

# spatstat Quick Reference 1.6-1

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:

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

### To simulate a random point pattern:

<code>runifpoint</code>	generate $n$ independent uniform random points
<code>rpoint</code>	generate $n$ independent random points
<code>rmpoint</code>	generate $n$ independent multitype random points
<code>rpoispp</code>	simulate the (in)homogeneous Poisson point process
<code>rmpoispp</code>	simulate the (in)homogeneous multitype Poisson point process
<code>rMaternI</code>	simulate the Matérn Model I inhibition process
<code>rMaternII</code>	simulate the Matérn Model II inhibition process
<code>rSSI</code>	simulate Simple Sequential Inhibition process
<code>rNeymanScott</code>	simulate a general Neyman-Scott process
<code>rMatClust</code>	simulate the Matérn Cluster process
<code>rThomas</code>	simulate the Thomas process
<code>rmh</code>	simulate Gibbs point process using Metropolis-Hastings

### Standard point pattern datasets:

Remember to say `data(bramblecanes)` etc.

<code>amacrine</code>	Austin Hughes' rabbit amacrine cells
<code>betacells</code>	Wässle et al. cat retinal ganglia data
<code>bramblecanes</code>	Bramble Canes data
<code>cells</code>	Crick-Ripley biological cells data
<code>copper</code>	Berman-Huntington copper deposits data
<code>demopat</code>	Synthetic point pattern
<code>finpines</code>	Finnish Pines data
<code>hamster</code>	Aherne's hamster tumour data
<code>japanesepines</code>	Japanese Pines data
<code>lansing</code>	Lansing Woods data
<code>longleaf</code>	Longleaf Pines data
<code>nztrees</code>	Mark-Esler-Ripley trees data
<code>redwood</code>	Strauss-Ripley redwood saplings data
<code>redwoodfull</code>	Strauss redwood saplings data (full set)
<code>simdat</code>	Simulated point pattern (inhomogeneous, with interaction)
<code>spruces</code>	Spruce trees in Saxonia
<code>swedishpines</code>	Strand-Ripley swedish pines data

### To manipulate a point pattern:

<code>plot.ppp</code>	plot a point pattern
	<code>plot(X)</code>
<code>"[.ppp"</code>	extract or replace a subset of a point pattern
	<code>pp[subset]</code>
	<code>pp[, subwindow]</code>
<code>superimpose</code>	superimpose any number of point patterns
<code>cut.ppp</code>	discretise the marks in a point pattern
<code>unmark</code>	remove marks
<code>setmarks</code>	attach marks or reset marks
<code>split.ppp</code>	divide pattern into sub-patterns
<code>rotate</code>	rotate pattern
<code>shift</code>	translate pattern
<code>affine</code>	apply affine transformation
<code>ksmooth.ppp</code>	kernel smoothing
<code>identify.ppp</code>	interactively identify points
See <code>spatstat.options</code> to control plotting behaviour.	

### To create a window:

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

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

### To manipulate a window:

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

### Digital approximations:

<code>as.mask</code>	Make a discrete pixel approximation of a given window
<code>nearest.raster.point</code>	map continuous coordinates to raster locations
<code>raster.x</code>	raster x coordinates
<code>raster.y</code>	raster y coordinates
See <code>spatstat.options</code> to control the approximation	

### Geometrical computations with windows:

<code>intersect.owin</code>	intersection of two windows
<code>union.owin</code>	union of two windows
<code>inside.owin</code>	determine whether a point is inside a window
<code>area.owin</code>	compute window's area
<code>diameter</code>	compute window frame's diameter
<code>eroded.areas</code>	compute areas of eroded windows
<code>bdist.points</code>	compute distances from data points to window boundary
<code>bdist.pixels</code>	compute distances from all pixels to window boundary
<code>distmap.owin</code>	distance transform image
<code>centroid.owin</code>	compute centroid (centre of mass) of window
<code>is.subset.owin</code>	determine whether one window contains another
<code>trim.owin</code>	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`.

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

## Quadrat methods

`quadratcount`    Quadrat counts

## 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  
`envelope`        simulation envelopes for  $K, G$  etc  
`Kinhom`          $K$  for inhomogeneous point patterns  
`Kest.fft`        fast  $K$ -function using FFT for large datasets  
`Kmeasure`        reduced second moment measure  
`nddist`          nearest neighbour distances  
`pairstat`        distances between all pairs of points  
`crossdist`       distances between points in two patterns  
`exactdt`        distance from any location to nearest data point  
`distmap`        distance map image

## 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 1.6 is performed by the function **ppm**. Its result is an object of class **ppm**.

**ppm** 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  
**anova.ppm** Analysis of deviance  
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  
**~harmonic(x,y,2)** Log-harmonic 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".

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

## Simulation and goodness-of-fit

<code>rmh.ppm</code>	simulate realisations of a fitted model
<code>envelope</code>	compute simulation envelopes for a fitted model