spatstat Quick Reference 1.4-3

Type `demo(spatstat)` for an overall demonstration.

Creation, manipulation and plotting of point patterns

An object of class `"ppp"` describes a point pattern. If the points have marks, these are included as a component vector `marks`.

To create a point pattern:

`ppp`	create a point pattern from (x, y) and window information `ppp(x, y, xlim, ylim)` for rectangular window `ppp(x, y, poly)` for polygonal window `ppp(x, y, mask)` for binary image window
`as.ppp`	convert other types of data to a `ppp` object
`setmarks`	
`%mark%`	attach/reassign marks to a point pattern

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
`rMaternI`	simulate the Matérn Model I inhibition process
`rMaternII`	simulate the Matérn Model II inhibition process
`rSSI`	simulate Simple Sequential Inhibition process
`rNeymanScott`	simulate a general Neyman-Scott process
`rMatClust`	simulate the Matérn Cluster process
`rThomas`	simulate the Thomas process
`rmh`	simulate Gibbs point process using Metropolis-Hastings

Standard point pattern datasets:

Remember to say `data(bramblecanes)` etc.

`amacrine`	Austin Hughes' rabbit amacrine cells
`bramblecanes`	Bramble Canes data
`cells`	Crick-Ripley biological cells data
`ganglia`	Wässle et al. cat retinal ganglia data
`hamster`	Aherne's hamster tumour data
`lansing`	Lansing Woods data
`longleaf`	Longleaf Pines data
`nztrees`	Mark-Esler-Ripley trees data
`redwood`	Strauss-Ripley redwood saplings data
`redwoodfull`	Strauss redwood saplings data (full set)
`swedishpines`	Strand-Ripley swedish pines data

To manipulate a point pattern:

`plot.ppp`	plot a point pattern `plot(X)`

`"[.ppp"`	extract a subset of a point pattern
	`pp[subset]`
	`pp[, subwindow]`

`superimpose`	superimpose any number of point patterns
`cut.ppp`	discretise the marks in a point pattern
`unmark`	remove marks
`setmarks`	attach marks or reset marks
`rotate`	rotate pattern
`shift`	translate pattern
`affine`	apply affine transformation
`ksmooth.ppp`	kernel smoothing
`identify.ppp`	interactively identify points

See `spatstat.options` to control plotting behaviour.

To create a window:

An object of class `"owin"` describes a spatial region (a window of observation).

`owin`	Create a window object `owin(xlim, ylim)` for rectangular window `owin(poly)` for polygonal window `owin(mask)` for binary image window
`as.owin`	Convert other data to a window object
`ripras`	Ripley-Rasson estimator of window, given only the points
`letterR`	polygonal window in the shape of the R logo

To manipulate a window:

`plot.owin`	plot a window. `plot(W)`
`bounding.box`	Find a tight bounding box for the window
`erode.owin`	erode window by a distance r
`complement.owin`	invert (inside $\leftrightarrow$ outside)
`rotate`	rotate window
`shift`	translate window
`affine`	apply affine transformation

Digital approximations:

`as.mask`	Make a discrete pixel approximation of a given window
`nearest.raster.point`	map continuous coordinates to raster locations
`raster.x`	raster x coordinates
`raster.y`	raster y coordinates

See `spatstat.options` to control the approximation

Geometrical computations with windows:

`inside.owin`	determine whether a point is inside a window
`area.owin`	compute window's area
`diameter`	compute window frame's diameter
`eroded.areas`	compute areas of eroded windows
`bdist.points`	compute distances from data points to window boundary
`bdist.pixels`	compute distances from all pixels to window boundary
`centroid.owin`	compute centroid (centre of mass) of window
`is.subset.owin`	determine whether one window contains another
`trim.owin`	intersect a window with a rectangle

Pixel images

An object of class `"im"` represents a pixel image. Such objects are returned by some of the functions in **spatstat** including `Kmeasure`, `setcov` and `ksmooth.ppp`.

`im`	create a pixel image
`as.im`	convert other data to a pixel image
`plot.im`	plot a pixel image on screen as a digital image
`contour.im`	draw contours of a pixel image
`persp.im`	draw perspective plot of a pixel image
`[.im`	extract subset of pixel image
`shift.im`	apply vector shift to pixel image
`X`	print very basic information about image `X`
`summary(X)`	summary of image `X`
`is.im`	test whether an object is a pixel image

Exploratory Data Analysis

Inspection of data

`summary(X)`	print useful summary of point pattern `X`
`X`	print basic description of point pattern `X`

Summary statistics for a point pattern:

`Fest`	empty space function F
`Gest`	nearest neighbour distribution function G
`Kest`	Ripley's K-function
`Jest`	J-function $J = (1 - G)/(1 - F)$
`allstats`	all four functions F, G, J, K
`pcf`	pair correlation function
`Kinhom`	K for inhomogeneous point patterns
`Kest.fft`	fast K-function using FFT for large datasets
`Kmeasure`	reduced second moment measure

Summary statistics for a multitype point pattern:

A multitype point pattern is represented by an object `X` of class "ppp" with a component `X$marks` which is a factor.

`Gcross,Gdot,Gmulti`	multitype nearest neighbour distributions $G_{ij}, G_{i\bullet}$
`Kcross,Kdot, Kmulti`	multitype K-functions $K_{ij}, K_{i\bullet}$
`Jcross,Jdot,Jmulti`	multitype J-functions $J_{ij}, J_{i\bullet}$
`alltypes`	estimates of the above for all i, j pairs

Summary statistics for a marked point pattern:

A marked point pattern is represented by an object `X` of class "ppp" with a component `X$marks`.

`markcorr`	mark correlation function
`Gmulti`	multitype nearest neighbour distribution
`Kmulti`	multitype K-function
`Jmulti`	multitype J-function

Alternatively use `cut.ppp` to convert a marked point pattern to a multitype point pattern.

Programming tools

`applynbd`	apply function to every neighbourhood in a point pattern

Model Fitting

To fit a point process model:

Model fitting in `spatstat` version 1.4 is performed by the function `mpl`. Its result is an object of class `ppm`.

`mpl`	Fit a point process model to a two-dimensional point pattern

Manipulating the fitted model:

`plot.ppm`	Plot the fitted model
`predict.ppm`	Compute the spatial trend and conditional intensity of the fitted point process model
`coef.ppm`	Extract the fitted model coefficients
`fitted.ppm`	Compute fitted conditional intensity at quadrature points
`update.ppm`	Update the fit
`rmh.ppm`	Simulate from fitted model
`print.ppm`	Print basic information about a fitted model
`summary.ppm`	Summarise a fitted model

See `spatstat.options` to control plotting of fitted model.

To specify a point process model:

The first order "trend" of the model is written as an `S` language formula.

`~1`	No trend (stationary)
`~x`	First order term $\lambda(x, y) = \exp(\alpha + \beta x)$ where x, y are Cartesian coordinates
`~polynom(x,y,3)`	Log-cubic polynomial trend

The higher order ("interaction") components are described by an object of class `interact`. Such objects are created by:

`Poisson()`	the Poisson point process
`Strauss()`	the Strauss process
`StraussHard()`	the Strauss/hard core point process
`Softcore()`	pairwise interaction, soft core potential
`PairPiece()`	pairwise interaction, piecewise constant
`DiggleGratton()`	Diggle-Gratton potential
`LennardJones()`	Lennard-Jones potential
`Pairwise()`	pairwise interaction, user-supplied potential
`Geyer()`	Geyer's saturation process
`Saturated()`	Saturated pair model, user-supplied potential
`OrdThresh()`	Ord process, threshold potential
`Ord()`	Ord model, user-supplied potential
`MultiStrauss()`	multitype Strauss process
`MultiStraussHard()`	multitype Strauss/hard core process

Finer control over model fitting:

A quadrature scheme is represented by an object of class "`quad`".

`quadscheme`	generate a Berman-Turner quadrature scheme for use by `mpl`
`default.dummy`	default pattern of dummy points
`gridcentres`	dummy points in a rectangular grid
`stratrand`	stratified random dummy pattern
`spokes`	radial pattern of dummy points
`corners`	dummy points at corners of the window
`gridweights`	quadrature weights by the grid-counting rule
`dirichlet.weights`	quadrature weights are Dirichlet tile areas
`print(Q)`	print basic information about quadrature scheme `Q`
`summary(Q)`	summary of quadrature scheme `Q`