class "rmhmodel", which is essentially a list of parameter values for the model.

There is a print method for this class, which prints a sensible description of the model chosen.

Warnings in Respect of "lookup"

For the lookup cif, the entries of the r component of par must be *strictly positive* and sorted into ascending order.

Note that if you specify the lookup pairwise interaction function via stepfun() the arguments x and y which are passed to stepfun() are slightly different from r and h: length(y) is equal to 1+length(x); the final entry of y must be equal to 1 — i.e. this value is explicitly supplied by the user rather than getting tacked on internally.

The step function returned by stepfun() must be right continuous (this is the default behaviour of stepfun()) otherwise an error is given.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

References

Baddeley, A., Turner, R., Mateu, J. and Bevan, A. (2013) Hybrids of Gibbs point process models and their implementation. *Journal of Statistical Software* **55**:11, 1–43. DOI: 10.18637/jss.v055.i11

Diggle, P. J. (2003) *Statistical Analysis of Spatial Point Patterns* (2nd ed.) Arnold, London.

Diggle, P.J. and Gratton, R.J. (1984) Monte Carlo methods of inference for implicit statistical models. *Journal of the Royal Statistical Society, series B* **46**, 193 – 212.

Diggle, P.J., Gates, D.J., and Stibbard, A. (1987) A nonparametric estimator for pairwise-interaction point processes. Biometrika **74**, 763 – 770. *Scandinavian Journal of Statistics* **21**, 359–373.

Fiksel, T. (1984) Estimation of parameterized pair potentials of marked and non-marked Gibbsian point processes. *Electronische Informationsverabeitung und Kybernetika* **20**, 270–278.

Geyer, C.J. (1999) Likelihood Inference for Spatial Point Processes. Chapter 3 in O.E. Barndorff-Nielsen, W.S. Kendall and M.N.M. Van Lieshout (eds) *Stochastic Geometry: Likelihood and Computation*, Chapman and Hall / CRC, Monographs on Statistics and Applied Probability, number 80. Pages 79–140.

See Also

rmh, rmhcontrol, rmhstart, ppm, AreaInter, BadGey, DiggleGatesStibbard, DiggleGratton, Fiksel, Geyer, Hardcore, Hybrid, LennardJones, MultiStrauss, MultiStraussHard, PairPiece, Penttinen, Poisson, Softcore, Strauss, StraussHard and Triplets.

Examples

```
# Strauss process:
mod01 <- rmhmodel(cif="strauss",par=list(beta=2,gamma=0.2,r=0.7),
             w=c(0,10,0,10))
mod01
```

```
# The above could also be simulated using 'rStrauss'

# Strauss with hardcore:
mod04 <- rmhmodel(cif="straush",par=list(beta=2,gamma=0.2,r=0.7,hc=0.3),
            w=owin(c(0,10),c(0,5)))

# Hard core:
mod05 <- rmhmodel(cif="hardcore",par=list(beta=2,hc=0.3),
           w=square(5))

# Soft core:
w    <- square(10)
mod07 <- rmhmodel(cif="sftcr",
                  par=list(beta=0.8,sigma=0.1,kappa=0.5),
                  w=w)

# Penttinen process:
modpen <- rmhmodel(cif="penttinen",par=list(beta=2,gamma=0.6,r=1),
              w=c(0,10,0,10))

# Area-interaction process:
mod42 <- rmhmodel(cif="areaint",par=list(beta=2,eta=1.6,r=0.7),
              w=c(0,10,0,10))

# Baddeley-Geyer process:
mod99 <- rmhmodel(cif="badgey",par=list(beta=0.3,
                  gamma=c(0.2,1.8,2.4),r=c(0.035,0.07,0.14),sat=5),
                  w=unit.square())

# Multitype Strauss:
beta <- c(0.027,0.008)
gmma <- matrix(c(0.43,0.98,0.98,0.36),2,2)
r    <- matrix(c(45,45,45,45),2,2)
mod08 <- rmhmodel(cif="straussm",
                  par=list(beta=beta,gamma=gmma,radii=r),
                  w=square(250))
# specify types
mod09 <- rmhmodel(cif="straussm",
                  par=list(beta=beta,gamma=gmma,radii=r),
                  w=square(250),
                  types=c("A", "B"))

# Multitype Hardcore:
rhc  <- matrix(c(9.1,5.0,5.0,2.5),2,2)
mod08hard <- rmhmodel(cif="multihard",
                  par=list(beta=beta,hradii=rhc),
                  w=square(250),
                  types=c("A", "B"))


# Multitype Strauss hardcore with trends for each type:
beta  <- c(0.27,0.08)
ri    <- matrix(c(45,45,45,45),2,2)
```

```
rhc  <- matrix(c(9.1,5.0,5.0,2.5),2,2)
tr3   <- function(x,y){x <- x/250; y <- y/250;
    exp((6*x + 5*y - 18*x^2 + 12*x*y - 9*y^2)/6)
                      }
                      # log quadratic trend
tr4   <- function(x,y){x <- x/250; y <- y/250;
                      exp(-0.6*x+0.5*y)}
                     # log linear trend
mod10 <- rmhmodel(cif="straushm",par=list(beta=beta,gamma=gmma,
              iradii=ri,hradii=rhc),w=c(0,250,0,250),
              trend=list(tr3,tr4))

# Triplets process:
mod11 <- rmhmodel(cif="triplets",par=list(beta=2,gamma=0.2,r=0.7),
              w=c(0,10,0,10))

# Lookup (interaction function h_2 from page 76, Diggle (2003)):
   r <- seq(from=0,to=0.2,length=101)[-1] # Drop 0.
   h <- 20*(r-0.05)
   h[r<0.05] <- 0
   h[r>0.10] <- 1
   mod17 <- rmhmodel(cif="lookup",par=list(beta=4000,h=h,r=r),w=c(0,1,0,1))

# hybrid model
modhy <- rmhmodel(cif=c('strauss', 'geyer'),
                  par=list(list(beta=100,gamma=0.5,r=0.05),
                           list(beta=1, gamma=0.7,r=0.1, sat=2)),
                  w=square(1))
modhy
```

`rmhmodel.list` *Define Point Process Model for Metropolis-Hastings Simulation.*

Description

Given a list of parameters, builds a description of a point process model for use in simulating the model by the Metropolis-Hastings algorithm.

Usage

```
   ## S3 method for class 'list'
rmhmodel(model, ...)
```

Arguments

`model`	A list of parameters. See Details.
`...`	Optional list of additional named parameters.

Details

The generic function rmhmodel takes a description of a point process model in some format, and converts it into an object of class "rmhmodel" so that simulations of the model can be generated using the Metropolis-Hastings algorithm rmh.

This function rmhmodel.list is the method for lists. The argument model should be a named list of parameters of the form

list(cif, par, w, trend, types)

where cif and par are required and the others are optional. For details about these components, see rmhmodel.default.

The subsequent arguments ... (if any) may also have these names, and they will take precedence over elements of the list model.

Value

An object of class "rmhmodel", which is essentially a validated list of parameter values for the model.

There is a print method for this class, which prints a sensible description of the model chosen.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

References

Diggle, P. J. (2003) *Statistical Analysis of Spatial Point Patterns* (2nd ed.) Arnold, London.

Diggle, P.J. and Gratton, R.J. (1984) Monte Carlo methods of inference for implicit statistical models. *Journal of the Royal Statistical Society, series B* **46**, 193 – 212.

Diggle, P.J., Gates, D.J., and Stibbard, A. (1987) A nonparametric estimator for pairwise-interaction point processes. Biometrika **74**, 763 – 770. *Scandinavian Journal of Statistics* **21**, 359–373.

Geyer, C.J. (1999) Likelihood Inference for Spatial Point Processes. Chapter 3 in O.E. Barndorff-Nielsen, W.S. Kendall and M.N.M. Van Lieshout (eds) *Stochastic Geometry: Likelihood and Computation*, Chapman and Hall / CRC, Monographs on Statistics and Applied Probability, number 80. Pages 79–140.

See Also

rmhmodel, rmhmodel.default, rmhmodel.ppm, rmh, rmhcontrol, rmhstart, ppm, Strauss, Softcore, StraussHard, MultiStrauss, MultiStraussHard, DiggleGratton, PairPiece

Examples

```
# Strauss process:
mod01 <- list(cif="strauss",par=list(beta=2,gamma=0.2,r=0.7),
              w=c(0,10,0,10))
mod01 <- rmhmodel(mod01)
```

```
# Strauss with hardcore:
mod04 <- list(cif="straush",par=list(beta=2,gamma=0.2,r=0.7,hc=0.3),
             w=owin(c(0,10),c(0,5)))
mod04 <- rmhmodel(mod04)

# Soft core:
w    <- square(10)
mod07 <- list(cif="sftcr",
                  par=list(beta=0.8,sigma=0.1,kappa=0.5),
                  w=w)
mod07 <- rmhmodel(mod07)

# Multitype Strauss:
beta <- c(0.027,0.008)
gmma <- matrix(c(0.43,0.98,0.98,0.36),2,2)
r    <- matrix(c(45,45,45,45),2,2)
mod08 <- list(cif="straussm",
                  par=list(beta=beta,gamma=gmma,radii=r),
                  w=square(250))
mod08 <- rmhmodel(mod08)

# specify types
mod09 <- rmhmodel(list(cif="straussm",
                  par=list(beta=beta,gamma=gmma,radii=r),
                  w=square(250),
                  types=c("A", "B")))

# Multitype Strauss hardcore with trends for each type:
beta  <- c(0.27,0.08)
ri    <- matrix(c(45,45,45,45),2,2)
rhc  <- matrix(c(9.1,5.0,5.0,2.5),2,2)
tr3   <- function(x,y){x <- x/250; y <- y/250;
    exp((6*x + 5*y - 18*x^2 + 12*x*y - 9*y^2)/6)
                      }
                      # log quadratic trend
tr4   <- function(x,y){x <- x/250; y <- y/250;
                      exp(-0.6*x+0.5*y)}
                     # log linear trend
mod10 <- list(cif="straushm",par=list(beta=beta,gamma=gmma,
              iradii=ri,hradii=rhc),w=c(0,250,0,250),
              trend=list(tr3,tr4))
mod10 <- rmhmodel(mod10)

# Lookup (interaction function h_2 from page 76, Diggle (2003)):
r <- seq(from=0,to=0.2,length=101)[-1] # Drop 0.
h <- 20*(r-0.05)
h[r<0.05] <- 0
h[r>0.10] <- 1
mod17 <- list(cif="lookup",par=list(beta=4000,h=h,r=r),w=c(0,1,0,1))
mod17 <- rmhmodel(mod17)
```

rmhstart *Determine Initial State for Metropolis-Hastings Simulation.*

Description

Builds a description of the initial state for the Metropolis-Hastings algorithm.

Usage

```
   rmhstart(start, ...)
   ## Default S3 method:
rmhstart(start=NULL, ..., n.start=NULL, x.start=NULL)
```

Arguments

start	An existing description of the initial state in some format. Incompatible with the arguments listed below.
...	There should be no other arguments.
n.start	Number of initial points (to be randomly generated). Incompatible with x.start.
x.start	Initial point pattern configuration. Incompatible with n.start.

Details

Simulated realisations of many point process models can be generated using the Metropolis-Hastings algorithm implemented in rmh.

This function rmhstart creates a full description of the initial state of the Metropolis-Hastings algorithm, *including possibly the initial state of the random number generator*, for use in a subsequent call to rmh. It also checks that the initial state is valid.

The initial state should be specified **either** by the first argument start **or** by the other arguments n.start, x.start etc.

If start is a list, then it should have components named n.start or x.start, with the same interpretation as described below.

The arguments are:

n.start The number of "initial" points to be randomly (uniformly) generated in the simulation window w. Incompatible with x.start.

For a multitype point process, n.start may be a vector (of length equal to the number of types) giving the number of points of each type to be generated.

If expansion of the simulation window is selected (see the argument expand to rmhcontrol), then the actual number of starting points in the simulation will be n.start multiplied by the expansion factor (ratio of the areas of the expanded window and original window).

For faster convergence of the Metropolis-Hastings algorithm, the value of n.start should be roughly equal to (an educated guess at) the expected number of points for the point process inside the window.

x.start Initial point pattern configuration. Incompatible with `n.start`.

`x.start` may be a point pattern (an object of class `ppp`), or an object which can be coerced to this class by `as.ppp`, or a dataset containing vectors `x` and `y`.

If `x.start` is specified, then expansion of the simulation window (the argument `expand` of `rmhcontrol`) is not permitted.

The parameters `n.start` and `x.start` are *incompatible*.

Value

An object of class `"rmhstart"`, which is essentially a list of parameters describing the initial point pattern and (optionally) the initial state of the random number generator.

There is a `print` method for this class, which prints a sensible description of the initial state.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

`rmh`, `rmhcontrol`, `rmhmodel`

Examples

```
# 30 random points
a <- rmhstart(n.start=30)
a

# a particular point pattern
b <- rmhstart(x.start=cells)
```

---

rMosaicField *Mosaic Random Field*

Description

Generate a realisation of a random field which is piecewise constant on the tiles of a given tessellation.

Usage

```
rMosaicField(X,
    rgen = function(n) { sample(0:1, n, replace = TRUE)},
    ...,
    rgenargs=NULL)
```

Arguments

X	A tessellation (object of class "tess").
...	Arguments passed to as.mask determining the pixel resolution.
rgen	Function that generates random values for the tiles of the tessellation.
rgenargs	List containing extra arguments that should be passed to rgen (typically specifying parameters of the distribution of the values).

Details

This function generates a realisation of a random field which is piecewise constant on the tiles of the given tessellation X. The values in each tile are independent and identically distributed.

Value

A pixel image (object of class "im").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

rpoislinetess, rMosaicSet

Examples

```
if(interactive()) {
  lambda <- 3
  d <- 256
  n <- 30
} else {
  lambda <- 1
  d <- 32
  n <- 5
}
X <- rpoislinetess(lambda)
plot(rMosaicField(X, runif, dimyx=d))
plot(rMosaicField(X, rnorm, rgenargs=list(mean=10, sd=2), dimyx=d))
Y <- dirichlet(runifpoint(n))
plot(rMosaicField(Y, rnorm, dimyx=d))
```

rMosaicSet *Mosaic Random Set*

Description

Generate a random set by taking a random selection of tiles of a given tessellation.

Usage

```
rMosaicSet(X, p=0.5)
```

Arguments

X	A tessellation (object of class "tess").
p	Probability of including a given tile. A number strictly between 0 and 1.

Details

Given a tessellation X, this function randomly selects some of the tiles of X, including each tile with probability p independently of the other tiles. The selected tiles are then combined to form a set in the plane.

One application of this is Switzer's (1965) example of a random set which has a Markov property. It is constructed by generating X according to a Poisson line tessellation (see rpoislinetess).

Value

A window (object of class "owin").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

References

Switzer, P. A random set process in the plane with a Markovian property. *Annals of Mathematical Statistics* **36** (1965) 1859–1863.

See Also

rpoislinetess, rMosaicField

Examples

```
 if(interactive()) {
  lambda <- 3
  n <- 30
 } else {
  lambda <- 1
  n <- 5
 }
# Switzer's random set
 X <- rpoislinetess(lambda)
 plot(rMosaicSet(X, 0.5), col="green", border=NA)

 # another example
 Y <- dirichlet(runifpoint(n))
 plot(rMosaicSet(Y, 0.4))
```

`rmpoint` *Generate N Random Multitype Points*

Description

Generate a random multitype point pattern with a fixed number of points, or a fixed number of points of each type.

Usage

```
rmpoint(n, f=1, fmax=NULL, win=unit.square(),
        types, ptypes,
        ..., giveup=1000,
        fail.action=c("error", "pass", "missing"),
        verbose=FALSE,
        nsim=1, drop=TRUE)
```

Arguments

`n`	Number of marked points to generate. Either a single number specifying the total number of points, or a vector specifying the number of points of each type.
`f`	The probability density of the multitype points, usually un-normalised. Either a constant, a vector, a function `f(x,y,m, ...)`, a pixel image, a list of functions `f(x,y,...)` or a list of pixel images.
`fmax`	An upper bound on the values of `f`. If missing, this number will be estimated.
`win`	Window in which to simulate the pattern. Ignored if `f` is a pixel image or list of pixel images.
`types`	All the possible types for the multitype pattern.
`ptypes`	Optional vector of probabilities for each type.
`...`	Arguments passed to `f` if it is a function.

giveup	Number of attempts in the rejection method after which the algorithm should stop trying to generate new points.
fail.action	Character string (partially matched) specifying what to do if the number of attempts exceeds giveup. If fail.action="error" (the default), a fatal error will be generated. If fail.action="pass", the pattern of accepted points (containing fewer than n points) will be returned. If fail.action="missing", the result will be a 'missing point pattern', an object belonging to the classes "NAobject" and "ppp".
verbose	Flag indicating whether to report details of performance of the simulation algorithm.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates random multitype point patterns consisting of a fixed number of points.

Three different models are available:

I. Random location and type: If n is a single number and the argument ptypes is missing, then n independent, identically distributed random multitype points are generated. Their locations (x[i],y[i]) and types m[i] have joint probability density proportional to $f(x, y, m)$.

II. Random type, and random location given type: If n is a single number and ptypes is given, then n independent, identically distributed random multitype points are generated. Their types m[i] have probability distribution ptypes. Given the types, the locations (x[i],y[i]) have conditional probability density proportional to $f(x, y, m)$.

III. Fixed types, and random location given type: If n is a vector, then we generate n[i] independent, identically distributed random points of type types[i]. For points of type m the conditional probability density of location (x, y) is proportional to $f(x, y, m)$.

Note that the density f is normalised in different ways in Model I and Models II and III. In Model I the normalised joint density is $g(x, y, m) = f(x, y, m)/Z$ where

$$Z = \sum_m \int \int \lambda(x, y, m) \mathrm{d}x \, \mathrm{d}y$$

while in Models II and III the normalised conditional density is $g(x, y \mid m) = f(x, y, m)/Z_m$ where

$$Z_m = \int \int \lambda(x, y, m) \mathrm{d}x \, \mathrm{d}y.$$

In Model I, the marginal distribution of types is $p_m = Z_m/Z$.

The unnormalised density f may be specified in any of the following ways.

single number: If f is a single number, the conditional density of location given type is uniform. That is, the points of each type are uniformly distributed. In Model I, the marginal distribution of types is also uniform (all possible types have equal probability).

vector: If f is a numeric vector, the conditional density of location given type is uniform. That is, the points of each type are uniformly distributed. In Model I, the marginal distribution of types is proportional to the vector f. In Model II, the marginal distribution of types is ptypes, that is, the values in f are ignored. The argument types defaults to names(f), or if that is null, 1:length(f).

function: If f is a function, it will be called in the form f(x,y,m,...) at spatial location (x,y) for points of type m. In Model I, the joint probability density of location and type is proportional to f(x,y,m,...). In Models II and III, the conditional probability density of location (x,y) given type m is proportional to f(x,y,m,...). The function f must work correctly with vectors x, y and m, returning a vector of function values. (Note that m will be a factor with levels types.) The value fmax must be given and must be an upper bound on the values of f(x,y,m,...) for all locations (x, y) inside the window win and all types m. The argument types must be given.

list of functions: If f is a list of functions, then the functions will be called in the form f[[i]](x,y,...) at spatial location (x,y) for points of type types[i]. In Model I, the joint probability density of location and type is proportional to f[[m]](x,y,...). In Models II and III, the conditional probability density of location (x,y) given type m is proportional to f[[m]](x,y,...). The function f[[i]] must work correctly with vectors x and y, returning a vector of function values. The value fmax must be given and must be an upper bound on the values of f[[i]](x,y,...) for all locations (x, y) inside the window win. The argument types defaults to names(f), or if that is null, 1:length(f).

pixel image: If f is a pixel image object of class "im" (see im.object), the unnormalised density at a location (x,y) for points of any type is equal to the pixel value of f for the pixel nearest to (x,y). In Model I, the marginal distribution of types is uniform. The argument win is ignored; the window of the pixel image is used instead. The argument types must be given.

list of pixel images: If f is a list of pixel images, then the image f[[i]] determines the density values of points of type types[i]. The argument win is ignored; the window of the pixel image is used instead. The argument types defaults to names(f), or if that is null, 1:length(f).

The implementation uses the rejection method. For Model I, rmpoispp is called repeatedly until n points have been generated. It gives up after giveup calls if there are still fewer than n points. For Model II, the types are first generated according to ptypes, then the locations of the points of each type are generated using rpoint. For Model III, the locations of the points of each type are generated using rpoint.

Value

A point pattern (an object of class "ppp") if nsim=1, or a list of point patterns if nsim > 1.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

See Also

im.object, owin.object, ppp.object.

Examples

```
abc <- c("a","b","c")

##### Model I

rmpoint(25, types=abc)
rmpoint(25, 1, types=abc)
# 25 points, equal probability for each type, uniformly distributed locations

rmpoint(25, function(x,y,m) {rep(1, length(x))}, types=abc)
# same as above
rmpoint(25, list(function(x,y){rep(1, length(x))},
                 function(x,y){rep(1, length(x))},
                 function(x,y){rep(1, length(x))}),
             types=abc)
# same as above

rmpoint(25, function(x,y,m) { x }, types=abc)
# 25 points, equal probability for each type,
# locations nonuniform with density proportional to x

rmpoint(25, function(x,y,m) { ifelse(m == "a", 1, x) }, types=abc)
rmpoint(25, list(function(x,y) { rep(1, length(x)) },
                 function(x,y) { x },
                 function(x,y) { x }),
                 types=abc)
# 25 points, UNEQUAL probabilities for each type,
# type "a" points uniformly distributed,
# type "b" and "c" points nonuniformly distributed.

##### Model II

rmpoint(25, 1, types=abc, ptypes=rep(1,3)/3)
rmpoint(25, 1, types=abc, ptypes=rep(1,3))
# 25 points, equal probability for each type,
# uniformly distributed locations

rmpoint(25, function(x,y,m) {rep(1, length(x))}, types=abc, ptypes=rep(1,3))
# same as above
rmpoint(25, list(function(x,y){rep(1, length(x))},
                 function(x,y){rep(1, length(x))},
                 function(x,y){rep(1, length(x))}),
             types=abc, ptypes=rep(1,3))
# same as above

rmpoint(25, function(x,y,m) { x }, types=abc, ptypes=rep(1,3))
# 25 points, equal probability for each type,
# locations nonuniform with density proportional to x

rmpoint(25, function(x,y,m) { ifelse(m == "a", 1, x) }, types=abc, ptypes=rep(1,3))
# 25 points, EQUAL probabilities for each type,
# type "a" points uniformly distributed,
```

```
# type "b" and "c" points nonuniformly distributed.

###### Model III

rmpoint(c(12, 8, 4), 1, types=abc)
# 12 points of type "a",
# 8 points of type "b",
# 4 points of type "c",
# each uniformly distributed

rmpoint(c(12, 8, 4), function(x,y,m) { ifelse(m=="a", 1, x)}, types=abc)
rmpoint(c(12, 8, 4), list(function(x,y) { rep(1, length(x)) },
                               function(x,y) { x },
                               function(x,y) { x }),
            types=abc)

# 12 points of type "a", uniformly distributed
# 8 points of type "b", nonuniform
# 4 points of type "c", nonuniform


#########

## Randomising an existing point pattern:
# same numbers of points of each type, uniform random locations (Model III)
rmpoint(table(marks(demopat)), 1, win=Window(demopat))

# same total number of points, distribution of types estimated from X,
# uniform random locations (Model II)
rmpoint(npoints(demopat), 1, types=levels(marks(demopat)), win=Window(demopat),
       ptypes=table(marks(demopat)))
```

`rmpoispp` *Generate Multitype Poisson Point Pattern*

Description

Generate a random point pattern, a realisation of the (homogeneous or inhomogeneous) multitype Poisson process.

Usage

```
rmpoispp(lambda, lmax=NULL, win, types, ...,
         nsim=1, drop=TRUE, warnwin=!missing(win))
```

Arguments

`lambda`	Intensity of the multitype Poisson process. Either a single positive number, a vector, a `function(x,y,m, ...)`, a pixel image, a list of functions `function(x,y, ...)`, or a list of pixel images.

lmax	An upper bound for the value of lambda. May be omitted
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin. Ignored if lambda is a pixel image or list of images.
types	All the possible types for the multitype pattern.
...	Arguments passed to lambda if it is a function.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
warnwin	Logical value specifying whether to issue a warning when win is ignored.

Details

This function generates a realisation of the marked Poisson point process with intensity lambda.

Note that the intensity function $\lambda(x, y, m)$ is the average number of points **of type m** per unit area near the location (x, y). Thus a marked point process with a constant intensity of 10 and three possible types will have an average of 30 points per unit area, with 10 points of each type on average.

The intensity function may be specified in any of the following ways.

single number: If lambda is a single number, then this algorithm generates a realisation of the uniform marked Poisson process inside the window win with intensity lambda for each type. The total intensity of points of all types is lambda * length(types). The argument types must be given and determines the possible types in the multitype pattern.

vector: If lambda is a numeric vector, then this algorithm generates a realisation of the stationary marked Poisson process inside the window win with intensity lambda[i] for points of type types[i]. The total intensity of points of all types is sum(lambda). The argument types defaults to names(lambda), or if that is null, 1:length(lambda).

function: If lambda is a function, the process has intensity lambda(x,y,m,...) at spatial location (x,y) for points of type m. The function lambda must work correctly with vectors x, y and m, returning a vector of function values. (Note that m will be a factor with levels equal to types.) The value lmax, if present, must be an upper bound on the values of lambda(x,y,m,...) for all locations (x, y) inside the window win and all types m. The argument types must be given.

list of functions: If lambda is a list of functions, the process has intensity lambda[[i]](x,y,...) at spatial location (x,y) for points of type types[i]. The function lambda[[i]] must work correctly with vectors x and y, returning a vector of function values. The value lmax, if given, must be an upper bound on the values of lambda(x,y,...) for all locations (x, y) inside the window win. The argument types defaults to names(lambda), or if that is null, 1:length(lambda).

pixel image: If lambda is a pixel image object of class "im" (see im.object), the intensity at a location (x,y) for points of any type is equal to the pixel value of lambda for the pixel nearest to (x,y). The argument win is ignored; the window of the pixel image is used instead. The argument types must be given.

list of pixel images: If `lambda` is a list of pixel images, then the image `lambda[[i]]` determines the intensity of points of type `types[i]`. The argument `win` is ignored; the window of the pixel image is used instead. The argument `types` defaults to `names(lambda)`, or if that is null, `1:length(lambda)`.

If `lmax` is missing, an approximate upper bound will be calculated.

To generate an inhomogeneous Poisson process the algorithm uses "thinning": it first generates a uniform Poisson process of intensity `lmax` for points of each type `m`, then randomly deletes or retains each point independently, with retention probability $p(x, y, m) = \lambda(x, y, m)/\mathrm{lmax}$.

Value

A point pattern (an object of class "ppp") if `nsim=1`, or a list of point patterns if `nsim > 1`. Each point pattern is multitype (it carries a vector of marks which is a factor).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

See Also

`rpoispp` for unmarked Poisson point process; `rmpoint` for a fixed number of random marked points; `im.object`, `owin.object`, `ppp.object`.

Examples

```
# uniform bivariate Poisson process with total intensity 100 in unit square
pp <- rmpoispp(50, types=c("a","b"))

# stationary bivariate Poisson process with intensity A = 30, B = 70
pp <- rmpoispp(c(30,70), types=c("A","B"))
pp <- rmpoispp(c(30,70))

# works in any window
pp <- rmpoispp(c(30,70), win=letterR, types=c("A","B"))

# inhomogeneous lambda(x,y,m)
# note argument 'm' is a factor
lam <- function(x,y,m) { 50 * (x^2 + y^3) * ifelse(m=="A", 2, 1)}
pp <- rmpoispp(lam, win=letterR, types=c("A","B"))
# extra arguments
lam <- function(x,y,m,scal) { scal * (x^2 + y^3) * ifelse(m=="A", 2, 1)}
pp <- rmpoispp(lam, win=letterR, types=c("A","B"), scal=50)

# list of functions lambda[[i]](x,y)
lams <- list(function(x,y){50 * x^2}, function(x,y){20 * abs(y)})
pp <- rmpoispp(lams, win=letterR, types=c("A","B"))
pp <- rmpoispp(lams, win=letterR)
# functions with extra arguments
lams <- list(function(x,y,scal){5 * scal * x^2},
```

```
             function(x,y, scal){2 * scal * abs(y)})
pp <- rmpoispp(lams, win=letterR, types=c("A","B"), scal=10)
pp <- rmpoispp(lams, win=letterR, scal=10)

# florid example
lams <- list(function(x,y){
      100*exp((6*x + 5*y - 18*x^2 + 12*x*y - 9*y^2)/6)
                        }
                        # log quadratic trend
             ,
             function(x,y){
                            100*exp(-0.6*x+0.5*y)
                        }
                       # log linear trend
             )
 X <- rmpoispp(lams, win=unit.square(), types=c("on", "off"))

# pixel image
Z <- as.im(function(x,y){30 * (x^2 + y^3)}, letterR)
pp <- rmpoispp(Z, types=c("A","B"))

# list of pixel images
ZZ <- list(
         as.im(function(x,y){20 * (x^2 + y^3)}, letterR),
         as.im(function(x,y){40 * (x^3 + y^2)}, letterR))
pp <- rmpoispp(ZZ, types=c("A","B"))
pp <- rmpoispp(ZZ)

# randomising an existing point pattern
rmpoispp(intensity(amacrine), win=Window(amacrine))
```

---

rNeymanScott *Simulate Neyman-Scott Process*

Description

Generate a random point pattern, a realisation of the Neyman-Scott cluster process.

Usage

```
rNeymanScott(kappa, expand, rcluster, win = unit.square(),
             ..., nsim=1, drop=TRUE,
             nonempty=TRUE, saveparents=TRUE,
             kappamax=NULL, mumax=NULL)
```

Arguments

kappa	Intensity of the Poisson process of cluster centres. A single positive number, a function, or a pixel image.

expand	Size of the expansion of the simulation window for generating parent points. A single non-negative number.
rcluster	A function which generates random clusters, or other data specifying the random cluster mechanism. See Details.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin.
...	Arguments passed to rcluster.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
nonempty	Logical. If TRUE (the default), a more efficient algorithm is used, in which parents are generated conditionally on having at least one offspring point. If FALSE, parents are generated even if they have no offspring. Both choices are valid; the default is recommended unless you need to simulate all the parent points for some other purpose.
saveparents	Logical value indicating whether to save the locations of the parent points as an attribute.
kappamax	Optional. Upper bound on the values of kappa when kappa is a function or pixel image.
mumax	Optional. Upper bound on the values of mu when mu=rcluster[[1]] is a function or pixel image.

Details

This algorithm generates a realisation of the general Neyman-Scott process, with the cluster mechanism given by the function rcluster.

First, the algorithm generates a Poisson point process of "parent" points with intensity kappa in an expanded window as explained below. Here kappa may be a single positive number, a function kappa(x,y), or a pixel image object of class "im" (see im.object). See rpoispp for details.

Second, each parent point is replaced by a random cluster of points. These clusters are combined together to yield a single point pattern, and the restriction of this pattern to the window win is then returned as the result of rNeymanScott.

The expanded window consists of as.rectangle(win) extended by the amount expand in each direction. The size of the expansion is saved in the attribute "expand" and may be extracted by attr(X, "expand") where X is the generated point pattern.

The argument rcluster specifies the cluster mechanism. It may be either:

- A function which will be called to generate each random cluster (the offspring points of each parent point). The function should expect to be called in the form rcluster(x0,y0,...) for a parent point at a location (x0,y0). The return value of rcluster should specify the coordinates of the points in the cluster; it may be a list containing elements x,y, or a point pattern (object of class "ppp"). If it is a marked point pattern then the result of rNeymanScott will be a marked point pattern.

- A `list(mu, f)` where `mu` specifies the mean number of offspring points in each cluster, and `f` generates the random displacements (vectors pointing from the parent to the offspring). In this case, the number of offspring in a cluster is assumed to have a Poisson distribution, implying that the Neyman-Scott process is also a Cox process. The first element `mu` should be either a single nonnegative number (interpreted as the mean of the Poisson distribution of cluster size) or a pixel image or a `function(x,y)` giving a spatially varying mean cluster size (interpreted in the sense of Waagepetersen, 2007). The second element `f` should be a function that will be called once in the form `f(n)` to generate `n` independent and identically distributed displacement vectors (i.e. as if there were a cluster of size `n` with a parent at the origin `(0,0)`). The function should return a point pattern (object of class `"ppp"`) or something acceptable to `xy.coords` that specifies the coordinates of `n` points.

If required, the intermediate stages of the simulation (the parents and the individual clusters) can also be extracted from the return value of `rNeymanScott` through the attributes `"parents"` and `"parentid"`. The attribute `"parents"` is the point pattern of parent points. The attribute `"parentid"` is an integer vector specifying the parent for each of the points in the simulated pattern.

Neyman-Scott models where `kappa` is a single number and `rcluster = list(mu,f)` can be fitted to data using the function `kppm`.

Value

A point pattern (an object of class `"ppp"`) if `nsim=1`, or a list of point patterns if `nsim > 1`.

Additionally, some intermediate results of the simulation are returned as attributes of this point pattern: see Details.

Inhomogeneous Neyman-Scott Processes

There are several different ways of specifying a spatially inhomogeneous Neyman-Scott process:

- The point process of parent points can be inhomogeneous. If the argument `kappa` is a `function(x,y)` or a pixel image (object of class `"im"`), then it is taken as specifying the intensity function of an inhomogeneous Poisson process according to which the parent points are generated.
- The number of points in a typical cluster can be spatially varying. If the argument `rcluster` is a list of two elements `mu, f` and the first entry `mu` is a `function(x,y)` or a pixel image (object of class `"im"`), then `mu` is interpreted as the reference intensity for offspring points, in the sense of Waagepetersen (2007). For a given parent point, the offspring constitute a Poisson process with intensity function equal to `mu(x, y) * g(x-x0, y-y0)` where `g` is the probability density of the offspring displacements generated by the function `f`.

  Equivalently, clusters are first generated with a constant expected number of points per cluster: the constant is `mumax`, the maximum of `mu`. Then the offspring are randomly *thinned* (see `rthin`) with spatially-varying retention probabilities given by `mu/mumax`.
- The entire mechanism for generating a cluster can be dependent on the location of the parent point. If the argument `rcluster` is a function, then the cluster associated with a parent point at location `(x0,y0)` will be generated by calling `rcluster(x0, y0, ...)`. The behaviour of this function could depend on the location `(x0,y0)` in any fashion.

Note that if `kappa` is an image, the spatial domain covered by this image must be large enough to include the *expanded* window in which the parent points are to be generated. This requirement means that `win` must be small enough so that the expansion of `as.rectangle(win)` is contained

in the spatial domain of kappa. As a result, one may wind up having to simulate the process in a window smaller than what is really desired.

In the first two cases, the intensity of the Neyman-Scott process is equal to kappa * mu if at least one of kappa or mu is a single number, and is otherwise equal to an integral involving kappa, mu and f.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

References

Neyman, J. and Scott, E.L. (1958) A statistical approach to problems of cosmology. *Journal of the Royal Statistical Society, Series B* **20**, 1–43.

Waagepetersen, R. (2007) An estimating function approach to inference for inhomogeneous Neyman-Scott processes. *Biometrics* **63**, 252–258.

See Also

rpoispp, rThomas, rGaussPoisson, rMatClust, rCauchy, rVarGamma

Examples

```
# each cluster consist of 10 points in a disc of radius 0.2
nclust <- function(x0, y0, radius, n) {
            return(runifdisc(n, radius, centre=c(x0, y0)))
          }
plot(rNeymanScott(10, 0.2, nclust, radius=0.2, n=5))

# multitype Neyman-Scott process (each cluster is a multitype process)
nclust2 <- function(x0, y0, radius, n, types=c("a", "b")) {
   X <- runifdisc(n, radius, centre=c(x0, y0))
   M <- sample(types, n, replace=TRUE)
   marks(X) <- M
   return(X)
}
plot(rNeymanScott(15,0.1,nclust2, radius=0.1, n=5))
```

---

rnoise *Random Pixel Noise*

Description

Generate a pixel image whose pixel values are random numbers following a specified probability distribution.

Usage

```
rnoise(rgen = runif, w = square(1), ...)
```

Arguments

rgen	Random generator for the pixel values. A function in the R language.
w	Window (region or pixel raster) in which to generate the image. Any data acceptable to as.mask.
...	Arguments, matched by name, to be passed to rgen to specify the parameters of the probability distribution, or passed to as.mask to control the pixel resolution.

Details

The argument w could be a window (class "owin"), a pixel image (class "im") or other data. It is first converted to a binary mask by as.mask using any relevant arguments in ....

Then each pixel inside the window (i.e. with logical value TRUE in the mask) is assigned a random numerical value by calling the function rgen.

The function rgen would typically be one of the standard random variable generators like runif (uniformly distributed random values) or rnorm (Gaussian random values). Its first argument n is the number of values to be generated. Other arguments to rgen must be matched by name.

Value

A pixel image (object of class "im").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

Rolf Turner <rolfturner@posteo.net>

and Ege Rubak <rubak@math.aau.dk>

See Also

as.mask, as.im, Distributions.

Examples

```
plot(rnoise(), main="Uniform noise")
plot(rnoise(rnorm, dimyx=32, mean=2, sd=1),
     main="White noise")
```

`rPenttinen` *Perfect Simulation of the Penttinen Process*

Description

Generate a random pattern of points, a simulated realisation of the Penttinen process, using a perfect simulation algorithm.

Usage

```
rPenttinen(beta, gamma=1, R, W = owin(), expand=TRUE, nsim=1, drop=TRUE)
```

Arguments

`beta`	intensity parameter (a positive number).
`gamma`	Interaction strength parameter (a number between 0 and 1).
`R`	disc radius (a non-negative number).
`W`	window (object of class `"owin"`) in which to generate the random pattern.
`expand`	Logical. If `FALSE`, simulation is performed in the window `W`, which must be rectangular. If `TRUE` (the default), simulation is performed on a larger window, and the result is clipped to the original window `W`. Alternatively `expand` can be an object of class `"rmhexpand"` (see `rmhexpand`) determining the expansion method.
`nsim`	Number of simulated realisations to be generated.
`drop`	Logical. If `nsim=1` and `drop=TRUE` (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a realisation of the Penttinen point process in the window `W` using a 'perfect simulation' algorithm.

Penttinen (1984, Example 2.1, page 18), citing Cormack (1979), described the pairwise interaction point process with interaction factor

$$h(d) = e^{\theta A(d)} = \gamma^{A(d)}$$

between each pair of points separated by a distance d. Here $A(d)$ is the area of intersection between two discs of radius R separated by a distance d, normalised so that $A(0) = 1$.

The simulation algorithm used to generate the point pattern is 'dominated coupling from the past' as implemented by Berthelsen and Møller (2002, 2003). This is a 'perfect simulation' or 'exact simulation' algorithm, so called because the output of the algorithm is guaranteed to have the correct probability distribution exactly (unlike the Metropolis-Hastings algorithm used in `rmh`, whose output is only approximately correct).

There is a tiny chance that the algorithm will run out of space before it has terminated. If this occurs, an error message will be generated.

Value

If nsim = 1, a point pattern (object of class "ppp"). If nsim > 1, a list of point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, based on original code for the Strauss process by Kasper Klitgaard Berthelsen.

References

Berthelsen, K.K. and Møller, J. (2002) A primer on perfect simulation for spatial point processes. *Bulletin of the Brazilian Mathematical Society* 33, 351-367.

Berthelsen, K.K. and Møller, J. (2003) Likelihood and non-parametric Bayesian MCMC inference for spatial point processes based on perfect simulation and path sampling. *Scandinavian Journal of Statistics* 30, 549-564.

Cormack, R.M. (1979) Spatial aspects of competition between individuals. Pages 151–212 in *Spatial and Temporal Analysis in Ecology*, eds. R.M. Cormack and J.K. Ord, International Co-operative Publishing House, Fairland, MD, USA.

Møller, J. and Waagepetersen, R. (2003). *Statistical Inference and Simulation for Spatial Point Processes.* Chapman and Hall/CRC.

Penttinen, A. (1984) *Modelling Interaction in Spatial Point Patterns: Parameter Estimation by the Maximum Likelihood Method.* Jyväskylä Studies in Computer Science, Economics and Statistics **7**, University of Jyväskylä, Finland.

See Also

rmh,

rStrauss, rHardcore, rStraussHard, rDiggleGratton, rDGS.

Penttinen.

Examples

```
X <- rPenttinen(50, 0.5, 0.02)
Z <- rPenttinen(50, 0.5, 0.01, nsim=2)
```

---

rpoint *Generate N Random Points*

Description

Generate a random point pattern containing n independent, identically distributed random points with any specified distribution.

Usage

```
rpoint(n, f, fmax=NULL, win=unit.square(),
       ..., giveup=1000, fail.action=c("error", "pass", "missing"),
       warn=TRUE, verbose=FALSE,
       nsim=1, drop=TRUE, forcewin=FALSE)
```

Arguments

n	Number of points to generate.
f	The probability density of the points, possibly un-normalised. Either a constant, a function f(x,y,...), or a pixel image object.
fmax	An upper bound on the values of f. If missing, this number will be estimated.
win	Window in which to simulate the pattern. (Ignored if f is a pixel image, unless forcewin=TRUE).
...	Arguments passed to the function f.
giveup	Number of attempts in the rejection method after which the algorithm should stop trying to generate new points.
fail.action	Character string (partially matched) specifying what to do if the number of attempts exceeds giveup. If fail.action="error" (the default), a fatal error will be generated. If fail.action="pass", the pattern of accepted points (containing fewer than n points) will be returned. If fail.action="missing", the result will be a 'missing point pattern', an object belonging to the classes "NAobject" and "ppp".
warn	Logical value specifying whether to issue a warning if n is very large. See Details.
verbose	Flag indicating whether to report details of performance of the simulation algorithm.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
forcewin	Logical. If TRUE, then simulations will be generated inside win in all cases. If FALSE (the default), the argument win is ignored when f is a pixel image.

Details

This function generates n independent, identically distributed random points with common probability density proportional to f.

The argument f may be

a numerical constant: uniformly distributed random points will be generated.

a function: random points will be generated in the window win with probability density proportional to f(x,y,...) where x and y are the cartesian coordinates. The function f must accept two *vectors* of coordinates x,y and return the corresponding vector of function values. Additional arguments ... of any kind may be passed to the function.

a pixel image: if f is a pixel image (object of class "im", see im.object) then random points will be generated with probability density proportional to the pixel values of f. To be precise, pixels are selected with probabilities proportional to the pixel values, and within each selected pixel, a point is generated with a uniform distribution inside the pixel.

The window of the simulated point pattern is determined as follows. If forcewin=FALSE (the default) then the argument win is ignored, and the simulation window is the window of the pixel image, Window(f). If forcefit=TRUE then the simulation window is win.

The algorithm is as follows:

- If f is a constant, we invoke runifpoint.
- If f is a function, then we use the rejection method. Proposal points are generated from the uniform distribution. A proposal point (x, y) is accepted with probability f(x,y,...)/fmax and otherwise rejected. The algorithm continues until n points have been accepted. It gives up after giveup * n proposals if there are still fewer than n points.
- If f is a pixel image, then a random sequence of pixels is selected (using sample) with probabilities proportional to the pixel values of f. Then for each pixel in the sequence we generate a uniformly distributed random point in that pixel.

The algorithm for pixel images is more efficient than that for functions.

If warn=TRUE (the default), a warning will be issued if n is very large. The threshold is spatstat.options("huge.npoints")
This warning has no consequences, but it helps to trap a number of common errors.

Value

A point pattern (an object of class "ppp") if nsim=1, or a list of point patterns if nsim > 1.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

ppp.object, owin.object, im.object, runifpoint

Examples

```
# 100 uniform random points in the unit square
X <- rpoint(100)

# 100 random points with probability density proportional to x^2 + y^2
X <- rpoint(100, function(x,y) { x^2 + y^2}, 1)

# `fmax' may be omitted
X <- rpoint(100, function(x,y) { x^2 + y^2})

# irregular window
X <- rpoint(100, function(x,y) { x^2 + y^2}, win=letterR)
```

```
# make a pixel image
Z <- setcov(letterR)
# 100 points with density proportional to pixel values
X <- rpoint(100, Z)
```

---

rpoint3 *Generate N Random Points in 3 Dimensions*

Description

Generate a random point pattern in three-dimensional space containing n independent, identically distributed random points with a specified distribution.

Usage

```
rpoint3(n, f, fmax = NULL, domain = box3(), ...,
  nsim = 1, drop = TRUE, ex = NULL, giveup = 1000)
```

Arguments

n	Number of points to generate.
f	The probability density of the points, possibly un-normalised. Either a constant, or a function f(x,y,z, ...).
fmax	An upper bound on the values of f.
domain	Three-dimensional box (object of class "box3") in which to generate the point pattern.
...	Additional arguments passed to the function f.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
ex	Optional. Example point pattern in three dimensions (object of class "pp3"). The number of points n and the spatial region domain will be taken from ex.
giveup	Integer. Maximum number of attempts permitted in the rejection method.

Details

This function generates n independent, identically distributed random points with common probability density proportional to f inside the specified three-dimensional region domain.

The argument f may be

a numerical constant: uniformly distributed random points will be generated.

a function: random points will be generated in the domain with probability density proportional to f(x, y, z, ...) where x, y, z are the cartesian coordinates. The function f must accept *vectors* of coordinates x,y,z and return the corresponding vector of function values. Additional arguments ... of any kind may be passed to the function.

The algorithm is as follows:

- If f is a constant, we invoke runifpoint3.
- If f is a function, then we use the rejection method. Proposal points are generated from the uniform distribution. A proposal point (x, y, z) is accepted with probability f(x,y,z,...)/fmax and otherwise rejected. The algorithm continues until n points have been accepted. It gives up after giveup * n proposals if there are still fewer than n points.

Value

A three-dimensional point pattern (an object of class "pp3") if nsim=1 and drop=TRUE, otherwise a list of three-dimensional point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

runifpoint3

Examples

```
rpoint3(10, 1)
rpoint3(10, f=function(x,y,z) {sqrt(x)}, fmax=1)
```

---

rpoisDirichletTess *Poisson Dirichlet Tessellation*

Description

Generate a random tessellation of a given spatial region, using the Dirichlet tessellation defined by a Poisson point process on the infinite plane.

Usage

```
rpoisDirichletTess(lambda, win = owin())
```

Arguments

lambda	Intensity of the Poisson point process. Either a single positive number, or a function which can be evaluated at any spatial location.
win	Window to be divided into tiles. An object of class "owin" or something acceptable to as.owin.

Details

This algorithm computes the Dirichlet-Voronoi tessellation of the window `win` induced by a Poisson point process on the infinite two-dimensional plane, avoiding edge effects.

The argument `lambda` is the intensity of the Poisson process. It should be either a single positive number, or a `function(x,y)` which can be evaluated at any spatial location (x, y).

The algorithm is a modification of Algorithm C_2 of Hinde and Miles (1980). It first generates a realisation of the Poisson process inside `win`, and computes the Dirichlet-Voronoi tessellation of this point pattern. Then in order to avoid edge effects, the algorithm calculates the maximum possible distance d outside the window such that a random point outside `w` could affect the outcome inside `w`. The window is expanded by this distance d; the realisation of the Poisson process in the expanded region is generated; and the Dirichlet-Voronoi tessellation of the augmented point pattern is computed, then restricted to the original window.

If `lambda` is a single number, the result is a realisation of a stationary random tessellation (restricted to `win`).

Value

A tessellation (object of class `"tess"`).

Also has an attribute `"X"` containing the realisation of the Poisson point process (restricted to those points which affected the result).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

References

Hinde, A.L. and Miles, R.E. (1980) Monte Carlo estimates of the distribution of the random polygons of the Voronoi tessellation with respect to a Poisson process. *J. Statist. Comput. Simul.* **10**, 205–223.

See Also

`rpoislinetess`

Examples

```
A <- rpoisDirichletTess(10)
X <- attr(A, "X")
plot(Frame(X), type="n", main="rpoisDirichletTess(10)")
plot(A, add=TRUE)
plot(X, add=TRUE, pch=16)
```

rpoisline *Generate Poisson Random Line Process*

Description

Generate a random pattern of line segments obtained from the Poisson line process.

Usage

```
rpoisline(lambda, win=owin())
```

Arguments

lambda	Intensity of the Poisson line process. A positive number.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin.

Details

This algorithm generates a realisation of the uniform Poisson line process, and clips it to the window win.

The argument lambda must be a positive number. It controls the intensity of the process. The expected number of lines intersecting a convex region of the plane is equal to lambda times the perimeter length of the region. The expected total length of the lines crossing a region of the plane is equal to lambda * pi times the area of the region.

Value

A line segment pattern (an object of class "psp").

The result also has an attribute called "lines" (an object of class "infline" specifying the original infinite random lines) and an attribute "linemap" (an integer vector mapping the line segments to their parent lines).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

psp

Examples

```
# uniform Poisson line process with intensity 10,
# clipped to the unit square
rpoisline(10)
```

rpoislinetess *Poisson Line Tessellation*

Description

Generate a random tessellation delineated by the lines of the Poisson line process.

Usage

```
rpoislinetess(lambda, win = owin())
```

Arguments

lambda	Intensity of the Poisson line process. A positive number.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin. Currently, the window must be a rectangle.

Details

This algorithm generates a realisation of the uniform Poisson line process, and divides the window win into tiles separated by these lines.

The argument lambda must be a positive number. It controls the intensity of the process. The expected number of lines intersecting a convex region of the plane is equal to lambda times the perimeter length of the region. The expected total length of the lines crossing a region of the plane is equal to lambda * pi times the area of the region.

Value

A tessellation (object of class "tess").

Also has an attribute "lines" containing the realisation of the Poisson line process, as an object of class "infline".

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>.

See Also

rpoisline to generate the lines only.

rpoisDirichletTess

Examples

```
X <- rpoislinetess(3)
plot(as.im(X), main="rpoislinetess(3)")
plot(X, add=TRUE)
```

rpoispp *Generate Poisson Point Pattern*

Description

Generate a random point pattern using the (homogeneous or inhomogeneous) Poisson process. Includes CSR (complete spatial randomness).

Usage

```
rpoispp(lambda, lmax=NULL, win=owin(), ...,
        nsim=1, drop=TRUE, ex=NULL,
        forcewin=FALSE, warnwin=TRUE)
```

Arguments

lambda	Intensity of the Poisson process. Either a single positive number, a function(x,y, ...), or a pixel image.
lmax	Optional. An upper bound for the value of lambda(x,y), if lambda is a function.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin. (Ignored if lambda is a pixel image unless forcewin=TRUE.)
...	Arguments passed to lambda if it is a function.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
ex	Optional. A point pattern to use as the example. If ex is given and lambda, lmax, win are missing, then lambda and win will be calculated from the point pattern ex.
forcewin	Logical value specifying whether to use the argument win as the simulation window when lambda is an image.
warnwin	Logical value specifying whether to issue a warning when win is ignored (which occurs when lambda is an image, win is present and forcewin=FALSE).

Details

If lambda is a single number, then this algorithm generates a realisation of the uniform Poisson process (also known as Complete Spatial Randomness, CSR) inside the window win with intensity lambda (points per unit area).

If lambda is a function, then this algorithm generates a realisation of the inhomogeneous Poisson process with intensity function lambda(x,y,...) at spatial location (x,y) inside the window win. The function lambda must work correctly with vectors x and y.

If lmax is given, it must be an upper bound on the values of lambda(x,y,...) for all locations (x, y) inside the window win. That is, we must have lambda(x,y,...) <= lmax for all locations (x,y). If this is not true then the results of the algorithm will be incorrect.

If lmax is missing or NULL, an approximate upper bound is computed by finding the maximum value of lambda(x,y,...) on a grid of locations (x,y) inside the window win, and adding a safety margin equal to 5 percent of the range of lambda values. This can be computationally intensive, so it is advisable to specify lmax if possible.

If lambda is a pixel image object of class "im" (see im.object), this algorithm generates a realisation of the inhomogeneous Poisson process with intensity equal to the pixel values of the image. (The value of the intensity function at an arbitrary location is the pixel value of the nearest pixel.) If forcewin=FALSE (the default), the simulation window will be the window of the pixel image (converted to a rectangle if possible using rescue.rectangle). If forcewin=TRUE, the simulation window will be the argument win.

For *marked* point patterns, use rmpoispp.

Value

A point pattern (an object of class "ppp") if nsim=1, or a list of point patterns if nsim > 1.

Warning

Note that lambda is the **intensity**, that is, the expected number of points **per unit area**. The total number of points in the simulated pattern will be random with expected value mu = lambda * a where a is the area of the window win.

Reproducibility

The simulation algorithm has changed in the following cases.

- lambda is a function which is constant on each tile of a tessellation (a function of class "tessfun" created by as.tess) and the tiles are rectangles or polygons. The algorithm was changed in spatstat.random version 3.3-3. The new algorithm generates uniform Poisson realisations in each tile of the tessellation. This algorithm is exact (i.e. does not involve spatial discretisation), and is usually faster than the old algorithm. Set the argument tilewise=FALSE to use the previous rejection algorithm.
- lambda is a pixel image. The algorithm was changed in **spatstat** version 1.42-3. The new faster algorithm randomly selects pixels with probability proportional to intensity, and generates point locations inside the selected pixels. Set spatstat.options(fastpois=FALSE) to use the previous, slower algorithm, if it is desired to reproduce results obtained with earlier versions.

In both cases the previous algorithm uses "thinning": it first generates a uniform Poisson process of intensity lmax, then randomly deletes or retains each point, independently of other points, with retention probability $p(x, y) = \lambda(x, y)/\text{lmax}$.

Thinning is still used when lambda is a function(x,y,...) in all other cases.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

rmpoispp for Poisson *marked* point patterns, runifpoint for a fixed number of independent uniform random points; rpoint, rmpoint for a fixed number of independent random points with any distribution; rMaternI, rMaternII, rSSI, rStrauss, rstrat for random point processes with spatial inhibition or regularity; rThomas, rGaussPoisson, rMatClust, rcell for random point processes exhibiting clustering; rmh.default for Gibbs processes. See also ppp.object, owin.object.

Examples

```
# uniform Poisson process with intensity 100 in the unit square
pp <- rpoispp(100)

# uniform Poisson process with intensity 1 in a 10 x 10 square
pp <- rpoispp(1, win=owin(c(0,10),c(0,10)))
# plots should look similar !

# inhomogeneous Poisson process in unit square
# with intensity lambda(x,y) = 100 * exp(-3*x)
# Intensity is bounded by 100
pp <- rpoispp(function(x,y) {100 * exp(-3*x)}, 100)

# How to tune the coefficient of x
lamb <- function(x,y,a) { 100 * exp( - a * x)}
pp <- rpoispp(lamb, 100, a=3)

# pixel image
Z <- as.im(function(x,y){100 * sqrt(x+y)}, unit.square())
pp <- rpoispp(Z)

# randomising an existing point pattern
rpoispp(intensity(cells), win=Window(cells))
rpoispp(ex=cells)
```

---

rpoispp3 *Generate Poisson Point Pattern in Three Dimensions*

Description

Generate a random three-dimensional point pattern using the homogeneous Poisson process.

Usage

```
rpoispp3(lambda, domain = box3(), ..., nsim=1, drop=TRUE, ex=NULL, lmax=NULL)
```

Arguments

lambda	Intensity of the Poisson process. A single positive number, or a function(x,y,z).
domain	Three-dimensional box in which the process should be generated. An object of class "box3".

...	Ignored.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
ex	An example point pattern (object of class "pp3") which will be used to determine the intensity lambda and the containing box domain.
lmax	Maximum possible value of the intensity function lambda, when lambda is a function.

Details

This function generates a realisation of the Poisson process in three dimensions, with intensity lambda (points per unit volume).

If lambda is a single number, the homogeneous Poisson process with constant intensity lambda is generated.

If lambda is a function with arguments x,y,z, the inhomogeneous Poisson process with intensity function lambda(x,y,z) is generated. In this case the argument lmax is required.

The realisation is generated inside the three-dimensional region domain which currently must be a rectangular box (object of class "box3").

Value

If nsim = 1 and drop=TRUE, a point pattern in three dimensions (an object of class "pp3"). If nsim > 1, a list of such point patterns.

Note

The intensity lambda is the expected number of points *per unit volume*.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>.

See Also

runifpoint3, pp3, box3

Examples

```
X <- rpoispp3(50)

rpoispp3(ex=osteo$pts[[1]])
```

`rpoisppOnLines` *Generate Poisson Point Pattern on Line Segments*

Description

Given a line segment pattern, generate a Poisson random point pattern on the line segments.

Usage

```
rpoisppOnLines(lambda, L, lmax = NULL, ..., nsim=1, drop=TRUE)
```

Arguments

`lambda`	Intensity of the Poisson process. A single number, a `function(x,y)`, a pixel image (object of class `"im"`), or a vector of numbers, a list of functions, or a list of images.
`L`	Line segment pattern (object of class `"psp"`) on which the points should be generated.
`lmax`	Optional upper bound (for increased computational efficiency). A known upper bound for the values of `lambda`, if `lambda` is a function or a pixel image. That is, `lmax` should be a number which is known to be greater than or equal to all values of `lambda`.
`...`	Additional arguments passed to `lambda` if it is a function.
`nsim`	Number of simulated realisations to be generated.
`drop`	Logical. If `nsim=1` and `drop=TRUE` (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This command generates a Poisson point process on the one-dimensional system of line segments in `L`. The result is a point pattern consisting of points lying on the line segments in `L`. The number of random points falling on any given line segment follows a Poisson distribution. The patterns of points on different segments are independent.

The intensity `lambda` is the expected number of points per unit **length** of line segment. It may be constant, or it may depend on spatial location.

In order to generate an unmarked Poisson process, the argument `lambda` may be a single number, or a `function(x,y)`, or a pixel image (object of class `"im"`).

In order to generate a *marked* Poisson process, `lambda` may be a numeric vector, a list of functions, or a list of images, each entry giving the intensity for a different mark value.

If `lambda` is not numeric, then the (Lewis-Shedler) rejection method is used. The rejection method requires knowledge of `lmax`, the maximum possible value of `lambda`. This should be either a single number, or a numeric vector of the same length as `lambda`. If `lmax` is not given, it will be computed approximately, by sampling many values of `lambda`.

If `lmax` is given, then it **must** be larger than any possible value of `lambda`, otherwise the results of the algorithm will be incorrect.

Value

If nsim = 1, a point pattern (object of class "ppp") in the same window as L. If nsim > 1, a list of such point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

See Also

psp, ppp, runifpointOnLines, rpoispp

Examples

```
live <- interactive()
L <- psp(runif(10), runif(10), runif(10), runif(10),  window=owin())
if(live) plot(L, main="")

# uniform intensity
Y <- rpoisppOnLines(4, L)
if(live) plot(Y, add=TRUE, pch="+")

# uniform MARKED process with types 'a' and 'b'
Y <- rpoisppOnLines(c(a=4, b=5), L)
if(live) {
  plot(L, main="")
  plot(Y, add=TRUE, pch="+")
}

# intensity is a function
Y <- rpoisppOnLines(function(x,y){ 10 * x^2}, L, 10)
if(live) {
  plot(L, main="")
  plot(Y, add=TRUE, pch="+")
}

# intensity is an image
Z <- as.im(function(x,y){10 * sqrt(x+y)}, unit.square())
Y <- rpoisppOnLines(Z, L, 15)
if(live) {
 plot(L, main="")
 plot(Y, add=TRUE, pch="+")
}
```

rpoisppx *Generate Poisson Point Pattern in Any Dimensions*

Description

Generate a random multi-dimensional point pattern using the homogeneous Poisson process.

Usage

```
rpoisppx(lambda, domain, nsim=1, drop=TRUE)
```

Arguments

lambda	Intensity of the Poisson process. A single positive number.
domain	Multi-dimensional box in which the process should be generated. An object of class "boxx".
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a single point pattern.

Details

This function generates a realisation of the homogeneous Poisson process in multi dimensions, with intensity lambda (points per unit volume).

The realisation is generated inside the multi-dimensional region domain which currently must be a rectangular box (object of class "boxx").

Value

If nsim = 1 and drop=TRUE, a point pattern (an object of class "ppx"). If nsim > 1 or drop=FALSE, a list of such point patterns.

Note

The intensity lambda is the expected number of points *per unit volume*.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>.

See Also

runifpointx, ppx, boxx

Examples

```
w <- boxx(x=c(0,1), y=c(0,1), z=c(0,1), t=c(0,3))
X <- rpoisppx(10, w)
```

rPoissonCluster *Simulate Poisson Cluster Process*

Description

Generate a random point pattern, a realisation of the general Poisson cluster process.

Usage

```
rPoissonCluster(kappa, expand, rcluster, win = owin(c(0,1),c(0,1)),
                ..., nsim=1, drop=TRUE, saveparents=TRUE, kappamax=NULL)
```

Arguments

kappa	Intensity of the Poisson process of cluster centres. A single positive number, a function, or a pixel image.
expand	Size of the expansion of the simulation window for generating parent points. A single non-negative number.
rcluster	A function which generates random clusters.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin.
...	Arguments passed to rcluster
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
saveparents	Logical value indicating whether to save the locations of the parent points as an attribute.
kappamax	Optional. Upper bound on the values of kappa when kappa is a function or pixel image.

Details

This algorithm generates a realisation of the general Poisson cluster process, with the cluster mechanism given by the function rcluster.

First, the algorithm generates a Poisson point process of "parent" points with intensity kappa in an expanded window as explained below.. Here kappa may be a single positive number, a function kappa(x, y), or a pixel image object of class "im" (see im.object). See rpoispp for details.

Second, each parent point is replaced by a random cluster of points, created by calling the function rcluster. These clusters are combined together to yield a single point pattern, and the restriction of this pattern to the window win is then returned as the result of rPoissonCluster.

The expanded window consists of as.rectangle(win) extended by the amount expand in each direction. The size of the expansion is saved in the attribute "expand" and may be extracted by attr(X, "expand") where X is the generated point pattern.

The function rcluster should expect to be called as rcluster(xp[i],yp[i],...) for each parent point at a location (xp[i],yp[i]). The return value of rcluster should be a list with elements x,y which are vectors of equal length giving the absolute x and y coordinates of the points in the cluster.

If the return value of rcluster is a point pattern (object of class "ppp") then it may have marks. The result of rPoissonCluster will then be a marked point pattern.

If required, the intermediate stages of the simulation (the parents and the individual clusters) can also be extracted from the return value of rPoissonCluster through the attributes "parents" and "parentid". The attribute "parents" is the point pattern of parent points. The attribute "parentid" is an integer vector specifying the parent for each of the points in the simulated pattern. (If these data are not required, it is more efficient to set saveparents=FALSE.)

Value

A point pattern (an object of class "ppp") if nsim=1, or a list of point patterns if nsim > 1.

Additionally, some intermediate results of the simulation are returned as attributes of the point pattern: see Details.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

See Also

rpoispp, rMatClust, rThomas, rCauchy, rVarGamma, rNeymanScott, rGaussPoisson.

Examples

```
# each cluster consist of 10 points in a disc of radius 0.2
nclust <- function(x0, y0, radius, n) {
           return(runifdisc(n, radius, centre=c(x0, y0)))
         }
plot(rPoissonCluster(10, 0.2, nclust, radius=0.2, n=5))

# multitype Neyman-Scott process (each cluster is a multitype process)
nclust2 <- function(x0, y0, radius, n, types=c("a", "b")) {
   X <- runifdisc(n, radius, centre=c(x0, y0))
   M <- sample(types, n, replace=TRUE)
   marks(X) <- M
   return(X)
}
plot(rPoissonCluster(15,0.1,nclust2, radius=0.1, n=5))
```

rpoistrunc *Truncated Poisson Distribution*

Description

Generate random realisations, or calculate probability distribution or quantiles, of a Poisson random variable which is conditioned to be nonzero, or conditioned to be at least a given number.

Usage

```
dpoisnonzero(x, lambda, log=FALSE)
ppoisnonzero(q, lambda, lower.tail=TRUE, log.p=FALSE)
qpoisnonzero(p, lambda, lower.tail=TRUE, log.p=FALSE)
rpoisnonzero(n, lambda, method=c("harding", "transform"), implem=c("R", "C"))

dpoistrunc(x, lambda, minimum=1, log=FALSE)
ppoistrunc(q, lambda, minimum=1, lower.tail=TRUE, log.p=FALSE)
qpoistrunc(p, lambda, minimum=1, lower.tail=TRUE, log.p=FALSE)
rpoistrunc(n, lambda, minimum=1, method=c("harding", "transform"), implem=c("R", "C"))
```

Arguments

x	Vector of quantiles.
q	Vector of quantiles.
p	Vector of probabilities.
n	Number of random values to be generated.
lambda	Mean value of the un-truncated Poisson distribution. A nonnegative number, or vector of nonnegative numbers.
minimum	Minimum permitted value for the random variables. A nonnegative integer, or vector of nonnegative integers.
lower.tail	Logical value. If TRUE (the default), probabilities are $P[X <= x]$; otherwise probabilities are $P[X > x]$.
log, log.p	Logical value. If TRUE, probabilities are given as the natural logarithm $\log(p)$.
method	Character string (partially matched) specifying the simulation algorithm to be used. See Details.
implem	Character string specifying whether to use the implementation in interpreted R code (implem="R", the default) or C code (implem="C").

Details

rpoisnonzero generates realisations of the Poisson distribution with mean lambda conditioned on the event that the values are not equal to zero. The functions dpoisnonzero, ppoisnonzero, qpoisnonzero calculate the probability mass function, cumulative distribution function, and quantile function, respectively.

rpoistrunc generates realisations of the Poisson distribution with mean lambda conditioned on the event that the values are greater than or equal to minimum. The default minimum=1 is equivalent to generating zero-truncated Poisson random variables using rpoisnonzero. The value minimum=0 is equivalent to generating un-truncated Poisson random variables using rpois. The functions dpoistrunc, ppoistrunc, qpoistrunc calculate the probability mass function, cumulative distribution function, and quantile function, respectively.

The arguments lambda and minimum can be vectors of length n, specifying different means for the un-truncated Poisson distribution, and different minimum values, for each of the n random output values.

For the random generators, if method="transform" the simulated values are generated by transforming a uniform random variable using the quantile function of the Poisson distribution. If method="harding" (the default) the simulated values are generated using an algorithm proposed by E.F. Harding which exploits properties of the Poisson point process. The Harding algorithm seems to be faster.

Value

Numerical vector.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, after ideas of Ted Harding and Peter Dalgaard.

References

E.F. Harding (2005) Email to R-help email group, 01 May 2005.

P. Dalgaard (2005) Email to R-help email group, 01 May 2005.

See Also

rpois for Poisson random variables.

recipEnzpois for the reciprocal moment of rpoisnonzero.

Examples

```
rpoisnonzero(10, 0.8)

rpoistrunc(10, 1, 2)
```

rPSNCP *Simulate Product Shot-noise Cox Process*

Description

Generate a random multitype point pattern, a realisation of the product shot-noise Cox process.

Usage

```
rPSNCP(lambda=rep(100, 4), kappa=rep(25, 4), omega=rep(0.03, 4),
       alpha=matrix(runif(16, -1, 3), nrow=4, ncol=4),
       kernels=NULL, nu.ker=NULL, win=owin(), nsim=1, drop=TRUE,
       ...,
       cnames=NULL, epsth=0.001)
```

Arguments

lambda	List of intensities of component processes. Either a numeric vector determining the constant (homogeneous) intensities or a list of pixel images (objects of class "im") determining the (inhomogeneous) intensity functions of component processes. The length of lambda determines the number of component processes.
kappa	Numeric vector of intensities of the Poisson process of cluster centres for component processes. Must have the same size as lambda.
omega	Numeric vector of bandwidths of cluster dispersal kernels for component processes. Must have the same size as lambda and kappa.
alpha	Matrix of interaction parameters. Square numeric matrix with the same number of rows and columns as the length of lambda, kappa and omega. All entries of alpha must be greater than -1.
kernels	Vector of character string determining the cluster dispersal kernels of component processes. Implemented kernels are Gaussian kernel ("Thomas") with bandwidth omega, Variance-Gamma (Bessel) kernel ("VarGamma") with bandwidth omega and shape parameter nu.ker and Cauchy kernel ("Cauchy") with bandwidth omega. Must have the same length as lambda, kappa and omega.
nu.ker	Numeric vector of bandwidths of shape parameters for Variance-Gamma kernels.
win	Window in which to simulate the pattern. An object of class "owin".
nsim	Number of simulated realisations to be generated.
cnames	Optional vector of character strings giving the names of the component processes.
...	Optional arguments passed to as.mask to determine the pixel array geometry. See as.mask.
epsth	Numerical threshold to determine the maximum interaction range for cluster kernels.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a realisation of a product shot-noise Cox process (PSNCP). This is a multitype (multivariate) Cox point process in which each element of the multivariate random intensity $\Lambda(u)$ of the process is obtained by

$$\Lambda_i(u) = \lambda_i(u) S_i(u) \prod_{j \neq i} E_{ji}(u)$$

where $\lambda_i(u)$ is the intensity of component i of the process,

$$S_i(u) = \frac{1}{\kappa_i} \sum_{v \in \Phi_i} k_i(u - v)$$

is the shot-noise random field for component i and

$$E_{ji}(u) = \exp(-\kappa_j \alpha_{ji} / k_j(0)) \prod_{v \in \Phi_j} 1 + \alpha_{ji} \frac{k_j(u - v)}{k_j(0)}$$

is a product field controlling impulses from the parent Poisson process Φ_j with constant intensity κ_j of component process j on $\Lambda_i(u)$. Here $k_i(u)$ is an isotropic kernel (probability density) function on R^2 with bandwidth ω_i and shape parameter ν_i, and $\alpha_{ji} > -1$ is the interaction parameter.

Value

A point pattern (an object of class `"ppp"`) if `nsim=1`, or a list of point patterns if `nsim > 1`. Each point pattern is multitype (it carries a vector of marks which is a factor).

Author(s)

Abdollah Jalilian. Modified by Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

References

Jalilian, A., Guan, Y., Mateu, J. and Waagepetersen, R. (2015) Multivariate product-shot-noise Cox point process models. *Biometrics* **71**(4), 1022–1033.

See Also

rmpoispp, rThomas, rVarGamma, rCauchy, rNeymanScott

Examples

```
online <- interactive()
# Example 1: homogeneous components
lambda <- c(250, 300, 180, 400)
kappa <- c(30, 25, 20, 25)
omega <- c(0.02, 0.025, 0.03, 0.02)
alpha <- matrix(runif(16, -1, 1), nrow=4, ncol=4)
if(!online) {
```

```
   lambda <- lambda[1:2]/10
   kappa  <- kappa[1:2]
   omega  <- omega[1:2]
   alpha  <- alpha[1:2, 1:2]
}
X <- rPSNCP(lambda, kappa, omega, alpha)
if(online) {
  plot(X)
  plot(split(X))
}

#Example 2: inhomogeneous components
z1 <- scaletointerval.im(bei.extra$elev, from=0, to=1)
z2 <- scaletointerval.im(bei.extra$grad, from=0, to=1)
if(!online) {
  ## reduce resolution to reduce check time
  z1 <- as.im(z1, dimyx=c(40,80))
  z2 <- as.im(z2, dimyx=c(40,80))
}
lambda <- list(
       exp(-8 + 1.5 * z1 + 0.5 * z2),
       exp(-7.25 + 1 * z1  - 1.5 * z2),
       exp(-6 - 1.5 * z1 + 0.5 * z2),
       exp(-7.5 + 2 * z1 - 3 * z2))
kappa <- c(35, 30, 20, 25) / (1000 * 500)
omega <- c(15, 35, 40, 25)
alpha <- matrix(runif(16, -1, 1), nrow=4, ncol=4)
if(!online) {
   lambda <- lapply(lambda[1:2], "/", e2=10)
   kappa  <- kappa[1:2]
   omega  <- omega[1:2]
   alpha  <- alpha[1:2, 1:2]
} else {
   sapply(lambda, integral)
}
X <- rPSNCP(lambda, kappa, omega, alpha, win = Window(bei), dimyx=dim(z1))
if(online) {
  plot(X)
  plot(split(X), cex=0.5)
}
```

`rshift` *Random Shift*

Description

Randomly shifts the points of a point pattern or line segment pattern. Generic.

Usage

```
rshift(X, ...)
```

Arguments

X	Pattern to be subjected to a random shift. A point pattern (class "ppp"), a line segment pattern (class "psp") or an object of class "splitppp".
...	Arguments controlling the generation of the random shift vector, or specifying which parts of the pattern will be shifted.

Details

This operation applies a random shift (vector displacement) to the points in a point pattern, or to the segments in a line segment pattern.

The argument X may be

- a point pattern (an object of class "ppp")
- a line segment pattern (an object of class "psp")
- an object of class "splitppp" (basically a list of point patterns, obtained from split.ppp).

The function rshift is generic, with methods for the three classes "ppp", "psp" and "splitppp".

See the help pages for these methods, rshift.ppp, rshift.psp and rshift.splitppp, for further information.

Value

An object of the same type as X.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>.

See Also

rshift.ppp, rshift.psp, rshift.splitppp

---

rshift.ppp *Randomly Shift a Point Pattern*

Description

Randomly shifts the points of a point pattern.

Usage

```
   ## S3 method for class 'ppp'
rshift(X, ..., which=NULL, group, nsim=1, drop=TRUE)
```

Arguments

X	Point pattern to be subjected to a random shift. An object of class "ppp"
...	Arguments that determine the random shift. See Details.
group	Optional. Factor specifying a grouping of the points of X, or NULL indicating that all points belong to the same group. Each group will be shifted together, and separately from other groups. By default, points in a marked point pattern are grouped according to their mark values, while points in an unmarked point pattern are treated as a single group.
which	Optional. Identifies which groups of the pattern will be shifted, while other groups are not shifted. A vector of levels of group.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This operation randomly shifts the locations of the points in a point pattern.

The function rshift is generic. This function rshift.ppp is the method for point patterns.

The most common use of this function is to shift the points in a multitype point pattern. By default, points of the same type are shifted in parallel (i.e. points of a common type are shifted by a common displacement vector), and independently of other types. This is useful for testing the hypothesis of independence of types (the null hypothesis that the sub-patterns of points of each type are independent point processes).

In general the points of X are divided into groups, then the points within a group are shifted by a common random displacement vector. Different groups of points are shifted independently. The grouping is determined as follows:

- If the argument group is present, then this determines the grouping.
- Otherwise, if X is a multitype point pattern, the marks determine the grouping.
- Otherwise, all points belong to a single group.

The argument group should be a factor, of length equal to the number of points in X. Alternatively group may be NULL, which specifies that all points of X belong to a single group.

By default, every group of points will be shifted. The argument which indicates that only some of the groups should be shifted, while other groups should be left unchanged. which must be a vector of levels of group (for example, a vector of types in a multitype pattern) indicating which groups are to be shifted.

The displacement vector, i.e. the vector by which the data points are shifted, is generated at random. Parameters that control the randomisation and the handling of edge effects are passed through the ... argument. They are

radius,width,height Parameters of the random shift vector.

edge String indicating how to deal with edges of the pattern. Options are "torus", "erode" and "none".

clip Optional. Window to which the final point pattern should be clipped.

If the window is a rectangle, the *default* behaviour is to generate a displacement vector at random with equal probability for all possible displacements. This means that the x and y coordinates of the displacement vector are independent random variables, uniformly distributed over the range of possible coordinates.

Alternatively, the displacement vector can be generated by another random mechanism, controlled by the arguments `radius`, `width` and `height`.

rectangular: if `width` and `height` are given, then the displacement vector is uniformly distributed in a rectangle of these dimensions, centred at the origin. The maximum possible displacement in the x direction is `width/2`. The maximum possible displacement in the y direction is `height/2`. The x and y displacements are independent. (If `width` and `height` are actually equal to the dimensions of the observation window, then this is equivalent to the default.)

radial: if `radius` is given, then the displacement vector is generated by choosing a random point inside a disc of the given radius, centred at the origin, with uniform probability density over the disc. Thus the argument `radius` determines the maximum possible displacement distance. The argument `radius` is incompatible with the arguments `width` and `height`.

The argument `edge` controls what happens when a shifted point lies outside the window of `X`. Options are:

"none": Points shifted outside the window of `X` simply disappear.

"torus": Toroidal or periodic boundary. Treat opposite edges of the window as identical, so that a point which disappears off the right-hand edge will re-appear at the left-hand edge. This is called a "toroidal shift" because it makes the rectangle topologically equivalent to the surface of a torus (doughnut).

The window must be a rectangle. Toroidal shifts are undefined if the window is non-rectangular.

"erode": Clip the point pattern to a smaller window.

If the random displacements are generated by a radial mechanism (see above), then the window of `X` is eroded by a distance equal to the value of the argument `radius`, using `erosion`.

If the random displacements are generated by a rectangular mechanism, then the window of `X` is (if it is not rectangular) eroded by a distance `max(height,width)` using `erosion`; or (if it is rectangular) trimmed by a margin of width `width` at the left and right sides and trimmed by a margin of height `height` at the top and bottom.

The rationale for this is that the clipping window is the largest window for which edge effects can be ignored.

The optional argument `clip` specifies a smaller window to which the pattern should be restricted.

If `nsim > 1`, then the simulation procedure is performed `nsim` times; the result is a list of `nsim` point patterns.

Value

A point pattern (object of class `"ppp"`) or a list of point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

rshift, rshift.psp

Examples

```
# random toroidal shift
# shift "on" and "off" points separately
X <- rshift(amacrine)

# shift "on" points and leave "off" points fixed
X <- rshift(amacrine, which="on")

# shift all points simultaneously
X <- rshift(amacrine, group=NULL)

# maximum displacement distance 0.1 units
X <- rshift(amacrine, radius=0.1, nsim=2)

# shift with erosion
X <- rshift(amacrine, radius=0.1, edge="erode")
```

---

rshift.psp *Randomly Shift a Line Segment Pattern*

Description

Randomly shifts the segments in a line segment pattern.

Usage

```
   ## S3 method for class 'psp'
rshift(X, ..., group=NULL, which=NULL)
```

Arguments

X	Line segment pattern to be subjected to a random shift. An object of class "psp".
...	Arguments controlling the randomisation and the handling of edge effects. See rshift.ppp.
group	Optional. Factor specifying a grouping of the line segments of X, or NULL indicating that all line segments belong to the same group. Each group will be shifted together, and separately from other groups.
which	Optional. Identifies which groups of the pattern will be shifted, while other groups are not shifted. A vector of levels of group.

Details

This operation randomly shifts the locations of the line segments in a line segment pattern.

The function `rshift` is generic. This function `rshift.psp` is the method for line segment patterns.

The line segments of `X` are first divided into groups, then the line segments within a group are shifted by a common random displacement vector. Different groups of line segments are shifted independently. If the argument `group` is present, then this determines the grouping. Otherwise, all line segments belong to a single group.

The argument `group` should be a factor, of length equal to the number of line segments in `X`. Alternatively `group` may be `NULL`, which specifies that all line segments of `X` belong to a single group.

By default, every group of line segments will be shifted. The argument `which` indicates that only some of the groups should be shifted, while other groups should be left unchanged. `which` must be a vector of levels of `group` indicating which groups are to be shifted.

The displacement vector, i.e. the vector by which the data line segments are shifted, is generated at random. The *default* behaviour is to generate a displacement vector at random with equal probability for all possible displacements. This means that the x and y coordinates of the displacement vector are independent random variables, uniformly distributed over the range of possible coordinates.

Alternatively, the displacement vector can be generated by another random mechanism, controlled by the arguments `radius`, `width` and `height`.

rectangular: if `width` and `height` are given, then the displacement vector is uniformly distributed in a rectangle of these dimensions, centred at the origin. The maximum possible displacement in the x direction is `width/2`. The maximum possible displacement in the y direction is `height/2`. The x and y displacements are independent. (If `width` and `height` are actually equal to the dimensions of the observation window, then this is equivalent to the default.)

radial: if `radius` is given, then the displacement vector is generated by choosing a random line segment inside a disc of the given radius, centred at the origin, with uniform probability density over the disc. Thus the argument `radius` determines the maximum possible displacement distance. The argument `radius` is incompatible with the arguments `width` and `height`.

The argument `edge` controls what happens when a shifted line segment lies partially or completely outside the window of `X`. Currently the only option is `"erode"` which specifies that the segments will be clipped to a smaller window.

The optional argument `clip` specifies a smaller window to which the pattern should be restricted.

Value

A line segment pattern (object of class `"psp"`).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

See Also

`rshift`, `rshift.ppp`

Examples

```
X <- psp(runif(20), runif(20), runif(20), runif(20),  window=owin())
Y <- rshift(X, radius=0.1)
```

`rshift.splitppp` *Randomly Shift a List of Point Patterns*

Description

Randomly shifts each point pattern in a list of point patterns.

Usage

```
   ## S3 method for class 'splitppp'
rshift(X, ..., which=seq_along(X), nsim=1, drop=TRUE)
```

Arguments

`X`	An object of class `"splitppp"`. Basically a list of point patterns.
`...`	Parameters controlling the generation of the random shift vector and the handling of edge effects. See `rshift.ppp`.
`which`	Optional. Identifies which patterns will be shifted, while other patterns are not shifted. Any valid subset index for `X`.
`nsim`	Number of simulated realisations to be generated.
`drop`	Logical. If `nsim=1` and `drop=TRUE` (the default), the result will be a split point pattern object, rather than a list containing the split point pattern.

Details

This operation applies a random shift to each of the point patterns in the list `X`.

The function `rshift` is generic. This function `rshift.splitppp` is the method for objects of class `"splitppp"`, which are essentially lists of point patterns, created by the function `split.ppp`.

By default, every pattern in the list `X` will be shifted. The argument `which` indicates that only some of the patterns should be shifted, while other groups should be left unchanged. `which` can be any valid subset index for `X`.

Each point pattern in the list `X` (or each pattern in `X[which]`) is shifted by a random displacement vector. The shifting is performed by `rshift.ppp`.

See the help page for `rshift.ppp` for details of the other arguments.

If `nsim > 1`, then the simulation procedure is performed `nsim` times; the result is a list of split point patterns.

Value

Another object of class `"splitppp"`, or a list of such objects.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

rshift, rshift.ppp

Examples

```
Y <- split(amacrine)

# random toroidal shift
# shift "on" and "off" points separately
X <- rshift(Y)

# shift "on" points and leave "off" points fixed
X <- rshift(Y, which="on")

# maximum displacement distance 0.1 units
X <- rshift(Y, radius=0.1)

# shift with erosion
X <- rshift(Y, radius=0.1, edge="erode")
```

---

rSSI *Simulate Simple Sequential Inhibition*

Description

Generate a random point pattern, a realisation of the Simple Sequential Inhibition (SSI) process, in two- or three-dimensional space.

Usage

```
rSSI(r, n=Inf, win = square(1), giveup = 1000, x.init=NULL, ...,
     f=NULL, fmax=NULL, nsim=1, drop=TRUE, verbose=TRUE)
```

Arguments

r	Inhibition distance.
n	Maximum number of points allowed. If n is finite, stop when the *total* number of points in the point pattern reaches n. If n is infinite (the default), stop only when it is apparently impossible to add any more points. See **Details**.
win	Window in which to simulate the pattern. An object of class "owin" or "box3" or something acceptable to as.owin or as.box3. The default window is the unit square in two dimensions, unless x.init is specified, when the default window is the window of x.init.

giveup	Number of rejected proposals after which the algorithm should terminate.
x.init	Optional. Initial configuration of points. A point pattern in two or three dimensions (object of class "ppp" or "pp3"). The pattern returned by rSSI consists of this pattern together with the points added via simple sequential inhibition. See **Details**.
...	Ignored.
f, fmax	Optional arguments passed to rpoint to specify a non-uniform probability density for the random points.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
verbose	Logical value specifying whether to print progress reports when nsim > 1.

Details

This algorithm generates one or more realisations of the Simple Sequential Inhibition point process inside the window win.

The window win may be a two-dimensional region (object of class "owin") or a three-dimensional box (object of class "box3"). Accordingly the result will be a point pattern in two- or three-dimensional space.

Starting with an empty window (or with the point pattern x.init if specified), the algorithm adds points one-by-one. Each new point is generated uniformly in the window and independently of preceding points. If the new point lies closer than r units from an existing point, then it is rejected and another random point is generated. The algorithm terminates when either

(a) the desired number n of points is reached, or

(b) the current point configuration has not changed for giveup iterations, suggesting that it is no longer possible to add new points.

If n is infinite (the default) then the algorithm terminates only when (b) occurs. The result is sometimes called a *Random Sequential Packing*.

Note that argument n specifies the maximum permitted **total** number of points in the pattern returned by rSSI(). If x.init is not NULL then the number of points that are *added* is at most n - npoints(x.init) if n is finite.

Thus if x.init is not NULL then argument n must be at least as large as npoints(x.init), otherwise an error is given. If n==npoints(x.init) then a warning is given and the call to rSSI() has no real effect; x.init is returned.

There is no requirement that the points of x.init be at a distance at least r from each other. All of the *added* points will be at a distance at least r from each other and from any point of x.init.

The points will be generated inside the window win and the result will be a point pattern in the same window.

The default window is the unit square, win = square(1), unless x.init is specified, when the default is win=domain(x.init), the window of x.init.

If both win and x.init are specified, and if the two windows are different, then a warning will be issued. Any points of x.init lying outside win will be removed, with a warning.

Value

A point pattern in two or three dimensional space (an object of class "ppp" or "pp3") if nsim=1 and drop=TRUE, otherwise a list of point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

rpoispp, rMaternI, rMaternII.

Examples

```
Vinf <- rSSI(0.07)

V100 <- rSSI(0.07, 100)

X <- runifpoint(100)
Y <- rSSI(0.03,142,x.init=X) # Y consists of X together with
                             # 42 added points.
if(online <- interactive()) {
  plot(Y, main="rSSI")
  plot(X,add=TRUE,chars=20,cols="red")
}

## inhomogeneous
Z <- rSSI(0.07, 50, f=function(x,y){x})
if(online) plot(Z)

## three dimensions
A <- rSSI(0.05, 20, box3())
if(online) plot(A)
```

---

rstrat *Simulate Stratified Random Point Pattern*

Description

Generates a "stratified random" pattern of points in a window, by dividing the window into rectangular tiles and placing k random points independently in each tile.

Usage

```
rstrat(win=square(1), nx, ny=nx, k = 1, nsim=1, drop=TRUE)
```

Arguments

win	A window. An object of class owin, or data in any format acceptable to as.owin().
nx	Number of tiles in each column.
ny	Number of tiles in each row.
k	Number of random points to generate in each tile.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a random pattern of points in a "stratified random" sampling design. It can be useful for generating random spatial sampling points.

The bounding rectangle of win is divided into a regular $nx \times ny$ grid of rectangular tiles. In each tile, k random points are generated independently with a uniform distribution in that tile.

Some of these grid points may lie outside the window win: if they do, they are deleted.

The result is a point pattern inside the window win.

This function is useful in creating dummy points for quadrature schemes (see quadscheme) as well as in simulating random point patterns.

Value

A point pattern (an object of class "ppp") if nsim=1, or a list of point patterns if nsim > 1.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>.

See Also

rsyst, runifpoint, quadscheme

Examples

```
X <- rstrat(nx=10)
plot(X)

# polygonal boundary
X <- rstrat(letterR, 5, 10, k=3)
plot(X)
```

rStrauss *Perfect Simulation of the Strauss Process*

Description

Generate a random pattern of points, a simulated realisation of the Strauss process, using a perfect simulation algorithm.

Usage

```
rStrauss(beta, gamma = 1, R = 0, W = owin(), expand=TRUE, nsim=1, drop=TRUE)
```

Arguments

beta	intensity parameter (a positive number).
gamma	interaction parameter (a number between 0 and 1, inclusive).
R	interaction radius (a non-negative number).
W	window (object of class "owin") in which to generate the random pattern.
expand	Logical. If FALSE, simulation is performed in the window W, which must be rectangular. If TRUE (the default), simulation is performed on a larger window, and the result is clipped to the original window W. Alternatively expand can be an object of class "rmhexpand" (see rmhexpand) determining the expansion method.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a realisation of the Strauss point process in the window W using a 'perfect simulation' algorithm.

The Strauss process (Strauss, 1975; Kelly and Ripley, 1976) is a model for spatial inhibition, ranging from a strong 'hard core' inhibition to a completely random pattern according to the value of gamma.

The Strauss process with interaction radius R and parameters β and γ is the pairwise interaction point process with probability density

$$f(x_1, \ldots, x_n) = \alpha \beta^{n(x)} \gamma^{s(x)}$$

where $x_1, \ldots, x_n$ represent the points of the pattern, $n(x)$ is the number of points in the pattern, $s(x)$ is the number of distinct unordered pairs of points that are closer than R units apart, and α is the normalising constant. Intuitively, each point of the pattern contributes a factor β to the probability density, and each pair of points closer than r units apart contributes a factor γ to the density.

The interaction parameter γ must be less than or equal to 1 in order that the process be well-defined (Kelly and Ripley, 1976). This model describes an "ordered" or "inhibitive" pattern. If $\gamma = 1$ it reduces to a Poisson process (complete spatial randomness) with intensity β. If $\gamma = 0$ it is called a

"hard core process" with hard core radius $R/2$, since no pair of points is permitted to lie closer than R units apart.

The simulation algorithm used to generate the point pattern is 'dominated coupling from the past' as implemented by Berthelsen and Møller (2002, 2003). This is a 'perfect simulation' or 'exact simulation' algorithm, so called because the output of the algorithm is guaranteed to have the correct probability distribution exactly (unlike the Metropolis-Hastings algorithm used in rmh, whose output is only approximately correct).

There is a tiny chance that the algorithm will run out of space before it has terminated. If this occurs, an error message will be generated.

Value

If `nsim = 1`, a point pattern (object of class `"ppp"`). If `nsim > 1`, a list of point patterns.

Author(s)

Kasper Klitgaard Berthelsen, adapted for **spatstat** by Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

References

Berthelsen, K.K. and Møller, J. (2002) A primer on perfect simulation for spatial point processes. *Bulletin of the Brazilian Mathematical Society* 33, 351-367.

Berthelsen, K.K. and Møller, J. (2003) Likelihood and non-parametric Bayesian MCMC inference for spatial point processes based on perfect simulation and path sampling. *Scandinavian Journal of Statistics* 30, 549-564.

Kelly, F.P. and Ripley, B.D. (1976) On Strauss's model for clustering. *Biometrika* **63**, 357–360.

Møller, J. and Waagepetersen, R. (2003). *Statistical Inference and Simulation for Spatial Point Processes.* Chapman and Hall/CRC.

Strauss, D.J. (1975) A model for clustering. *Biometrika* **62**, 467–475.

See Also

rmh, Strauss, rHardcore, rStraussHard, rDiggleGratton, rDGS, rPenttinen.

Examples

```
X <- rStrauss(0.05,0.2,1.5,square(50))
```

rStraussHard *Perfect Simulation of the Strauss-Hardcore Process*

Description

Generate a random pattern of points, a simulated realisation of the Strauss-Hardcore process, using a perfect simulation algorithm.

Usage

```
rStraussHard(beta, gamma = 1, R = 0, H = 0, W = owin(),
             expand=TRUE, nsim=1, drop=TRUE)
```

Arguments

beta	intensity parameter (a positive number).
gamma	interaction parameter (a number between 0 and 1, inclusive).
R	interaction radius (a non-negative number).
H	hard core distance (a non-negative number smaller than R).
W	window (object of class "owin") in which to generate the random pattern. Currently this must be a rectangular window.
expand	Logical. If FALSE, simulation is performed in the window W, which must be rectangular. If TRUE (the default), simulation is performed on a larger window, and the result is clipped to the original window W. Alternatively expand can be an object of class "rmhexpand" (see rmhexpand) determining the expansion method.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This function generates a realisation of the Strauss-Hardcore point process in the window W using a 'perfect simulation' algorithm.

The Strauss-Hardcore process is described in StraussHard.

The simulation algorithm used to generate the point pattern is 'dominated coupling from the past' as implemented by Berthelsen and Møller (2002, 2003). This is a 'perfect simulation' or 'exact simulation' algorithm, so called because the output of the algorithm is guaranteed to have the correct probability distribution exactly (unlike the Metropolis-Hastings algorithm used in rmh, whose output is only approximately correct).

A limitation of the perfect simulation algorithm is that the interaction parameter γ must be less than or equal to 1. To simulate a Strauss-hardcore process with $\gamma > 1$, use rmh.

There is a tiny chance that the algorithm will run out of space before it has terminated. If this occurs, an error message will be generated.

Value

If `nsim = 1`, a point pattern (object of class `"ppp"`). If `nsim > 1`, a list of point patterns.

Author(s)

Kasper Klitgaard Berthelsen and Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

References

Berthelsen, K.K. and Møller, J. (2002) A primer on perfect simulation for spatial point processes. *Bulletin of the Brazilian Mathematical Society* 33, 351-367.

Berthelsen, K.K. and Møller, J. (2003) Likelihood and non-parametric Bayesian MCMC inference for spatial point processes based on perfect simulation and path sampling. *Scandinavian Journal of Statistics* 30, 549-564.

Møller, J. and Waagepetersen, R. (2003). *Statistical Inference and Simulation for Spatial Point Processes.* Chapman and Hall/CRC.

See Also

`rmh`, `StraussHard`.

`rHardcore`, `rStrauss`, `rDiggleGratton`, `rDGS`, `rPenttinen`.

Examples

```
Z <- rStraussHard(100,0.7,0.05,0.02)
Y <- rStraussHard(100,0.7,0.05,0.01, nsim=2)
```

`rtemper` *Simulated Annealing or Simulated Tempering for Gibbs Point Processes*

Description

Performs simulated annealing or simulated tempering for a Gibbs point process model using a specified annealing schedule.

Usage

```
rtemper(model, invtemp, nrep, ..., track=FALSE, start = NULL, verbose = FALSE)
```

Arguments

model	A Gibbs point process model: a fitted Gibbs point process model (object of class "ppm"), or any data acceptable to rmhmodel.
invtemp	A numeric vector of positive numbers. The sequence of values of inverse temperature that will be used.
nrep	An integer vector of the same length as invtemp. The value nrep[i] specifies the number of steps of the Metropolis-Hastings algorithm that will be performed at inverse temperature invtemp[i].
start	Initial starting state for the simulation. Any data acceptable to rmhstart.
track	Logical flag indicating whether to save the transition history of the simulations.
...	Additional arguments passed to rmh.default.
verbose	Logical value indicating whether to print progress reports.

Details

The Metropolis-Hastings simulation algorithm rmh is run for nrep[1] steps at inverse temperature invtemp[1], then for nrep[2] steps at inverse temperature invtemp[2], and so on.

Setting the inverse temperature to a value α means that the probability density of the Gibbs model, $f(x)$, is replaced by $g(x) = C\, f(x)^\alpha$ where C is a normalising constant depending on α. Larger values of α exaggerate the high and low values of probability density, while smaller values of α flatten out the probability density.

For example if the original model is a Strauss process, the modified model is close to a hard core process for large values of inverse temperature, and close to a Poisson process for small values of inverse temperature.

Value

A point pattern (object of class "ppp").

If track=TRUE, the result also has an attribute "history" which is a data frame with columns proposaltype, accepted, numerator and denominator, as described in rmh.default.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

rmh.default, rmh.

Examples

```
stra <- rmhmodel(cif="strauss",
                 par=list(beta=2,gamma=0.2,r=0.7),
                 w=square(10))
nr <- if(interactive()) 1e5 else 1e3
Y <- rtemper(stra, c(1, 2, 4, 8), nr * (1:4), verbose=TRUE, track=TRUE)
```

rthin *Random Thinning*

Description

Applies independent random thinning to a point pattern or segment pattern.

Usage

```
rthin(X, P, ..., nsim=1, drop=TRUE)
```

Arguments

X	A point pattern (object of class "ppp" or "lpp" or "pp3" or "ppx") or line segment pattern (object of class "psp") that will be thinned.
P	Data giving the retention probabilities, i.e. the probability that each point or line in X will be retained. Either a single number, or a vector of numbers, or a function(x,y) in the R language, or a function object (class "funxy" or "linfun"), or a pixel image (object of class "im" or "linim").
...	Additional arguments passed to P, if it is a function.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

In a random thinning operation, each point of the point pattern X is randomly either deleted or retained (i.e. not deleted). The result is a point pattern, consisting of those points of X that were retained.

Independent random thinning means that the retention/deletion of each point is independent of other points.

The argument P determines the probability of **retaining** each point. It may be

a single number, so that each point will be retained with the same probability P;

a vector of numbers, so that the ith point of X will be retained with probability P[i];

a function P(x,y)**,** so that a point at a location (x,y) will be retained with probability P(x,y);

an object of class "funxy" **or** "linfun"**,** so that points in the pattern X will be retained with probabilities P(X);

a pixel image, containing values of the retention probability for all locations in a region encompassing the point pattern.

If P is a function P(x,y), it should be 'vectorised', that is, it should accept vector arguments x,y and should yield a numeric vector of the same length. The function may have extra arguments which are passed through the ... argument.

Value

An object of the same kind as X if nsim=1, or a list of such objects if nsim > 1.

Reproducibility

The algorithm for random thinning was changed in **spatstat** version 1.42-3. Set spatstat.options(fastthin=FALSE) to use the previous, slower algorithm, if it is desired to reproduce results obtained with earlier versions.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

Examples

```
plot(redwood, main="thinning")

# delete 20% of points
Y <- rthin(redwood, 0.8)
points(Y, col="green", cex=1.4)

# function
f <- function(x,y) { ifelse(x < 0.4, 1, 0.5) }
Y <- rthin(redwood, f)

# pixel image
Z <- as.im(f, Window(redwood))
Y <- rthin(redwood, Z)


# thin other kinds of patterns
E <- rthin(osteo$pts[[1]], 0.6)
L <- rthin(copper$Lines, 0.5)
```

---

rthinclumps *Random Thinning of Clumps*

Description

Finds the topologically-connected clumps of a spatial region and randomly deletes some of the clumps.

Usage

```
rthinclumps(W, p, ...)
```

Arguments

W	Window (object of class "owin" or pixel image (object of class "im").
p	Probability of *retaining* each clump. A single number between 0 and 1.
...	Additional arguments passed to connected.im or connected.owin to determine the connected clumps.

Details

The argument W specifies a region of space, typically consisting of several clumps that are not connected to each other. The algorithm randomly deletes or retains each clump. The fate of each clump is independent of other clumps.

If W is a spatial window (class "owin") then it will be divided into clumps using connected.owin. Each clump will either be retained (with probability p) or deleted in its entirety (with probability 1-p).

If W is a pixel image (class "im") then its domain will be divided into clumps using connected.im. The default behaviour depends on the type of pixel values. If the pixel values are logical, then the spatial region will be taken to consist of all pixels whose value is TRUE. Otherwise, the spatial region is taken to consist of all pixels whose value is defined (i.e. not equal to NA). This behaviour can be changed using the argument background passed to connected.im.

The result is a window comprising all the clumps that were retained.

Value

Window (object of class "owin").

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

rthin for thinning other kinds of objects.

Examples

```
A <- (distmap(cells) < 0.06)
opa <- par(mfrow=c(1,2))
plot(A)
plot(rthinclumps(A, 0.5))
par(opa)
```

rThomas *Simulate Thomas Process*

Description

Generate a random point pattern, a simulated realisation of the Thomas cluster process.

Usage

```
rThomas(kappa, scale, mu, win = square(1),
          nsim=1, drop=TRUE,
          ...,
          n.cond=NULL, w.cond=NULL,
          algorithm=c("BKBC", "naive"),
          nonempty=TRUE,
          poisthresh=1e-6,
          expand = 4*scale,
          saveparents=FALSE, saveLambda=FALSE,
          kappamax=NULL, mumax=NULL, LambdaOnly=FALSE, sigma)
```

Arguments

kappa	Intensity of the Poisson process of cluster centres. A single positive number, a function, or a pixel image.
scale	Cluster size. Standard deviation of random displacement (along each coordinate axis) of a point from its cluster centre. A single positive number.
mu	Mean number of points per cluster (a single positive number) or reference intensity for the cluster points (a function or a pixel image).
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
...	Passed to clusterfield to control the image resolution when saveLambda=TRUE.
n.cond	Optional. Integer specifying a fixed number of points. See the section on *Conditional Simulation*.
w.cond	Optional. Conditioning region. A window (object of class "owin") specifying the region which must contain exactly n.cond points. See the section on *Conditional Simulation*.
algorithm	String (partially matched) specifying the simulation algorithm. See Details.
nonempty	Logical. If TRUE (the default), a more efficient algorithm is used, in which parents are generated conditionally on having at least one offspring point in the window. If FALSE, parents are generated even if they have no offspring in the window. The default is recommended unless you need to simulate all the parent points for some other purpose.

poisthresh	Numerical threshold below which the model will be treated as a Poisson process. See Details.
expand	Window expansion distance. A single number. The distance by which the original window will be expanded in order to generate parent points. Has a sensible default.
saveparents	Logical value indicating whether to save the locations of the parent points as an attribute.
saveLambda	Logical. If TRUE then the random intensity corresponding to the simulated parent points will also be calculated and saved, and returned as an attribute of the point pattern.
kappamax	Optional. Numerical value which is an upper bound for the values of kappa, when kappa is a pixel image or a function.
mumax	Optional. Numerical value which is an upper bound for the values of mu, when mu is a pixel image or a function.
LambdaOnly	Logical value specifying whether to return only the random intensity, rather than the point pattern.
sigma	Deprecated. Equivalent to scale.

Details

This algorithm generates a realisation of the ('modified') Thomas process, a special case of the Neyman-Scott process, inside the window win.

In the simplest case, where kappa and mu are single numbers, the cluster process is formed by first generating a uniform Poisson point process of "parent" points with intensity kappa. Then each parent point is replaced by a random cluster of "offspring" points, the number of points per cluster being Poisson (mu) distributed, and their positions being isotropic Gaussian displacements from the cluster parent location. The resulting point pattern is a realisation of the classical "stationary Thomas process" generated inside the window win. This point process has intensity kappa * mu.

Note that, for correct simulation of the model, the parent points are not restricted to lie inside the window win; the parent process is effectively the uniform Poisson process on the infinite plane.

The algorithm can also generate spatially inhomogeneous versions of the Thomas process:

- The parent points can be spatially inhomogeneous. If the argument kappa is a function(x,y) or a pixel image (object of class "im"), then it is taken as specifying the intensity function of an inhomogeneous Poisson process that generates the parent points.
- The offspring points can be inhomogeneous. If the argument mu is a function(x,y) or a pixel image (object of class "im"), then it is interpreted as the reference density for offspring points, in the sense of Waagepetersen (2007). For a given parent point, the offspring constitute a Poisson process with intensity function equal to mu * f, where f is the Gaussian probability density centred at the parent point. Equivalently we first generate, for each parent point, a Poisson (mumax) random number of offspring (where M is the maximum value of mu) with independent Gaussian displacements from the parent location, and then randomly thin the offspring points, with retention probability mu/M.
- Both the parent points and the offspring points can be spatially inhomogeneous, as described above.

Note that if kappa is a pixel image, its domain must be larger than the window win. This is because an offspring point inside win could have its parent point lying outside win. In order to allow this, the simulation algorithm first expands the original window win by a distance expand and generates the Poisson process of parent points on this larger window. If kappa is a pixel image, its domain must contain this larger window.

The intensity of the Thomas process is kappa * mu if either kappa or mu is a single number. In the general case the intensity is an integral involving kappa, mu and f.

If the pair correlation function of the model is very close to that of a Poisson process, deviating by less than poisthresh, then the model is approximately a Poisson process, and will be simulated as a Poisson process with intensity kappa * mu, using rpoispp. This avoids computations that would otherwise require huge amounts of memory.

Value

A point pattern (object of class "ppp") or a list of point patterns.

Additionally, some intermediate results of the simulation are returned as attributes of this point pattern (see rNeymanScott). Furthermore, the simulated intensity function is returned as an attribute "Lambda", if saveLambda=TRUE.

If LambdaOnly=TRUE the result is a pixel image (object of class "im") or a list of pixel images.

Simulation Algorithm

Two simulation algorithms are implemented.

- The *naive* algorithm generates the cluster process by directly following the description given above. First the window win is expanded by a distance equal to expand. Then the parent points are generated in the expanded window according to a Poisson process with intensity kappa. Then each parent point is replaced by a finite cluster of offspring points as described above. The naive algorithm is used if algorithm="naive" or if nonempty=FALSE.
- The *BKBC* algorithm, proposed by Baddeley and Chang (2023), is a modification of the algorithm of Brix and Kendall (2002). Parents are generated in the infinite plane, subject to the condition that they have at least one offspring point inside the window win. The BKBC algorithm is used when algorithm="BKBC" (the default) and nonempty=TRUE (the default).

The naive algorithm becomes very slow when scale is large, while the BKBC algorithm is uniformly fast (Baddeley and Chang, 2023).

If saveparents=TRUE, then the simulated point pattern will have an attribute "parents" containing the coordinates of the parent points, and an attribute "parentid" mapping each offspring point to its parent.

If nonempty=TRUE (the default), then parents are generated subject to the condition that they have at least one offspring point in the window win. nonempty=FALSE, then parents without offspring will be included; this option is not available in the *BKBC* algorithm.

Note that if kappa is a pixel image, its domain must be larger than the window win. This is because an offspring point inside win could have its parent point lying outside win. In order to allow this, the naive simulation algorithm first expands the original window win by a distance equal to expand and generates the Poisson process of parent points on this larger window. If kappa is a pixel image, its domain must contain this larger window.

If the pair correlation function of the model is very close to that of a Poisson process, with maximum deviation less than `poisthresh`, then the model is approximately a Poisson process. This is detected by the naive algorithm which then simulates a Poisson process with intensity `kappa * mu`, using `rpoispp`. This avoids computations that would otherwise require huge amounts of memory.

Conditional Simulation

If `n.cond` is specified, it should be a single integer. Simulation will be conditional on the event that the pattern contains exactly `n.cond` points (or contains exactly `n.cond` points inside the region `w.cond` if it is given).

Conditional simulation uses the rejection algorithm described in Section 6.2 of Moller, Syversveen and Waagepetersen (1998). There is a maximum number of proposals which will be attempted. Consequently the return value may contain fewer than `nsim` point patterns.

The current algorithm for conditional simulation ignores the argument `saveparents` and does not save the parent points.

Fitting cluster models to data

The Thomas model with homogeneous parents (i.e. where `kappa` is a single number) where the offspring are either homogeneous or inhomogeneous (`mu` is a single number, a function or pixel image) can be fitted to point pattern data using `kppm`, or fitted to the inhomogeneous K function using `thomas.estK` or `thomas.estpcf`.

Currently **spatstat** does not support fitting the Thomas cluster process model with inhomogeneous parents.

A Thomas cluster process model fitted by `kppm` can be simulated automatically using `simulate.kppm` (which invokes `rThomas` to perform the simulation).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ya-Mei Chang <yamei628@gmail.com>.

References

Baddeley, A. and Chang, Y.-M. (2023) Robust algorithms for simulating cluster point processes. *Journal of Statistical Computation and Simulation* **93**, 1950–1975.

Brix, A. and Kendall, W.S. (2002) Simulation of cluster point processes without edge effects. *Advances in Applied Probability* **34**, 267–280.

Diggle, P. J., Besag, J. and Gleaves, J. T. (1976) Statistical analysis of spatial point patterns by means of distance methods. *Biometrics* **32** 659–667.

Møller, J., Syversveen, A. and Waagepetersen, R. (1998) Log Gaussian Cox Processes. *Scandinavian Journal of Statistics* **25**, 451–482.

Thomas, M. (1949) A generalisation of Poisson's binomial limit for use in ecology. *Biometrika* **36**, 18–25.

Waagepetersen, R. (2007) An estimating function approach to inference for inhomogeneous Neyman-Scott processes. *Biometrics* **63**, 252–258.

See Also

rpoispp, rMatClust, rCauchy, rVarGamma, rNeymanScott, rGaussPoisson.

For fitting the model, see kppm, clusterfit.

Examples

```
#homogeneous
X <- rThomas(10, 0.2, 5)
#inhomogeneous
Z <- as.im(function(x,y){ 5 * exp(2 * x - 1) }, owin())
Y <- rThomas(10, 0.2, Z)
```

---

runifdisc *Generate N Uniform Random Points in a Disc*

Description

Generate a random point pattern containing n independent uniform random points in a circular disc.

Usage

```
runifdisc(n, radius=1, centre=c(0,0), ..., nsim=1, drop=TRUE, boxed=FALSE)
```

Arguments

n	Number of points.
radius	Radius of the circle.
centre	Coordinates of the centre of the circle.
...	Arguments passed to disc controlling the accuracy of approximation to the circle.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
boxed	Logical value indicating whether to replace the disc window by a square window, for computational efficiency.

Details

This function generates n independent random points, uniformly distributed in a circular disc.

It is faster (for a circular window) than the general code used in runifpoint.

To generate random points in an ellipse, first generate points in a circle using runifdisc, then transform to an ellipse using affine, as shown in the examples.

To generate random points in other windows, use runifpoint. To generate non-uniform random points, use rpoint.

Value

A point pattern (an object of class "ppp") if nsim=1, or a list of point patterns if nsim > 1.

Speed of computation

The random point coordinates are generated by a very fast algorithm.

Computation time includes the time taken to construct the circular window containing the resulting point pattern, which is generated by disc. Arguments passed to disc control the accuracy of the polygonal approximation to the circle, and therefore affect the computation time.

If boxed=TRUE, the circular window is not constructed, and is replaced by a square window, which is much faster to compute. The random points are still generated uniformly inside the disc by the same fast algorithm.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>.

See Also

disc, runifpoint, rpoint

Examples

```
# 100 random points in the unit disc
plot(runifdisc(100))
# 42 random points in the ellipse with major axis 3 and minor axis 1
X <- runifdisc(42)
Y <- affine(X, mat=diag(c(3,1)))
plot(Y)
```

---

runifpoint *Generate N Uniform Random Points*

Description

Generate a random point pattern containing n independent uniform random points.

Usage

```
runifpoint(n, win=owin(c(0,1),c(0,1)), giveup=1000, warn=TRUE, ...,
           fail.action=c("error", "pass", "missing"),
           nsim=1, drop=TRUE, ex=NULL)
```

Arguments

n	Number of points.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin. (Alternatively a tessellation; see the section on tessellations).
giveup	Number of attempts in the rejection method after which the algorithm should stop trying to generate new points.
warn	Logical. Whether to issue a warning if n is very large. See Details.
...	Ignored.
fail.action	Character string (partially matched) specifying what to do if the number of attempts exceeds giveup. If fail.action="error" (the default), a fatal error will be generated. If fail.action="pass", the pattern of accepted points (containing fewer than n points) will be returned. If fail.action="missing", the result will be a 'missing point pattern', an object belonging to the classes "NAobject" and "ppp".
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
ex	Optional. A point pattern to use as the example. If ex is given and n and win are missing, then n and win will be calculated from the point pattern ex.

Details

This function generates n independent random points, uniformly distributed in the window win. (For nonuniform distributions, see rpoint.)

The algorithm depends on the type of window, as follows:

- If win is a rectangle then n independent random points, uniformly distributed in the rectangle, are generated by assigning uniform random values to their cartesian coordinates.
- If win is a binary image mask, then a random sequence of pixels is selected (using sample) with equal probabilities. Then for each pixel in the sequence we generate a uniformly distributed random point in that pixel.
- If win is a polygonal window, the algorithm uses the rejection method. It finds a rectangle enclosing the window, generates points in this rectangle, and tests whether they fall in the desired window. It gives up when giveup * n tests have been performed without yielding n successes.

The algorithm for binary image masks is faster than the rejection method but involves discretisation.

If warn=TRUE (the default), a warning will be issued if n is very large. The threshold is spatstat.options("huge.npoints")
This warning has no consequences, but it helps to trap a number of common errors.

Value

A point pattern (an object of class "ppp") or a list of point patterns.

Tessellation

The argument win may be a tessellation (object of class "tess", see tess). Then the specified number of points n will be randomly generated inside each tile of the tessellation. The argument n may be either a single integer, or an integer vector specifying the number of points to be generated in each individual tile. The result will be a point pattern in the window as.owin(win).

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

ppp.object, owin.object, rpoispp, rpoint

Examples

```
# 100 random points in the unit square
pp <- runifpoint(100)
# irregular window
letterR
# polygonal
pp <- runifpoint(100, letterR)
# binary image mask
pp <- runifpoint(100, as.mask(letterR))

# randomising an existing point pattern
runifpoint(npoints(cells), win=Window(cells))
runifpoint(ex=cells)

# tessellation
A <- quadrats(unit.square(), 2, 3)
# different numbers of points in each cell
X <- runifpoint(1:6, A)
```

---

runifpoint3 *Generate N Uniform Random Points in Three Dimensions*

Description

Generate a random point pattern containing n independent, uniform random points in three dimensions.

Usage

```
runifpoint3(n, domain = box3(), ..., nsim=1, drop=TRUE, ex=NULL)
```

Arguments

n	Number of points to be generated.
domain	Three-dimensional box in which the process should be generated. An object of class "box3".
...	Ignored.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
ex	An example point pattern (object of class "pp3") which will be used to determine the number of points n and the containing box domain.

Details

This function generates n independent random points, uniformly distributed in the three-dimensional box domain.

Value

If nsim = 1 and drop=TRUE, a point pattern in three dimensions (an object of class "pp3"). If nsim > 1, a list of such point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>.

See Also

rpoispp3, rpoint3, pp3, box3

Examples

```
X <- runifpoint3(50)

runifpoint3(ex=osteo$pts[[1]])
```

---

runifpointOnLines *Generate N Uniform Random Points On Line Segments*

Description

Given a line segment pattern, generate a random point pattern consisting of n points uniformly distributed on the line segments.

Usage

```
runifpointOnLines(n, L, nsim=1, drop=TRUE)
```

Arguments

n	Number of points to generate.
L	Line segment pattern (object of class "psp") on which the points should lie.
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.

Details

This command generates a point pattern consisting of n independent random points, each point uniformly distributed on the line segment pattern. This means that, for each random point,

- the probability of falling on a particular segment is proportional to the length of the segment; and
- given that the point falls on a particular segment, it has uniform probability density along that segment.

If n is a single integer, the result is an unmarked point pattern containing n points. If n is a vector of integers, the result is a marked point pattern, with m different types of points, where m = length(n), in which there are n[j] points of type j.

Value

If nsim = 1, a point pattern (object of class "ppp") with the same window as L. If nsim > 1, a list of point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>

See Also

psp, ppp, pointsOnLines, runifpoint

Examples

```
X <- psp(runif(10), runif(10), runif(10), runif(10),  window=owin())
Y <- runifpointOnLines(20, X)
plot(X, main="")
plot(Y, add=TRUE)
Z <- runifpointOnLines(c(5,5), X)
```

runifpointx *Generate N Uniform Random Points in Any Dimensions*

Description

Generate a random point pattern containing n independent, uniform random points in any number of spatial dimensions.

Usage

```
runifpointx(n, domain, nsim=1, drop=TRUE)
```

Arguments

n	Number of points to be generated.
domain	Multi-dimensional box in which the process should be generated. An object of class "boxx".
nsim	Number of simulated realisations to be generated.
drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a single point pattern.

Details

This function generates a pattern of n independent random points, uniformly distributed in the multi-dimensional box domain.

Value

If nsim = 1 and drop=TRUE, a point pattern (an object of class "ppx"). If nsim > 1 or drop=FALSE, a list of such point patterns.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Rolf Turner <rolfturner@posteo.net>.

See Also

rpoisppx, ppx, boxx

Examples

```
w <- boxx(x=c(0,1), y=c(0,1), z=c(0,1), t=c(0,3))
X <- runifpointx(50, w)
```

rUnround *Random Un-Rounding of Spatial Location*

Description

Given a point pattern in which the spatial coordinates have been discretised (rounded), randomly displace each point to reverse the effect of rounding.

Usage

```
rUnround(X, ...)

## S3 method for class 'ppp'
rUnround(X, ..., xstep=NULL, ystep=xstep,
    nsim = 1, drop=TRUE, giveup = 1000)
```

Arguments

X	Point pattern (object of class "ppp").
...	Arguments passed to as.mask specifying the discretisation, if xstep and ystep are not given.
xstep, ystep	Numeric values giving the resolution of the discretised coordinates.
giveup	Maximum number of attempts to place a point inside the window.
nsim	Number of simulated realisations of the unrounded pattern.
drop	Logical value specifying, when nsim=1, whether to return a point pattern (drop=TRUE, the default) or a list containing one point pattern (drop=FALSE).

Details

This function is similar to rjitter except that it is designed to reverse the effect of discretisation of coordinates.

The spatial coordinates of X are modified by adding independent, uniformly distributed random numbers to the x and y coordinates. If the points of X were all located at the centres of pixels in a pixel grid, then the points of rUnround(X) will be uniformly randomly distributed within the same pixels.

Value

Another point pattern (object of class "ppp") in the same window as X and with the same number of points as X.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>.

See Also

rjitter, discretise

Examples

```
X <- runifrect(30, Frame(letterR))[letterR]
Y <- discretise(X, eps=0.05)
Z <- rUnround(Y)
```

---

rVarGamma *Simulate Neyman-Scott Point Process with Variance Gamma cluster kernel*

Description

Generate a random point pattern, a simulated realisation of the Neyman-Scott process with Variance Gamma (Bessel) cluster kernel.

Usage

```
rVarGamma(kappa, scale, mu,
          nu,
          win = square(1),
          nsim=1, drop=TRUE,
          ...,
          n.cond = NULL, w.cond = NULL,
          algorithm=c("BKBC", "naive"),
          nonempty=TRUE,
          thresh = 0.001,
          poisthresh=1e-6,
          expand = NULL,
          saveparents=FALSE, saveLambda=FALSE,
          kappamax=NULL, mumax=NULL, LambdaOnly=FALSE)
```

Arguments

kappa	Intensity of the Poisson process of cluster centres. A single positive number, a function, or a pixel image.
scale	Scale parameter for cluster kernel. Determines the size of clusters. A single positive number, in the same units as the spatial coordinates.
mu	Mean number of points per cluster (a single positive number) or reference intensity for the cluster points (a function or a pixel image).
nu	Shape parameter for the cluster kernel. A number greater than -1.
win	Window in which to simulate the pattern. An object of class "owin" or something acceptable to as.owin.
nsim	Number of simulated realisations to be generated.

drop	Logical. If nsim=1 and drop=TRUE (the default), the result will be a point pattern, rather than a list containing a point pattern.
...	Passed to clusterfield to control the image resolution when saveLambda=TRUE, and to clusterradius when expand is missing or NULL.
n.cond	Optional. Integer specifying a fixed number of points. See the section on *Conditional Simulation*.
w.cond	Optional. Conditioning region. A window (object of class "owin") specifying the region which must contain exactly n.cond points. See the section on *Conditional Simulation*.
algorithm	String (partially matched) specifying the simulation algorithm. See Details.
nonempty	Logical. If TRUE (the default), a more efficient algorithm is used, in which parents are generated conditionally on having at least one offspring point. If FALSE, parents are generated even if they have no offspring. Both choices are valid; the default is recommended unless you need to simulate all the parent points for some other purpose.
thresh	Threshold relative to the cluster kernel value at the origin (parent location) determining when the cluster kernel will be treated as zero for simulation purposes. Will be overridden by argument expand if that is given.
poisthresh	Numerical threshold below which the model will be treated as a Poisson process. See Details.
expand	Window expansion distance. A single number. The distance by which the original window will be expanded in order to generate parent points. Has a sensible default, determined by calling clusterradius with the numeric threshold value given in thresh.
saveparents	Logical value indicating whether to save the locations of the parent points as an attribute.
saveLambda	Logical. If TRUE then the random intensity corresponding to the simulated parent points will also be calculated and saved, and returns as an attribute of the point pattern.
kappamax	Optional. Numerical value which is an upper bound for the values of kappa, when kappa is a pixel image or a function.
mumax	Optional. Numerical value which is an upper bound for the values of mu, when mu is a pixel image or a function.
LambdaOnly	Logical value specifying whether to return only the random intensity, rather than the point pattern.

Details

This algorithm generates a realisation of the Neyman-Scott process with Variance Gamma (Bessel) cluster kernel, inside the window win.

The process is constructed by first generating a Poisson point process of "parent" points with intensity kappa. Then each parent point is replaced by a random cluster of points, the number of points in each cluster being random with a Poisson (mu) distribution, and the points being placed independently and uniformly according to a Variance Gamma kernel.

Note that, for correct simulation of the model, the parent points are not restricted to lie inside the window win; the parent process is effectively the uniform Poisson process on the infinite plane.

The shape of the kernel is determined by the dimensionless index nu. This is the parameter nu' (nu-prime) $\nu' = \alpha/2 - 1$ appearing in equation (12) on page 126 of Jalilian et al (2013).

The scale of the kernel is determined by the argument scale, which is the parameter η appearing in equations (12) and (13) of Jalilian et al (2013). It is expressed in units of length (the same as the unit of length for the window win).

The algorithm can also generate spatially inhomogeneous versions of the cluster process:

- The parent points can be spatially inhomogeneous. If the argument kappa is a function(x,y) or a pixel image (object of class "im"), then it is taken as specifying the intensity function of an inhomogeneous Poisson process that generates the parent points.
- The offspring points can be inhomogeneous. If the argument mu is a function(x,y) or a pixel image (object of class "im"), then it is interpreted as the reference density for offspring points, in the sense of Waagepetersen (2006).

If the pair correlation function of the model is very close to that of a Poisson process, deviating by less than poisthresh, then the model is approximately a Poisson process, and will be simulated as a Poisson process with intensity kappa * mu, using rpoispp. This avoids computations that would otherwise require huge amounts of memory.

Value

A point pattern (object of class "ppp") or a list of point patterns.

Additionally, some intermediate results of the simulation are returned as attributes of this point pattern (see rNeymanScott). Furthermore, the simulated intensity function is returned as an attribute "Lambda", if saveLambda=TRUE.

If LambdaOnly=TRUE the result is a pixel image (object of class "im") or a list of pixel images.

Simulation Algorithm

Two simulation algorithms are implemented.

- The *naive* algorithm generates the cluster process by directly following the description given above. First the window win is expanded by a distance equal to expand. Then the parent points are generated in the expanded window according to a Poisson process with intensity kappa. Then each parent point is replaced by a finite cluster of offspring points as described above. The naive algorithm is used if algorithm="naive" or if nonempty=FALSE.
- The *BKBC* algorithm, proposed by Baddeley and Chang (2023), is a modification of the algorithm of Brix and Kendall (2002). Parents are generated in the infinite plane, subject to the condition that they have at least one offspring point inside the window win. The BKBC algorithm is used when algorithm="BKBC" (the default) and nonempty=TRUE (the default).

The naive algorithm becomes very slow when scale is large, while the BKBC algorithm is uniformly fast (Baddeley and Chang, 2023).

If saveparents=TRUE, then the simulated point pattern will have an attribute "parents" containing the coordinates of the parent points, and an attribute "parentid" mapping each offspring point to its parent.

If nonempty=TRUE (the default), then parents are generated subject to the condition that they have at least one offspring point in the window win. nonempty=FALSE, then parents without offspring will be included; this option is not available in the *BKBC* algorithm.

Note that if kappa is a pixel image, its domain must be larger than the window win. This is because an offspring point inside win could have its parent point lying outside win. In order to allow this, the naive simulation algorithm first expands the original window win by a distance equal to expand and generates the Poisson process of parent points on this larger window. If kappa is a pixel image, its domain must contain this larger window.

If the pair correlation function of the model is very close to that of a Poisson process, with maximum deviation less than poisthresh, then the model is approximately a Poisson process. This is detected by the naive algorithm which then simulates a Poisson process with intensity kappa * mu, using rpoispp. This avoids computations that would otherwise require huge amounts of memory.

Conditional Simulation

If n.cond is specified, it should be a single integer. Simulation will be conditional on the event that the pattern contains exactly n.cond points (or contains exactly n.cond points inside the region w.cond if it is given).

Conditional simulation uses the rejection algorithm described in Section 6.2 of Moller, Syversveen and Waagepetersen (1998). There is a maximum number of proposals which will be attempted. Consequently the return value may contain fewer than nsim point patterns.

The current algorithm for conditional simulation ignores the argument saveparents and does not save the parent points.

Fitting cluster models to data

The Variance-Gamma cluster model with homogeneous parents (i.e. where kappa is a single number) where the offspring are either homogeneous or inhomogeneous (mu is a single number, a function or pixel image) can be fitted to point pattern data using kppm, or fitted to the inhomogeneous K function using vargamma.estK or vargamma.estpcf.

Currently **spatstat** does not support fitting the Variance-Gamma cluster process model with inhomogeneous parents.

A Variance-Gamma cluster process model fitted by kppm can be simulated automatically using simulate.kppm (which invokes rVarGamma to perform the simulation).

Warning

The argument nu is the parameter ν' (nu-prime) $\nu' = \alpha/2 - 1$ appearing in equation (12) on page 126 of Jalilian et al (2013). This is different from the parameter called ν appearing in equation(14) on page 127 of Jalilian et al (2013), defined by $\nu = \alpha - 1$. This has been a frequent source of confusion.

Author(s)

Original algorithm by Abdollah Jalilian and Rasmus Waagepetersen. Adapted for **spatstat** by Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>. Brix-Kendall-Baddeley-Chang algorithm implemented by Adrian Baddeley <Adrian.Baddeley@curtin.edu.au> and Ya-Mei Chang <yamei628@gmail.com>.

References

Baddeley, A. and Chang, Y.-M. (2023) Robust algorithms for simulating cluster point processes. *Journal of Statistical Computation and Simulation* **93**, 1950–1975.

Brix, A. and Kendall, W.S. (2002) Simulation of cluster point processes without edge effects. *Advances in Applied Probability* **34**, 267–280.

Jalilian, A., Guan, Y. and Waagepetersen, R. (2013) Decomposition of variance for spatial Cox processes. *Scandinavian Journal of Statistics* **40**, 119-137.

Møller, J., Syversveen, A. and Waagepetersen, R. (1998) Log Gaussian Cox Processes. *Scandinavian Journal of Statistics* **25**, 451–482.

Waagepetersen, R. (2007) An estimating function approach to inference for inhomogeneous Neyman-Scott processes. *Biometrics* **63**, 252–258.

See Also

rpoispp, rMatClust, rThomas, rCauchy, rNeymanScott, rGaussPoisson.

For fitting the model, see kppm, clusterfit, vargamma.estK, vargamma.estpcf.

Examples

```
# homogeneous
X <- rVarGamma(kappa=5, scale=2, mu=5, nu=-1/4)
# inhomogeneous
ff <- function(x,y){ exp(2 - 3 * abs(x)) }
fmax <- exp(2)
Z <- as.im(ff, W= owin())
Y <- rVarGamma(kappa=5, scale=2, mu=Z, nu=0)
YY <- rVarGamma(kappa=ff, scale=2, mu=3, nu=0, kappamax=fmax)
```

---

update.rmhcontrol *Update Control Parameters of Metropolis-Hastings Algorithm*

Description

update method for class "rmhcontrol".

Usage

```
  ## S3 method for class 'rmhcontrol'
update(object, ...)
```

Arguments

object	Object of class "rmhcontrol" containing control parameters for a Metropolis-Hastings algorithm.
...	Arguments to be updated in the new call to rmhcontrol.

Details

This is a method for the generic function update for the class "rmhcontrol". An object of class "rmhcontrol" describes a set of control parameters for the Metropolis-Hastings simulation algorithm. See rmhcontrol).

update.rmhcontrol will modify the parameters specified by object according to the new arguments given.

Value

Another object of class "rmhcontrol".

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

Examples

```
a <- rmhcontrol(expand=1)
update(a, expand=2)
```

---

will.expand *Test Expansion Rule*

Description

Determines whether an expansion rule will actually expand the window or not.

Usage

```
will.expand(x)
```

Arguments

x	Expansion rule. An object of class "rmhexpand".

Details

An object of class "rmhexpand" describes a rule for expanding a simulation window. See rmhexpand for details.

One possible expansion rule is to do nothing, i.e. not to expand the window.

This command inspects the expansion rule x and determines whether it will or will not actually expand the window. It returns TRUE if the window will be expanded.

Value

Logical value.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>

and Rolf Turner <rolfturner@posteo.net>

See Also

rmhexpand, expand.owin

Examples

```
x <- rmhexpand(distance=0.2)
y <- rmhexpand(area=1)
will.expand(x)
will.expand(y)
```

Window.rmhmodel *Extract Window of Spatial Object*

Description

Given a spatial object (such as a point pattern or pixel image) in two dimensions, these functions extract the window in which the object is defined.

Usage

```
 ## S3 method for class 'rmhmodel'
Window(X, ...)
```

Arguments

X	A spatial object.
...	Ignored.

Details

These are methods for the generic function Window which extract the spatial window in which the object X is defined.

Value

An object of class "owin" (see owin.object) specifying an observation window.

Author(s)

Adrian Baddeley <Adrian.Baddeley@curtin.edu.au>, Rolf Turner <rolfturner@posteo.net> and Ege Rubak <rubak@math.aau.dk>.

See Also

Window, Window.ppp, Window.psp.

owin.object

Examples

```
A <- rmhmodel(cif='poisson', par=list(beta=10), w=square(2))
Window(A)
```

Index

∗ **Diffusion**
rdiffuse, 46
∗ **Tessellation**
rpoisDirichletTess, 121
rpoislinetess, 124
∗ **Three-dimensional**
rpoint3, 120
rpoispp3, 127
runifpoint3, 164
∗ **datagen**
default.expand, 17
default.rmhcontrol, 19
quadratresample, 29
rags, 30
ragsAreaInter, 31
ragsMultiHard, 33
rCauchy, 34
rcell, 38
rcellnumber, 40
rclusterBKBC, 41
rDGS, 44
rdiffuse, 46
rDiggleGratton, 47
rGaussPoisson, 52
rGRFgauss, 53
rHardcore, 55
rjitter.psp, 57
rlabel, 59
rLGCP, 61
rMatClust, 64
rMaternI, 68
rMaternII, 69
rmh, 71
rmh.default, 72
rmhcontrol, 83
rmhexpand, 87
rmhmodel, 89
rmhmodel.default, 90
rmhmodel.list, 97
rmhstart, 100
rMosaicField, 101
rMosaicSet, 103
rmpoint, 104
rmpoispp, 108
rNeymanScott, 111
rnoise, 114
rPenttinen, 116
rpoint, 117
rpoint3, 120
rpoisDirichletTess, 121
rpoisline, 123
rpoislinetess, 124
rpoispp, 125
rpoispp3, 127
rpoisppOnLines, 129
rpoisppx, 131
rPoissonCluster, 132
rpoistrunc, 134
rPSNCP, 136
rshift, 138
rshift.ppp, 139
rshift.psp, 142
rshift.splitppp, 144
rSSI, 145
rstrat, 147
rStrauss, 149
rStraussHard, 151
rtemper, 152
rthin, 154
rthinclumps, 155
rThomas, 157
runifdisc, 161
runifpoint, 162
runifpoint3, 164
runifpointOnLines, 165
runifpointx, 167
rUnround, 168
rVarGamma, 169

∗ **distribution**
dmixpois, 20
dpakes, 23
recipEnzpois, 51
rknn, 58
∗ **manip**
as.owin.rmhmodel, 10
domain.rmhmodel, 21
expand.owin, 24
rthin, 154
rthinclumps, 155
rUnround, 168
will.expand, 174
Window.rmhmodel, 175
∗ **math**
gauss.hermite, 25
recipEnzpois, 51
∗ **methods**
update.rmhcontrol, 173
∗ **models**
clusterprocess, 15
is.stationary, 26
methods.clusterprocess, 27
reach, 49
update.rmhcontrol, 173
∗ **package**
spatstat.random-package, 4
∗ **spatial**
as.owin.rmhmodel, 10
clusterfield, 12
clusterkernel, 14
clusterprocess, 15
clusterradius, 16
default.expand, 17
default.rmhcontrol, 19
domain.rmhmodel, 21
expand.owin, 24
is.stationary, 26
methods.clusterprocess, 27
quadratresample, 29
rags, 30
ragsAreaInter, 31
ragsMultiHard, 33
rCauchy, 34
rcell, 38
rclusterBKBC, 41
rDGS, 44
rDiggleGratton, 47
reach, 49
rGaussPoisson, 52
rGRFgauss, 53
rHardcore, 55
rjitter.psp, 57
rknn, 58
rlabel, 59
rLGCP, 61
rMatClust, 64
rMaternI, 68
rMaternII, 69
rmh, 71
rmh.default, 72
rmhcontrol, 83
rmhexpand, 87
rmhmodel, 89
rmhmodel.default, 90
rmhmodel.list, 97
rmhstart, 100
rMosaicField, 101
rMosaicSet, 103
rmpoint, 104
rmpoispp, 108
rNeymanScott, 111
rnoise, 114
rPenttinen, 116
rpoint, 117
rpoint3, 120
rpoisDirichletTess, 121
rpoisline, 123
rpoislinetess, 124
rpoispp, 125
rpoispp3, 127
rpoisppOnLines, 129
rpoisppx, 131
rPoissonCluster, 132
rPSNCP, 136
rshift, 138
rshift.ppp, 139
rshift.psp, 142
rshift.splitppp, 144
rSSI, 145
rstrat, 147
rStrauss, 149
rStraussHard, 151
rtemper, 152
rthin, 154
rthinclumps, 155

rThomas, 157
runifdisc, 161
runifpoint, 162
runifpoint3, 164
runifpointOnLines, 165
runifpointx, 167
rUnround, 168
rVarGamma, 169
spatstat.random-package, 4
update.rmhcontrol, 173
will.expand, 174
Window.rmhmodel, 175
.Random.seed, *74*, *76*

affine, *24*, *25*, *161*
AreaInter, *32*, *77*, *95*
as.box3, *145*
as.im, *115*
as.mask, *13*, *46*, *54*, *61*, *102*, *115*, *136*, *168*
as.owin, *11*, *12*, *35*, *39*, *52*, *64*, *68*, *70*, *73*, *94*, *109*, *112*, *121*, *123–125*, *132*, *145*, *148*, *157*, *163*, *164*, *169*
as.owin.lpp, *12*
as.owin.ppm, *12*
as.owin.rmhmodel, 10
as.ppp, *73*, *101*
as.rectangle, *112*, *132*
as.tess, *126*

BadGey, *77*, *95*
box3, *128*, *165*
boxx, *131*, *167*

cauchy.estK, *37*
cauchy.estpcf, *37*
clusterfield, 12, *14*, *35*, *65*, *157*, *170*
clusterfield.kppm, *13*
clusterfit, *38*, *68*, *161*, *173*
clusterkernel, 14, *17*
clusterkernel.kppm, *14*
clusterprocess, 15, *28*
clusterradius, 16, *35*, *170*
clusterradius.clusterprocess *(*methods.clusterprocess*)*, 27
clusterstrength, *15*
connected.im, *156*
connected.owin, *156*
cut.ppp, *60*

default.expand, 17, *19*
default.rmhcontrol, 19
density.ppp, *12*, *13*
densityHeat.ppp, *47*
DiggleGatesStibbard, *46*, *77*, *95*
DiggleGratton, *49*, *77*, *90*, *95*, *98*
dilation, *18*
dilation.owin, *88*
disc, *161*, *162*
discretise, *169*
Distributions, *115*
dknn *(*rknn*)*, 58
dmixpois, *8*, 20
domain, *22*
domain.lpp, *22*
domain.ppm, *22*
domain.quadratcount, *22*
domain.quadrattest, *22*
domain.rmhmodel, 21
dpakes, 23
dpois, *21*
dpoisnonzero *(*rpoistrunc*)*, 134
dpoistrunc *(*rpoistrunc*)*, 134

envelope, *18*
erosion, *141*
expand.owin, 24, *89*, *175*

Fiksel, *77*, *95*
Frame, *22*

gauss.hermite, *21*, 25
Geyer, *77*, *95*

Hardcore, *57*, *77*, *95*
Hybrid, *77*, *95*

im.object, *94*, *106*, *109*, *110*, *112*, *119*, *126*, *132*
indefinteg, *42*
intensity.clusterprocess *(*methods.clusterprocess*)*, 27
is.marked, *27*
is.poisson *(*is.stationary*)*, 26
is.stationary, 26

Kest, *40*
Kmodel, *15*
kppm, *13–15*, *17*, *36–38*, *42*, *63*, *67*, *68*, *113*, *160*, *161*, *172*, *173*

layered, *11*
LennardJones, *77*, *93*, *95*
lgcp.estK, *63*

matclust.estK, *67*
matclust.estpcf, *67*
methods.clusterprocess, *15*, 27
MultiHard, *31*, *33*
MultiStrauss, *77*, *90*, *95*, *98*
MultiStraussHard, *77*, *90*, *95*, *98*

OrdThresh, *50*
owin, *12*, *39*, *148*
owin.object, *10*, *12*, *106*, *110*, *119*, *127*, *164*, *175*, *176*

PairPiece, *77*, *90*, *95*, *98*
pcfmodel, *15*
Penttinen, *77*, *90*, *93*, *95*, *117*
persist, *15*
pixellate.ppp, *13*
pknn *(*rknn*)*, 58
pmixpois *(*dmixpois*)*, 20
pointsOnLines, *166*
Poisson, *77*, *95*
pp3, *128*, *165*
ppakes *(*dpakes*)*, 23
ppm, *19*, *50*, *77*, *89*, *90*, *95*, *98*
ppois, *21*
ppoisnonzero *(*rpoistrunc*)*, 134
ppoistrunc *(*rpoistrunc*)*, 134
ppp, *77*, *130*, *166*
ppp.object, *74*, *106*, *110*, *119*, *127*, *164*
ppx, *131*, *167*
predict.clusterprocess *(*methods.clusterprocess*)*, 27
print.clusterprocess *(*methods.clusterprocess*)*, 27
psib, *15*
psp, *123*, *130*, *166*

qknn *(*rknn*)*, 58
qmixpois *(*dmixpois*)*, 20
qpakes *(*dpakes*)*, 23
qpois, *21*
qpoisnonzero *(*rpoistrunc*)*, 134
qpoistrunc *(*rpoistrunc*)*, 134
qqplot.ppm, *18*
quadratcount, *30*
quadratresample, *7–9*, 29
quadrats, *30*
quadscheme, *148*

rags, *7*, 30, *32*, *34*
ragsAreaInter, *7*, *31*, 31, *33*, *34*
ragsMultiHard, *7*, *31*, *32*, 33
rCauchy, *6*, *8*, *13*, *16*, *17*, 34, *42*, *44*, *68*, *114*, *133*, *137*, *161*, *173*
rcell, *6*, *8*, 38, *41*, *127*
rcellnumber, *39*, *40*, 40
rclusterBKBC, 41
rDGS, *6*, *8*, 44, *49*, *57*, *117*, *150*, *152*
rdiffuse, 46
rDiggleGratton, *6*, *8*, *46*, 47, *57*, *117*, *150*, *152*
reach, *17*, *18*, 49
reach.clusterprocess *(*methods.clusterprocess*)*, 27
reach.dppm, *50*
reach.kppm, *50*
reach.ppm, *50*
recipEnzpois, 51, *135*
rescue.rectangle, *126*
rexplode, *7*
rGaussPoisson, *6*, *8*, *38*, 52, *63*, *68*, *114*, *127*, *133*, *161*, *173*
rGRFexpo *(*rGRFgauss*)*, 53
rGRFgauss, 53, *63*
rGRFgencauchy *(*rGRFgauss*)*, 53
rGRFmatern *(*rGRFgauss*)*, 53
rGRFstable *(*rGRFgauss*)*, 53
rHardcore, *6*, *8*, *46*, *49*, 55, *117*, *150*, *152*
rjitter, *7*, *57*, *58*, *168*, *169*
rjitter.psp, 57
rknn, *8*, 58
rlabel, *7*, 59
rLGCP, *8*, *55*, 61
rMatClust, *6*, *8*, *13*, *16*, *17*, *38*, *42*, *44*, *53*, *63*, 64, *69*, *70*, *114*, *127*, *133*, *161*, *173*
rMaternI, *6*, *8*, 68, *70*, *127*, *147*
rMaternII, *6*, *8*, *69*, 69, *127*, *147*
rmh, *6*, *11*, *18*, *45*, *46*, *48*, *49*, *56*, *57*, 71, *73*, *77*, *84–91*, *94*, *95*, *98*, *100*, *101*, *116*, *117*, *150–153*
rmh.default, *71*, *72*, 72, *127*, *153*
rmh.ppm, *71*, *77*
rmhcontrol, *18*, *19*, *73–75*, 83, *89*, *90*, *95*, *98*, *100*, *101*, *173*, *174*

rmhcontrol.default, *19*
rmhexpand, *18*, *24, 25*, *45*, *48*, *56*, *83–86*, 87, *116*, *149*, *151*, *174, 175*
rmhmodel, *50*, *84*, *86*, *89*, 89, *91*, *98*, *101*, *153*
rmhmodel.default, *73*, *89, 90*, 90, *98*
rmhmodel.list, *89, 90*, 97
rmhmodel.ppm, *89, 90*, *98*
rmhstart, *74*, *84*, *86*, *90*, *95*, *98*, 100, *153*
rmixpois *(*dmixpois*)*, 20
rMosaicField, *9*, 101, *103*
rMosaicSet, *8*, *102*, 103
rmpoint, *6*, *8*, 104, *110*, *127*
rmpoispp, *6*, *8*, *33*, *71*, *106*, 108, *126, 127*, *137*
rNeymanScott, *6*, *8*, *17*, *36*, *38*, *42–44*, *53*, *63*, *66*, *68*, 111, *133*, *137*, *159*, *161*, *171*, *173*
rnoise, *7*, 114
rnorm, *115*
rotate, *24, 25*
rpakes *(*dpakes*)*, 23
rPenttinen, *6*, *8*, *46*, *49*, *57*, *93*, 116, *150*, *152*
rpoint, *6*, *8*, *106*, 117, *127*, *146*, *161–164*
rpoint3, 120, *165*
rpois, *21*, *135*
rpoisDirichletTess, *7*, 121, *124*
rpoisline, *7, 8*, 123, *124*
rpoislinetess, *7, 8*, *102, 103*, *122*, 124
rpoisnonzero, *52*
rpoisnonzero *(*rpoistrunc*)*, 134
rpoispp, *6*, *8*, *36–38*, *53*, *63*, *67–71*, *110*, *112*, *114*, 125, *130*, *132, 133*, *147*, *159–161*, *164*, *171–173*
rpoispp3, *7*, 127, *165*
rpoisppOnLines, *7, 8*, 129
rpoisppx, *7*, 131, *167*
rPoissonCluster, *6*, 132
rpoistrunc, 134
rPSNCP, 136
rshift, *7–9*, 138, *142–145*
rshift.ppp, *139*, 139, *142–145*
rshift.psp, *139*, *142*, 142
rshift.splitppp, *139*, 144
rSSI, *6*, *8*, *69, 70*, *127*, 145
rstrat, *6*, *8*, *40*, *127*, 147
rStrauss, *6*, *8*, *46*, *49, 50*, *56, 57*, *77*, *93*, *117*, *127*, 149, *152*
rStraussHard, *6*, *8*, *46*, *49*, *57*, *117*, *150*, 151
rsyst, *40*, *148*
rtemper, 152
rthin, *6–9*, *113*, 154, *156*
rthinclumps, 155
rThomas, *6*, *8*, *13*, *16, 17*, *38*, *42*, *44*, *53*, *68*, *114*, *127*, *133*, *137*, 157, *173*
runif, *115*
runifdisc, *6*, *8*, 161
runifpoint, *6*, *8*, *40*, *119*, *127*, *148*, *161, 162*, 162, *166*
runifpoint3, *7*, *121*, *128*, 164
runifpointOnLines, *7, 8*, *130*, 165
runifpointx, *7*, *131*, 167
rUnround, *47*, 168
rVarGamma, *6*, *8*, *13*, *16, 17*, *38*, *42*, *44*, *68*, *114*, *133*, *137*, *161*, 169

sample, *119*, *163*
set.seed, *76*
shift, *24, 25*
simulate.clusterprocess *(*methods.clusterprocess*)*, 27
simulate.kppm, *37*, *67*, *160*, *172*
simulate.ppm, *18*
Softcore, *77*, *90*, *95*, *98*
spatstat.options, *43*, *84*, *86*, *119*, *163*
spatstat.random *(*spatstat.random-package*)*, 4
spatstat.random-package, 4
split.ppp, *139*, *144*
stepfun, *94, 95*
Strauss, *50*, *77*, *90*, *95*, *98*, *150*
StraussHard, *77*, *90*, *95*, *98*, *151, 152*
summary.ppp, *74*

tess, *164*
thomas.estK, *160*
thomas.estpcf, *160*
Triplets, *77*, *90*, *95*

update, *174*
update.rmhcontrol, *19*, 173

varblock, *30*
varcount, *15*
vargamma.estK, *172, 173*
vargamma.estpcf, *172, 173*

will.expand, *89*, 174
Window, *22*, *175, 176*

Window.ppp, *176*
Window.psp, *176*
Window.rmhmodel, 175

xy.coords, *113*