Introduction to spatstat

Adrian Baddeley and Rolf Turner

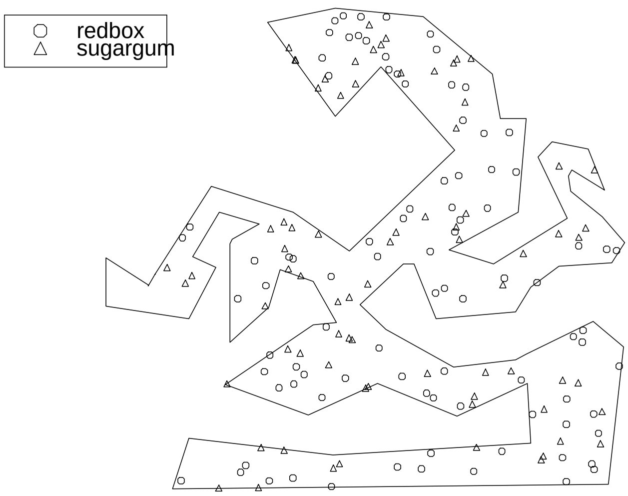
spatstat version 1.8-6

Abstract

spatstat is a library in S-PLUS/R for the statistical analysis of point pattern data. This document is a brief introduction to the package for users.

1 Introduction

Spatstat is a contributed library in S-PLUS and R for the statistical analysis of spatial data. Version 1.x of the library deals mainly with patterns of points in the plane. [1] The points may carry 'marks', and the spatial region in which the points were recorded may have arbitrary shape. Here is an example:



The package supports

- creation, manipulation and plotting of point patterns
- exploratory data analysis
- simulation of point process models
- parametric model-fitting
- residuals and diagnostic plots

[1] Line segment patterns are also supported.

The point process models to be fitted may be quite general Gibbs/Markov models; they may include spatial trend, dependence on covariates, and interpoint interactions of any order (i.e. not restricted to pairwise interactions). Models are specified by a `formula` in the S language, and are fitted using a single function `ppm` analogous to `glm` and `gam`.

This document is an introduction to the main features of `spatstat` and its use. Please see the "`spatstat` Quick Reference" page for an annotated list of all functions in the library. See the online help or printed manual for detailed information about each function.

Demonstration

You may like to try the following quick demonstration of the package. A more extensive demonstration can be seen by typing `demo(spatstat)` and `demo(diagnose)`.

`library(spatstat)`	Attach `spatstat` library
`data(swedishpines)`	Find "Swedish Pines" dataset
`X <- swedishpines`	Rename it
`plot(X)`	Plot it
`summary(X)`	Print a useful summary of it
`K <- Kest(X)`	Estimate its K function
`plot(K)`	Plot the estimated K function
`plot(allstats(X))`	Plot the F, G, J and K functions
`fit <- ppm(X, ~1, Strauss(r=7))`	Fit a Strauss process model
`fit`	Describe the fitted model
`Xsim <- rmh(model=fit, start=list(n.start=X$n), control=list(nrep=1e4))`	Simulate from fitted model
`plot(Xsim)`	Plot simulated pattern
`data(demopat)`	Artificial data in irregular window, with 2 types of points
`plot(demopat, box=FALSE)`	Plot the pattern
`plot(alltypes(demopat, "K"))`	Plot array of cross-type K functions
`pfit <- ppm(demopat, ~marks + polynom(x,2), Poisson())`	Fit inhomogeneous multitype Poisson point process model
`plot(pfit)`	Plot the fitted intensity surface

2 Data

2.1 Overview

The main types of data supported in `spatstat` are *point patterns*, *windows*, *pixel images* and *line segment patterns.*

Point patterns

A point pattern is represented in `spatstat` by an object of class `"ppp"`. This makes it easy to plot a point pattern, manipulate it and subject it to analysis.

A dataset in this format contains the x, y coordinates of the points, optional 'mark' values attached to the points, and a description of the spatial region or 'window' in which the pattern was observed. See `help(ppp.object)` for further details.

To obtain a `"ppp"` object you can

- use one of the datasets supplied with the package;
- create one from data in R, using `ppp()`;
- create one from data in a text file, using `scanpp()`;
- convert data from other R libraries, using `as.ppp()`;
- generate a random pattern using one of the simulation routines.

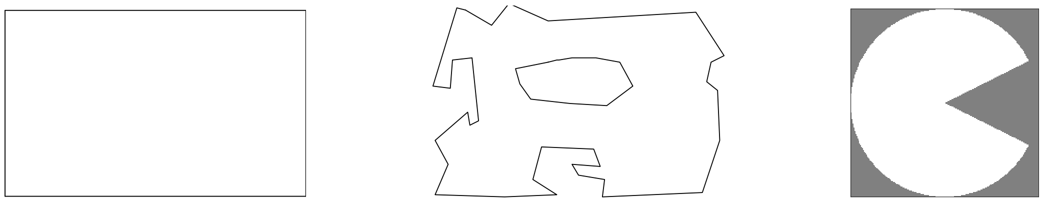
These possibilities are elaborated below.

Spatial windows

Note that, when you create a new point pattern object, you need to specify the spatial region or window in which the pattern was observed. There is intentionally no automatic "guessing" of the window dimensions from the points alone. [2]

The window may have arbitrary shape; it may be a rectangle, a polygon, a collection of polygons (including holes), or a binary image.

[2] However, the function `ripras` will compute an estimate of the window given only the coordinates of the points.



If the observation window needs to be stored as a dataset in its own right, it is represented in `spatstat` by an object of class `"owin"`. See `help(owin.object)` for further details. Objects of this class can be plotted and manipulated in a few simple ways. They can be created using the function `owin()`.

The simplest way to create a point pattern with a non-rectangular window is to use the functions `ppp()` and/or `owin()`.

Marks

Each point in a spatial point pattern may carry additional information called a 'mark'. For example, points which are classified into two or more different types (on/off, case/control, species, colour, etc) may be regarded as marked points, with a mark which identifies which type they are. Data recording the locations and heights of trees in a forest can be regarded as a marked point pattern where the mark attached to a tree's location is the tree height.

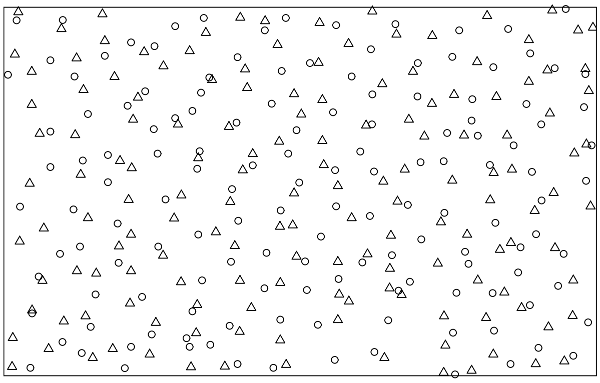
The current version of `spatstat` supports marked point patterns of two kinds:

continuous marks : the mark attached to each point is a single real number (e.g. tree height);

multitype pattern : points are classified into several types; the mark attached to each point is a level of a factor (e.g. tree species).

The mark values must be given in a vector `marks` of the same length as the coordinate vectors `x` and `y`. This is interpreted so that `marks[i]` is the mark attached to the point `(x[i],y[i])`.

Note: To distinguish between the cases of continuous marks and multitype points, `spatstat` requires that for a multitype point pattern, `marks` must be a `factor`.



2.2 Standard point pattern datasets

Some standard point pattern datasets are supplied with the package. They include:

`amacrine`	Austin Hughes' rabbit amacrine cells	multitype
`ants`	Harkness-Isham ant nests data	irregular window, multitype
`betacells`	Wässle et al. cat retinal ganglia data	multitype
`bramblecanes`	Bramble Canes data	multitype
`cells`	Crick-Ripley biological cells data	
`chorley`	Chorley-South Ribble cancer data	irregular window, multitype
`copper`	Copper deposits data	spatial covariates
`demopat`	artificial data	irregular window, multitype
`finpines`	Finnish Pines data	continuous marks
`hamster`	Aherne's hamster tumour data	multitype
`humberside`	childhood leukaemia data	irregular window, multitype
`japanesepines`	Japanese Pines data	
`lansing`	Lansing Woods data	multitype
`longleaf`	Longleaf Pines data	continuous marks
`nztrees`	Mark-Esler-Ripley trees data	
`redwood`	Strauss-Ripley redwood saplings data	
`redwoodfull`	Strauss redwood saplings data (full set)	
`residualspaper`	Data from a journal article	
`simdat`	Simulated point pattern	
`spruces`	Spruce trees in Saxonia	continuous marks
`swedishpines`	Strand-Ripley Swedish pines data	

See the Demonstration in the Introduction for an example of how to use these datasets.

2.3 Creating point patterns using `ppp()`

The function `ppp()` will create a point pattern (an object of class `"ppp"`) from data in R.

Point pattern in rectangular window

Suppose the x, y coordinates of the points of the pattern are contained in vectors `x` and `y` of equal length. If the window of observation is a rectangle, then

```
ppp(x, y, xrange, yrange)
```

will create the point pattern. Here `xrange, yrange` must be vectors of length 2 giving the x and y dimensions, respectively, of the rectangle. For example `ppp(x, y, c(0,1), c(0,1))` would give you a point pattern in the unit square; this is the default so you could also just type `ppp(x, y)`.

To create a marked point pattern, use the additional argument `marks`:

```
ppp(x, y, xrange, yrange, marks=m)
```

where `m` is a vector of the same length as `x` and `y`. Remember that if you intend to create a multitype pattern (where the points are classified into a finite number of possible types) then `m` must be a `factor` (use `factor` or `as.factor` to make it one).

Note that we have to use the "name=value" syntax to specify the marks argument. For example

```
ppp(runif(100),runif(100), marks=factor(sample(1:2,100,replace=TRUE)))
```

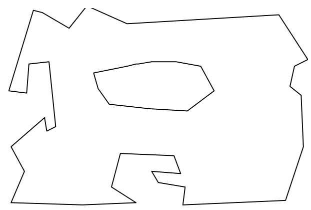
would make a multitype point pattern of 100 random points, uniformly distributed in the unit square, with random types 1 and 2.

Point pattern in polygonal window

Spatstat supports polygonal windows of arbitrary shape and topology. That is, the boundary of the window may consist of one or more closed polygonal curves, which do not intersect themselves or each other. The window may have 'holes'. Type

```
ppp(x, y, poly=p)
```

to create a point pattern with a polygonal window. Again, x and y are the vectors of coordinates of the points. The argument poly=p indicates that the window is polygonal and its boundary is given by the dataset p. Note we must use the "name=value" syntax to give the argument poly.



If the window boundary is a single polygon, then p should be a list with components x and y giving the coordinates of the vertices of the window boundary, **traversed anticlockwise**. For example,

```
ppp(x, y, poly=list(x=c(0,1,0), y=c(0,0,1)))
```

will create a point pattern inside the triangle with corners $(0,0)$, $(1,0)$ and $(0,1)$.

Note that polygons should **not** be closed, i.e. the last vertex should **not** equal the first vertex. The same convention is used in the standard plotting function polygon(), so you can check that p is correct by using polygon(p) to display it.

If the window boundary consists of several separate polygons, then p should be a list, each of whose components p[[i]] is a list with components x and y describing one of the polygons. The vertices of each polygon should be traversed **anticlockwise for external boundaries** and **clockwise for internal boundaries (holes)**. For example, in

```
ppp(x, y, poly=list(
                  list(x=c(0,10,0), y=c(0,0,10)),
                  list(x=c(5,5,6,6), y=c(5,6,6,5)))
```

the window is a large triangle with a small square hole. Notice that the first boundary polygon is traversed anticlockwise and the second clockwise because it is a hole.

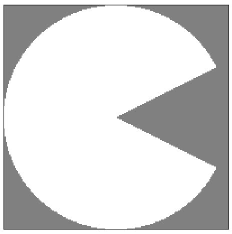
A marked point pattern is created by adding the argument marks exactly as above.

Point pattern in binary mask

The window for the point pattern may be described by a discrete pixel approximation. Type

```
ppp(x, y, mask=m, xrange, yrange)
```

to create the pattern. Here `m` should be a matrix with logical entries; it will be interpreted as a binary pixel image whose entries are `TRUE` where the corresponding pixel belongs to the window.



The rectangle with dimensions `xrange, yrange` is divided into equal rectangular pixels. The correspondence between matrix indices `m[i,j]` and cartesian coordinates is slightly idiosyncratic: the **rows** of `m` correspond to the y coordinate, and the columns to the x coordinate. The entry `m[i,j]` is `TRUE` if the point `(xx[j],yy[i])` (sic) belongs to the window, where `xx, yy` are vectors of pixel coordinates equally spaced over `xrange` and `yrange` respectively.

Image masks can be read from data files, or created by analytic equations. For example to create a point pattern inside the unit disc:

```
w <- owin(c(-1,1), c(-1,1))
w <- as.mask(w)
X <- raster.x(w)
Y <- raster.y(w)
M <- (X^2 + Y^2 <= 1)
pp <- ppp(x, y, c(-1,1), c(-1,1), mask=M)
```

The first line creates a window (an object of class `"owin"`) representing the rectangle $[-1, 1] \times [-1, 1]$. The next line converts this to a binary image mask (a rectangular grid of pixels, with default dimensions 100×100) in which all of the pixel values are `TRUE`. The next two lines create matrices `X, Y` of the same dimensions as the pixel image, which contain respectively the x and y

coordinates of each pixel. The fourth line defines a logical matrix `M` whose entries are `TRUE` where the inequality $x^2+y^2 \leq 1$ holds, in other words, where the centre of the pixel lies inside the unit disc. The last line creates a point pattern with this window.

A marked point pattern is created by adding the argument `marks` exactly as above.

2.4 Point pattern in existing window

You may already have a window `W` (an object of class `"owin"`) ready to hand, and now want to create a pattern of points in this window. This can be done with

```
ppp(x, y, window=W)
```

For example you may want to put a new point pattern inside the window of an existing point pattern `X`; the window is accessed as `X$window`, so type

```
ppp(x, y, window=X$window)
```

To generate random points inside an existing window, it is easiest to use the simulation functions described in section 2.7.

2.5 Scanning point pattern data from text files

The simple function `scanpp()` will read point pattern coordinate data from a text file (in table format) and create a point pattern object from it. See `help(scanpp)` for details.

2.6 Converting other data types

The convenient function `as.ppp()` converts data from other formats into point pattern objects in `spatstat`. It will accept point pattern objects from the Venables-Ripley `spatial` library (class `"spp"`), data frames with appropriate dimensions or column labels, and raw data. See `help(as.ppp)` for details.

2.7 Generating random point patterns

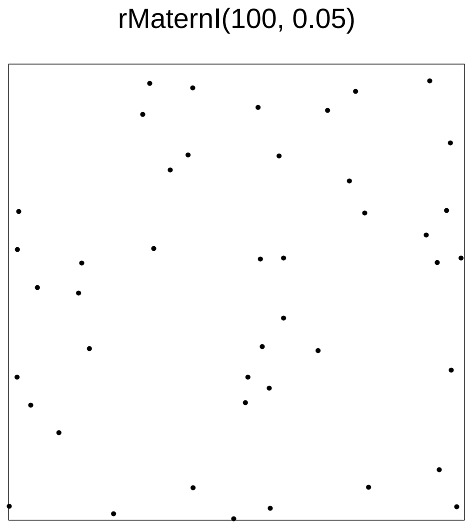
The following functions in `spatstat` generate random patterns of points from various stochastic models. They return a point pattern (as an object of class `"ppp"`).

`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
`rNeymanScott`	simulate a general Neyman-Scott process
`rMatClust`	simulate the Matérn Cluster process
`rThomas`	simulate the Thomas process
`rlabel`	randomly (re)label the points of an existing pattern
`rtoro`	randomly shift the points of an existing pattern
`rmh`	run Metropolis-Hastings algorithm

For example

```
plot(rMaternI(200,0.05))
```

will plot one realisation of the Matérn Model I inhibition process with parameters $\beta = 200$ and $r = 0.05$. See the help entries for these functions for further details.



The function `rmh` is a basic implementation of the Metropolis-Hastings algorithm for simulating point processes. A range of different processes can be simulated. The function `rmh` is generic and has two methods, `rmh.default` which simulates a point process model specified explicitly by a list of its parameters, and `rmh.ppm` which simulates a point process model that has been fitted to data by the fitting function `ppm`.

The implementation of `rmh` in version 1.8 of `spatstat` can currently generate simulated realisations of the Strauss process; Strauss process with a hard core; the Soft Core process; Geyer's saturation process; pairwise interaction processes proposed by Diggle, Gratton and Stibbard (in `rmh.default` only) and by Diggle and Gratton; and multitype versions of the Strauss and Strauss/hard core processes. It can also generate processes with an arbitrary pairwise interaction function given as a vector of values. All these processes may have a spatial trend.

For examples, see `help(rmh.default)` and `help(rmh.ppm)`.

2.8 Pixel images

A 'pixel image' is an array of values attached to a rectangular grid of spatial locations. It may be visualised on the screen as a digital image, a contour map, or a relief surface.

In spatstat a pixel image is represented by an object of class "im". The individual pixel values may be numeric (real or integer), logical, or factor levels.

Functions which create a pixel image are

as.im	convert data to pixel image
im	create a pixel image from raw data

Functions which compute a pixel image from other data include

distmap	distance map
density.ppp	kernel smoothing of point pattern
density.psp	kernel smoothing of line segment pattern
Kmeasure	reduced second moment measure of point pattern
setcov	set covariance function of spatial window

2.9 Line segment patterns

A spatial pattern of line segments is represented by an object of class "psp". Like a point pattern, a line segment pattern is assumed to have been observed inside a spatial window, which must be specified as part of its description.

The functions which create a line segment pattern are

as.psp	convert data to line segment pattern
psp	create a line segment pattern from raw data

3 Manipulating data

3.1 Plotting

To plot the point pattern object X, type

```
plot(X)
```

which invokes plot.ppp(). See help(plot.ppp) for details. Plotting is isometric, i.e. the physical scales of the x and y axes are the same.

To plot just the window of observation of X, just type plot(X$window). This calls plot.owin().

A marked point pattern is represented graphically by using different plotting symbols for the points of each type (if it's a multitype point pattern, where X$marks is a factor) or by drawing circles of different radii proportional to the mark value (if the mark is a continuous variable). If you just want to see the locations of the points without the marks, type plot(unmark(X)).

The colours, plotting characters, line widths and so on can be modified by adding arguments to the `plot` methods. Default plotting behaviour can also be controlled using the function `spatstat.options`. See the help files for `spatstat.options, plot.owin, plot.ppp`.

3.2 Subsets of point patterns

The `spatstat` library supports the extraction and replacement of subsets of a point pattern, using the array indexing operator `"["`.

Extraction means either "*thinning*" (retaining/deleting some points of a point pattern) or "*trimming*" (reducing the window of observation to a smaller subregion and retaining only those points which lie in the subregion).

If `X` is a point pattern object then

```
X[subset, ]
```

will cause the point pattern to be "thinned". The argument `subset` should be a logical vector of length equal to the number of points in `X`. The points `(X$x[i], X$y[i])` for which `subset[i]=FALSE` will be deleted. The result is another point pattern object, with the same window as `X`, but containing a subset of the points of `X`.

The pattern will be "trimmed" if we call

```
X[ , window]
```

where `window` is an object of class `"owin"` specifying the window of observation to which the point pattern `X` will be restricted. Only those points of `X` lying inside the new `window` will be retained.

Subsets of a point pattern can also be replaced by other data. For example

```
X[subset] <- Y
```

will replace the designated subset of `X` by the pattern `Y`.

See `help(subset.ppp)` for full details.

3.3 Other operations on point patterns

Use the function `unmark` to remove marks from a marked point pattern. For example `plot(unmark(X))` will plot just the locations of the points in a marked point pattern `X`.

Use the function `setmarks` or `%mark%` to attach marks to an unmarked point pattern, or to reset the existing values of the marks. For example

```
X <- runifpoint(100) %mark% rexp(100)
```

generates 100 independent uniformly random points in the unit square and then attaches random marks to the points, each mark having a negative exponential distribution. Similarly

```
X <- rpoispp(42)
M <- sample(letters[1:4], X$n, replace=TRUE)
X <- X %mark% factor(M)
```

generates a Poisson point process with intensity 42 in the unit square, then attaches random types `a` to `d` to the points. Notice that the length of `M` depends on the number of points in `X`. Note also the use of `factor` in the last line.

Use the function `cut.ppp` to transform the marks of a point pattern from numerical values into factor levels.

The function `split.ppp` will divide up a point pattern into a list of several point patterns according to the values of a factor. It can be used to convert a multitype point pattern into a list of point patterns, each consisting of the points of a single type. The result of `split` can easily be plotted.

The function `superimpose` will combine several point patterns into a single point pattern. It accepts any number of arguments, which must all be `"ppp"` objects:

```
U <- superimpose(X, Y, Z)
```

The functions `rotate, shift` and `affine` will subject the point pattern to a planar rotation, translation and affine transformation respectively.

The function `ksmooth.ppp` performs kernel smoothing of a point pattern.

The function `identify.ppp`, a method for `identify`, allows the user to examine a point pattern interactively.

The functions `duplicated.ppp` and `unique.ppp` allow the user to determine whether any points in a point pattern are duplicated, and to remove duplicates.

3.4 Manipulating spatial windows

As explained above, a point pattern object contains a description of the spatial region or window in which the pattern was observed. This is an object of class `"owin"`. It is often convenient to create, manipulate and plot these windows in their own right. The following functions are available; see their help files for details.

Creating new windows

`owin`	create a window
`as.owin`	convert other data into a window
`is.owin`	test whether object is a window
`bounding.box`	Find smallest rectangle enclosing the window
`erode.owin`	Erode window by a distance r
`rotate.owin`	Rotate the window
`shift.owin`	Translate the window in the plane
`affine.owin`	Apply an affine transformation
`complement.owin`	Invert (inside ↔ outside)
`is.subset.owin`	Test whether one window contains another
`intersect.owin`	Intersection of two windows
`union.owin`	Union of two windows
`ripras`	Estimate the window, given only the points

Just for fun, we provide the dataset `letterR`, a polygonal window approximating the shape of the R logo.

Digital approximations:

It is possible (and sometimes necessary) to approximate a window using a discrete grid of pixels.

`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

The default accuracy of the approximation can be controlled using `spatstat.options`.

Geometrical computations with windows:

`intersect.owin`	intersection of two windows
`union.owin`	union of two windows
`inside.owin`	determine whether a point is inside a window
`area.owin`	compute window's area
`diameter`	compute window'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
`distmap`	distance map image
`centroid.owin`	compute centroid (centre of mass) of window

4 Exploratory Data Analysis

4.1 Basic inspection

A useful summary of information about a point pattern dataset `X` can be obtained by typing

```
summary(X)
```

which invokes the function `summary.ppp`. It computes the average intensity of points, summarises the marks if `X` is a marked point pattern, and describes the window.

To decide whether the intensity of points is constant over the window, try

```
plot(ksmooth.ppp(X))
```

which computes a kernel-smoothed estimate of the local intensity function and displays it as an image.

For a multitype pattern it is useful to separate out the points of each type, using `split.ppp`. For example

```
plot(split(X))
```

plots the sub-patterns side-by-side.

4.2 Quadrat methods

Quadrat counting is a simple way to inspect point pattern data. The window of observation is divided into a grid of rectangular tiles, and the number of data points in each tile is counted. Quadrat counts can be obtained using the function `quadratcount`.

4.3 Summary statistics

The library will compute estimates of the summary functions

$F(r)$, the empty space function

$G(r)$, the nearest neighbour distance distribution function

$K(r)$, the reduced second moment function ("Ripley's K")

$J(r)$, the function $J = (1 - G)/(1 - F)$

$g(r)$, the pair correlation function $g(r) = [\frac{d}{dr}K(r)]/(2\pi r)$

for a point pattern, and their analogues for marked point patterns.

These estimates can be used for exploratory data analysis and in formal inference about a spatial point pattern. They are well described in the literature, e.g. Ripley (1981), Diggle (1983), Cressie (1991), Stoyan et al (1995). The J-function was introduced by van Lieshout and Baddeley (1996).

The point pattern has to be assumed to be "stationary" (statistically homogeneous under translations) in order that the functions F, G, J, K be well-defined and the corresponding estimators approximately unbiased. (There is an extension of the K function to inhomogeneous patterns; see below).

The empty space function F of a stationary point process X is the cumulative distribution function of the distance from a fixed point in space to the nearest point of X. The nearest neighbour function G is the c.d.f. of the distance from a point *of the pattern* X to the nearest other point of X. The J function is the ratio $J(r) = (1 - G(r))/(1 - F(r))$. The K function is defined so that $\lambda K(r)$ equals the expected number of additional points of X within a distance r of a point of X, where λ is the intensity (expected number of points per unit area).

In exploratory analyses, the estimates of F, G, J and K are useful statistics. F summarises the sizes of gaps in the pattern; G summarises the clustering of close pairs of points; J is a comparison between these two effects; and K is a second order measure of spatial association.

For inferential purposes, the estimates of F, G, J, K are usually compared to their true values for a completely random (Poisson) point process, which are

$$
\begin{aligned}
F(r) &= 1 - \exp(-\lambda \pi r^2) \\
G(r) &= 1 - \exp(-\lambda \pi r^2) \\
J(r) &= 1 \\
K(r) &= \pi r^2
\end{aligned}
$$

where again λ is the intensity. Deviations between the empirical and theoretical curves may suggest spatial clustering or spatial regularity.

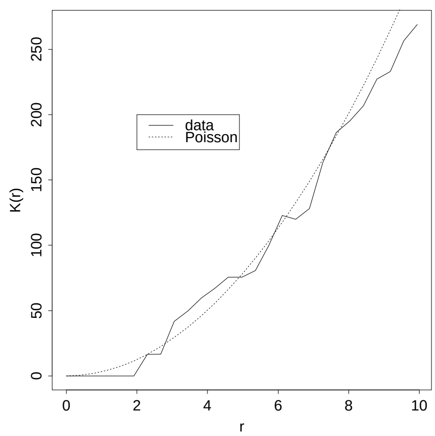
4.4 Implementation in `spatstat`

The corresponding `spatstat` library functions are :

`Fest`	empty space function F
`Gest`	nearest neighbour distribution function G
`Kest`	Ripley's K-function
`Jest`	J-function
`allstats`	all four functions F, G, J, K
`pcf`	pair correlation function g

(Some others are listed below).

The routines `Fest, Gest, Jest, Kest, pcf` each return an object of class `"fv"`. This is a data frame with some extra features making it easier to plot. A column labelled `r` in the data frame contains the values of the argument r for which the summary function ($F(r)$, etc) has been evaluated. Other columns give the estimates of the summary function itself ($F(r)$, etc) by various methods. Another column `theo` contains the theoretical (Poisson) value of the same function.



These columns can be plotted against each other for the purposes of exploratory data analysis. For example

```
G <- Gest(X)
plot(G$r, G$km, type="l")
```

will give you a basic plot of $\widehat{G}(r)$ against r where $\widehat{G}(r)$ is the Kaplan-Meier estimate of $G(r)$ computed by `Gest`. More elegantly

```
G <- Gest(X)
plot(G)
```

will produce a nice default plot of $\widehat{G}(r)$ against r using the plot method (`plot.fv`). This plot method allows you to specify a plot of several curves together using a `formula`:

```
plot(G, cbind(km, rs, theo) ~ r)
```

will plot the Kaplan-Meier estimate `G$km`, the border corrected (reduced sample) estimate `G$rs`, and the theoretical Poisson value, against r. This can be abbreviated using the symbol • to represent all the available estimates:

```
plot(G, . ~ r)
```

If you prefer a P–P style plot, try

```
plot(G, cbind(km, theo) ~ theo)
```

or

```
plot(G, . ~ theo)
```

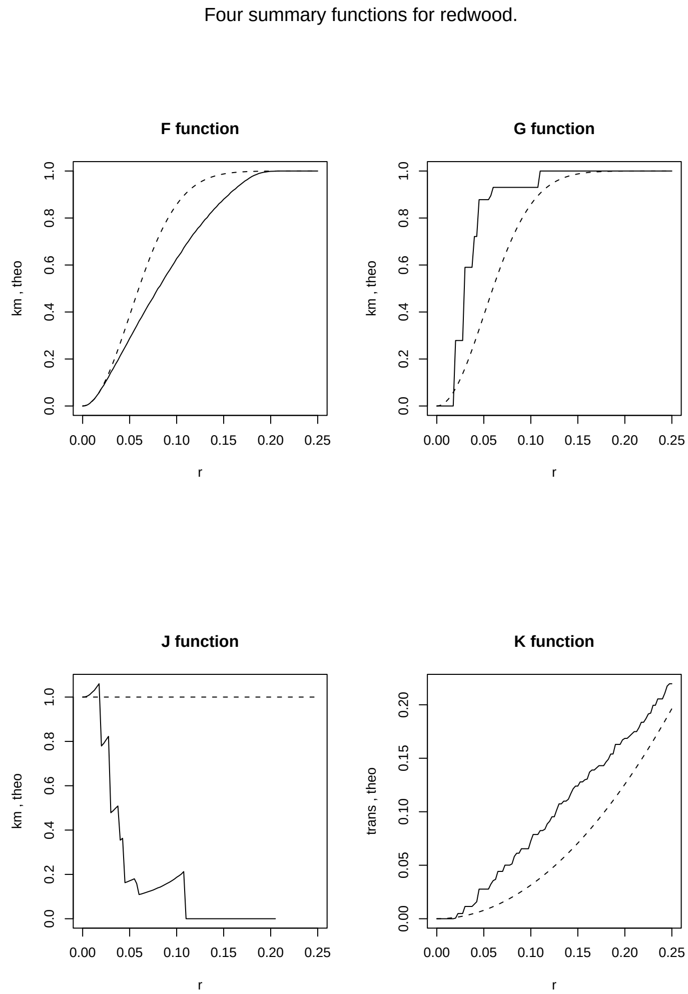
Ripley's K function is often transformed to $L(r) = \sqrt{K(r)/\pi}$. To estimate and plot the L function, type

```
K <- Kest(X)
plot(K, sqrt(./pi) ~ r)
```

For a quick first analysis of a point pattern it is often convenient to hit

```
plot(allstats(X))
```

which plots estimates of the F, G, J and K functions in a single display. See section 4.10 for more information on `allstats`.



There are also several related alternative functions. For the second order statistics, alternatives are:

`Kinhom`	K function for inhomogeneous patterns
`Kest.fft`	fast K-function using FFT for large datasets
`Kmeasure`	reduced second moment measure

See the help files for these functions.

Distances between points are also computed (without edge correction) by:

`nndist`	nearest neighbour distances
`pairdist`	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

4.5 Function envelopes

A popular strategy for assessing whether a point pattern is 'random' (Poisson) or has interpoint interactions, is to plot a summary function for the data (e.g. the K function) together with the upper and lower envelopes of the K functions for 99 simulated realisations of a completely random (uniform Poisson) point process. This can be done with the command `envelope`.

4.6 Comparing and combining summary functions

The command `eval.fv` will evaluate any expression involving summary functions. For example, you can use it to subtract one K function from another:

```
KX <- Kest(X)
KY <- Kest(Y)
Kdiff <- eval.fv(KX - KY)
plot(Kdiff)
```

will evaluate and plot the difference between the K functions of two point patterns `X` and `Y`.

4.7 Pixel images

Functions which display or manipulate a pixel image include

`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
`print.im`	print basic information about pixel image
`summary.im`	summary of pixel image
`is.im`	test whether an object is a pixel image
`compatible.im`	test whether two images have compatible dimensions
`eval.im`	evaluate an expression involving pixel images
`shift.im`	apply vector shift to pixel image

4.8 Line segment patterns

Functions which display or manipulate a line segment pattern include

`plot.psp`	plot a line segment pattern
`[.psp`	extract subset of line segment pattern
`print.psp`	print basic information about line segment pattern
`summary.psp`	print summary of line segment pattern
`endpoints.psp`	extract endpoints of line segments
`midpoints.psp`	compute midpoints of line segments
`lengths.psp`	compute lengths of line segments
`angles.psp`	compute orientation angles of line segments
`distmap.psp`	compute distance map of line segments
`density.psp`	kernel smoothing of line segments
`affine.psp`	affine transformation of line segments
`shift.psp`	vector shift of line segment pattern
`rotate.psp`	rotation of line segment pattern

4.9 Summary statistics for a multitype point pattern:

Analogues of the G, J and K functions have been defined in the literature for "multitype" point patterns, that is, patterns in which each point is classified as belonging to one of a finite number of possible types (e.g. on/off, species, colour). The best known of these is the cross K function $K_{ij}(r)$ derived by counting, for each point of type i, the number of type j points lying closer than r units away.

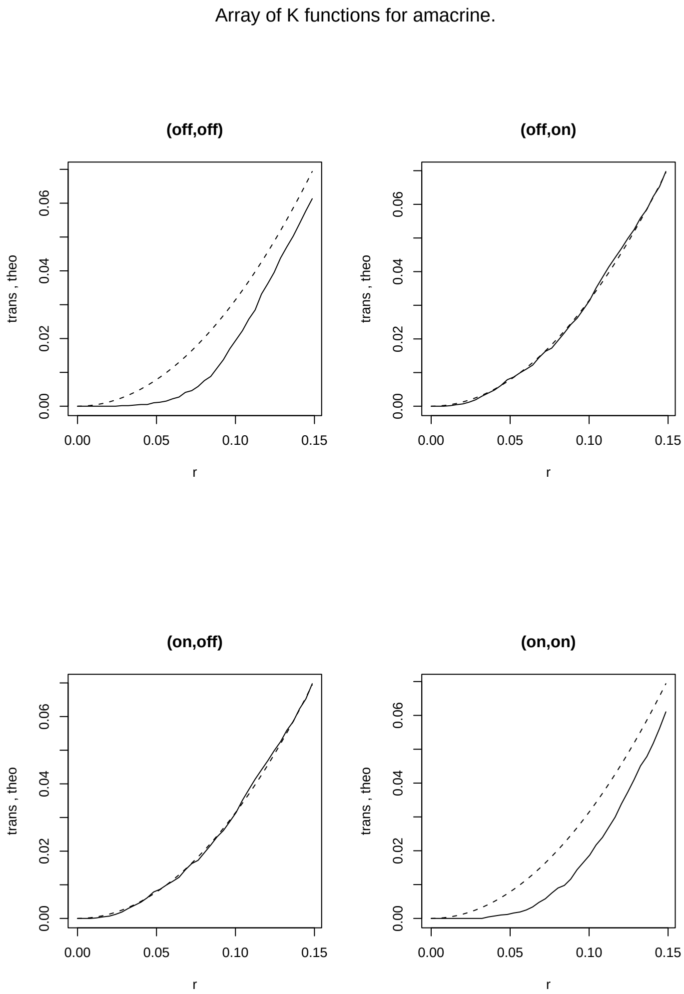
`Gcross,Gdot,Gmulti`	multitype nearest neighbour distributions $G_{ij}, G_{i\bullet}$
`Kcross,Kdot, Kmulti`	multitype K-functions $K_{ij}, K_{i\bullet}$
`Jcross,Jdot,Jmulti`	multitype J-functions J_{ij}, $J_{i\bullet}$
`alltypes`	array of multitype functions
`Iest`	multitype I-function

These functions (with the exception of `alltypes`) operate in a very similar way to `Gest, Jest, Kest` with additional arguments specifying the type(s) of points to be studied.

To compute and plot the cross K function $K_{ij}(r)$ for all possible pairs of types i and j,

```
plot(alltypes(X,"K"))
```

See the next section for further information.



4.10 Function arrays

A function array is a collection of functions $f_{i,j}(r)$ indexed by integers i and j. An example is the set of cross K functions $K_{ij}(r)$ for all possible pairs of types i and j in a multitype point pattern. It is best to think of this as a genuine matrix or array.

A function array is represented in `spatstat` by an object of type `"fasp"`. It can be stored, plotted, indexed and subsetted in a natural way. If `Z` is a function array, then

```
plot(Z)
plot(Z[,3:5])
```

will plot the entire array, and then plot the subarray consisting only of columns 3 to 5. See `help(fasp.object)`, `help(plot.fasp)` and `help("[.fasp")` for details.

The value returned by `alltypes` is a function array. `alltypes` computes a summary statistic for each possible type, or each possible pairs of types, in a multitype point pattern. For example if `X` is a multitype point pattern with 3 possible types,

```
Z <- alltypes(X, "K")
```

yields a 3×3 function array such that (say) `Z[1,2]` represents the cross-type K function $K_{1,2}(r)$ between types 1 and 2. The command `plot(Z)` will plot the entire set of cross K functions as a two-dimensional array of plot panels. Arguments to `plot.fasp` can be used to change the plotting style, the range of the axes, and to select which estimator of K_{ij} is plotted.

The value returned by `allstats` is a 2×2 function array containing the F, G, J and K functions of an (unmarked) point pattern.

4.11 Summary functions for a continuously marked point pattern

Some point patterns are marked, but not multitype. That is, the points may carry marks that do not belong to a finite list of possible types. The marks might be continuous numerical values, complex numbers, etc.

An example in `spatstat` is the dataset `longleaf` where the marks represent tree diameters. You can easily recognise whether a point pattern is multitype or not by the behaviour of the plot function: a multitype pattern is plotted using different plotting symbols for each type, while a marked point pattern with numerical marks is plotted using circles of radius proportional to the marks.

There are a few ways to study such patterns in `spatstat`:

- the function markcorr computes the mark correlation function of an arbitrary marked point pattern. See the help file for markcorr.
- you can convert a marked point pattern to a multitype point pattern using the function cut.ppp, for example, classifying the marks into High, Medium and Low, then apply the abovementioned functions for multitype point patterns. This is usually a good exploratory step.
- the functions Kmulti, Gmulti, Jmulti operate on arbitrary marked point patterns. They require arguments I, J identifying two subsets of the point pattern. These two subsets will be treated as two discrete types.
- ignore the marks (use the function unmark to remove them) and analyse only the locations of the points.

4.12 Programming tools

spatstat also contains some programming tools to help you perform calculations with point patterns. One of these is the function applynbd which can be used to visit each point of the point pattern, identify its neighbouring points, and apply any desired operation to these neighbours.

For example the following code calculates the distance from each point in the pattern redwood to its second nearest neighbour:

```
nnd2 <- applynbd(redwood,
                 N = 2,
                 function(Y, cur, d, r){max(d)},
                 exclude=TRUE)
```

See the help on applynbd for examples and details.

You can also use applynbd to perform animations in which each point of the point pattern is visited and a graphical display is executed. There is an example in demo(spatstat).

5 Model fitting

spatstat enables parametric models of spatial point processes to be fitted to point pattern data. The scope of possible models is very wide. Models may include spatial trend, dependence on covariates, and interpoint interactions of any order (i.e. we are not restricted to pairwise interactions). Models are specified by formulae in the S language and fitted by a function ppm() analogous to glm() and gam().

Models can be fitted either by the method of maximum pseudolikelihood, or by an approximate maximum likelihood method. Maximum pseudolikelihood is very fast (using a computational device developed by Berman & Turner (1992) and Baddeley & Turner (2000)). Approximate maximum likelihood is slower, but has better statistical properties when the model has strong interpoint interactions.

For example if X is a point pattern,

```
ppm(X, ~1, Strauss(r=0.1), ....)
```

fits the stationary Strauss process with interaction radius $r = 0.1$, and

```
ppm(X, ~x, Strauss(r=0.07), ....)
```

fits the non-stationary Strauss process with a loglinear spatial trend of the form $b(x, y) = \exp(a + bx)$.

The value returned by ppm() is a "fitted point process model" of class "ppm". It can be plotted and predicted, in a manner analogous to the plotting and prediction of fitted generalised linear models.

Simulation of the fitted model is also possible using rmh.

5.1 Models

Here is a very brief summary of parametric models for point processes. See Baddeley & Turner (2000), Cox & Isham (1980), and the excellent surveys by Ripley (1988, 1989).

The point pattern dataset **x** is assumed to be a realisation of a random point process X in W. Typically the null model (or the null hypothesis) will be the homogeneous Poisson point process. Other models will be specified by their likelihood with respect to the Poisson process. Thus we assume X has a probability density $f(\mathbf{x};\theta)$ with respect to the distribution of the Poisson process with intensity 1 on W. The distribution is governed by a p-dimensional parameter θ.

We frequently use the *Papangelou conditional intensity* defined, for a location $u \in W$ and a point pattern $\mathbf{x}$, as

$$\lambda_\theta(u, \mathbf{x}) = \frac{f(\mathbf{x} \cup \{u\};\ \theta)}{f(\mathbf{x} \setminus u;\ \theta)}$$

Effectively our technique fits a model to the conditional intensity.

Here are four important examples.

the homogeneous Poisson process with intensity $\lambda > 0$ has conditional intensity

$$\lambda(u, \mathbf{x}) = \lambda$$

the inhomogeneous Poisson process on W with rate or intensity function $\lambda : W \to \mathsf{R}$, has conditional intensity

$$\lambda(u, \mathbf{x}) = \lambda(u).$$

In statistical models, the intensity $\lambda_\theta(u)$ will depend on θ to reflect 'spatial trend' (a change in intensity across the region of observation) or dependence on a covariate.

the Strauss process on W with parameters $\beta > 0$ and $0 \leq \gamma \leq 1$ and interaction radius $r > 0$, has conditional intensity

$$\lambda(u, \mathbf{x}) = \beta \cdot \gamma^{t(u,\mathbf{x})}$$

where $t(u, \mathbf{x})$ is the number of points of $\mathbf{x}$ that lie within a distance r of the location u. If $\gamma < 1$, the term $\gamma^{t(u,\mathbf{x})}$ makes it unlikely that the pattern will contain many points that are close together.

the pairwise interaction process on W with trend or activity function $b_\theta : W \to \mathsf{R}_+$ and interaction function $h_\theta : W \times W \to \mathsf{R}_+$ has conditional intensity

$$\lambda(u, \mathbf{x}) = b_\theta(u) \prod_i h_\theta(u, x_i)$$

The term $b_\theta(u)$ influences the intensity of points, and introduces a spatial trend if $b_\theta(\cdot)$ is not constant. The terms $h_\theta(u, x_i)$ introduce dependence ('interaction') between different points of the process X.

Our technique only estimates parameters θ for which the model is in "canonical exponential family" form,

$$
\begin{aligned}
f(\mathbf{x}; \theta) &= \alpha(\theta) \exp(\theta^{\mathsf{T}} V(\mathbf{x})) \\
\lambda_\theta(u, \mathbf{x}) &= \exp(\theta^{\mathsf{T}} S(u, \mathbf{x}))
\end{aligned}
$$

where $V(\mathbf{x})$ and $S(u, \mathbf{x})$ are statistics, and $\alpha(\theta)$ is the normalising constant.

5.2 Implementation in `spatstat`

The model-fitting function is called `ppm()` and is strongly analogous to `glm()` or `gam()`. It is called in the form

```
ppm(X, formula, interaction, ...)
```

where `X` is the point pattern dataset, `formula` is an S language formula describing the systematic part of the model, and `interact` is an object of class `"interact"` describing the stochastic dependence between points in the pattern.

What this means is that we write the conditional intensity $\lambda_\theta(u, \mathbf{x})$ as a loglinear expression with two components:

$$\lambda(u, \mathbf{x}) = \exp(\theta_1 B(u) + \theta_2 C(u, \mathbf{x}))$$

where $\theta = (\theta_1, \theta_2)$ are parameters to be estimated.

The term $B(u)$ depends only on the spatial location u, so it represents "spatial trend" or spatial covariate effects. It is treated as a "systematic" component of the model, analogous to the systematic part of a generalised linear model, and is described in `spatstat` by an S language formula.

The term $C(u, \mathbf{x})$ represents "stochastic interactions" or dependence between the points of the random point process. It is regarded as a "distributional" component of the model analogous to the distribution family in a generalised linear model. It is described in `spatstat` by an object of class `"interact"` that we create using specialised `spatstat` functions.

For example

```
ppm(X, ~1, Strauss(r=0.1), ....)
```

fits the stationary Strauss process with interaction radius $r = 0.1$. The spatial trend formula `~1` is a constant, meaning the process is stationary. The argument `Strauss(r=0.1)` is an object representing the interpoint interaction structure of the Strauss process with interaction radius $r = 0.1$. Similarly

```
ppm(X, ~x, Strauss(r=0.1), ....)
```

fits the non-stationary Strauss process with a loglinear spatial trend of the form $b(x, y) = \exp(a + bx)$ where a and b are parameters to be fitted, and x, y are the cartesian coordinates.

Spatial trend

The `formula` argument of `ppm()` describes any spatial trend and covariate effects. The default is `~1`, which corresponds to a process without spatial trend or covariate effects. The formula `~x` corresponds to a spatial trend of the form $\lambda(x, y) = \exp(a + bx)$, while `~x + y` corresponds to $\lambda(x, y) = \exp(a + bx + cy)$ where x, y are the Cartesian coordinates. These could be replaced by any `S` language formula (with empty left hand side) in terms of the reserved names `x, y` and `marks`, or in terms of some spatial covariates which you must then supply.

You can easily construct spatial covariates from the Cartesian coordinates. For example

```
ppm(X, ~ ifelse(x > 2, 0, 1), Poisson())
```

fits an inhomogeneous Poisson process with different, constant intensities on each side of the line $x = 2$.

`spatstat` provides a function `polynom` which generates polynomials in 1 or 2 variables. For example

```
~ polynom(x, y, 2)
```

represents a polynomial of order 2 in the Cartesian coordinates x and y. This would give a "log-quadratic" spatial trend. The distinction between `polynom` and `poly` is explained below. Similarly

```
~ harmonic(x, y, 2)
```

represents the most general *harmonic* polynomial of order 2 in x and y.

It is slightly more tricky to include *observed* spatial covariates; see section 5.7.

Interaction terms

The higher order ("interaction") structure can be specified using one of the following functions. They yield an object (of class `"interact"`) describing the interpoint interaction structure of the model.

`Poisson()` .... Poisson process
`Strauss()` .... Strauss process
`StraussHard()` ... Strauss process with a hard core
`Softcore()` .... Pairwise interaction, soft core potential
`PairPiece()` ..... Pairwise interaction, piecewise constant potential
`DiggleGratton()` Diggle-Gratton potential
`LennardJones()` . Lennard-Jones potential
`Geyer()` .... Geyer's saturation process
`OrdThresh()` ..... Ord process with threshold potential

Note that `ppm()` estimates only the "exponential family" type parameters of a point process model. These are parameters θ such that the loglikelihood is linear in θ. Other so-called "irregular" parameters (such as the interaction radius r of the Strauss process) cannot be estimated by this technique, and their values must be specified a priori, as arguments to the interaction function).

For more advanced use, the following functions will accept "user-defined potentials" in the form of an arbitrary `S` language function. They effectively allow arbitrary point process models of these three classes.

`Pairwise()` .... Pairwise interaction, user-supplied potential
`Ord()` .... Ord model, user-supplied potential
`Saturated()`... Saturated pairwise model, user-supplied potential

The brave user may also generate completely new point process models using the foregoing as templates.

5.3 Fitted models

The value returned by `ppm()` is a "fitted point process model" of class `"ppm"`. It can be stored, inspected, plotted and predicted.

```
fit <- ppm(X, ~1, Strauss(r=0.1), ...)
fit
plot(fit)
pf <- predict(fit)
coef(fit)
```

Printing the fitted object `fit` will produce text output describing the fitted model. Plotting the object will display the spatial trend and the conditional intensity, as perspective plots, contour plots and image plots. The `predict` method computes either the spatial trend or the conditional intensity at new locations.

Methods are provided for the following generic operations applied to `"ppm"` objects:

`predict()`	prediction (spatial trend, conditional intensity)
`plot()`	plotting
`coef()`	extraction of fitted coefficients
`fitted()`	fitted conditional intensity or trend at data points
`update()`	update the fit
`summary()`	print extensive summary information
`anova()`	analysis of deviance

A `"ppm"` object contains full information about the data to which the model was fitted. These data can be extracted using the following:

`quad.ppm()`	extract quadrature scheme
`data.ppm()`	extract data point pattern
`dummy.ppm()`	extract dummy points of quadrature scheme

A trap for young players

Note that problems may arise if you use `predict` on a point process model whose systematic component is expressed in terms of one of the functions `poly()`, `bs()`, `lo()`, or `ns()`. For example

```
fit <- ppm(X, ~ poly(x,2), Poisson())
p <- predict(fit)
```

The same problem occurs with `predict` for generalised linear models and generalised additive models. Each of the abovementioned functions returns a data frame, containing variables that are transformations of the variables given as arguments of the function. However the transformations themselves depend on the values of the arguments. For example `poly` performs Gram-Schmidt orthonormalisation. Hence the fitted coefficients contained in the `fit` object are not appropriate when we predict at new locations — **not even for the default call to `predict(fit)` above**.

For this reason we have supplied the function `polynom` which does not perform any data-dependent transformation, and yields a data frame whose columns are just the powers of its arguments. Replacing `poly` by `polynom` in the code above *does* work correctly.

This problem does not affect the function `harmonic`.

5.4 Simulation and goodness-of-fit

The command `rmh.ppm` will generate simulated realisations of a fitted point process model.

The command `envelope` will compute the upper and lower envelopes of a summary statistic (such as the K function) for simulated realisations from a fitted point process model. This can be used as a test of goodness-of-fit for the fitted model.

5.5 Fitting models to multitype point patterns

The function `ppm()` will also fit models to multitype point patterns. A multitype point pattern is a point pattern in which the points are each classified into one of a finite number of possible types (e.g. species, colours, on/off states). In `spatstat` a multitype point pattern is represented by a `"ppp"` object `X` containing a vector `X$marks`, which must be a `factor`.

Interaction component

Naturally an appropriate specification of the interaction for such a model must be available. Apart from the Poisson process, so far interaction functions have been written for the following:

`MultiStrauss()`	multitype Strauss process
`MultiStraussHard()`	multitype Strauss/hard core process

For the multitype Strauss process, a matrix of "interaction radii" must be specified. If there are m distinct levels (possible values) of the marks, we require a matrix `r` in which `r[i,j]` is the interaction radius r_{ij} between types i and j. For the multitype Strauss/hard core model, a matrix of "hardcore radii" must be supplied as well. These matrices will be of dimension $m \times m$ and must be symmetric. See the help files for these functions.

Trend component

The first-order component ("trend") of a multitype point process model may depend on the marks. For example, a stationary multitype Poisson point process could have different (constant) intensities for each possible mark. A general nonstationary process could have a different spatial trend surface for each possible mark.

In order to represent the dependence of the trend on the marks, the trend formula passed to `ppm()` may involve the reserved name `marks`.

The trend formula `~ 1` states that the trend is constant and does not depend on the marks. The formula `~marks` indicates that there is a separate, constant intensity for each possible mark. The correct way to fit the multitype Poisson process is

```
ppm(X, ~ marks, Poisson())
```

Getting more elaborate, the trend formula might involve both the marks and the spatial locations or spatial covariates. For example the trend formula `~marks + polynom(x,y,2)` signifies that the first order trend is a log-quadratic function of the cartesian coordinates, multiplied by a constant factor depending on the mark.

The formulae

```
~ marks * polynom(x,2)
~ marks + marks:polynom(x,2)
```

both specify that, for each mark, the first order trend is a different log-quadratic function of the cartesian coordinates. The second form looks "wrong" since it includes a "`marks` by `polynom`" interaction without having `polynom` in the model, but since `polynom` is a covariate rather than a factor this is is allowed, and makes perfectly good sense. As a result the two foregoing models are in fact equivalent. However, they will give output that is slightly different in appearance. For instance, suppose that there are 3 distinct marks. The first form of the model gives a "baseline" polynomial, say P_0, and two polynomials say P_1 and P_2. Assume that either Helmert or sum contrasts were used, so that the "sum constraints" apply. The trends corresponding to each of the marks would be given by $\exp(C_1 + P_0 + P_1)$, $\exp(C_2 + P_0 + P_2)$, and $\exp(C_3 + P_0 - P_1 - P_2)$ respectively, where C_1, C_2, and C_3 are the appropriate constant terms corresponding to each of the three marks.

The second model simply gives 3 polynomials, say p_1, p_2, and p_3, corresponding to each of the 3 marks. The trends would then be given by $\exp(c_1 + p_1)$, $\exp(c_2 + p_2)$, and $\exp(c_3 + p_3)$.

5.6 Quadrature schemes

The function `ppm` is an implementation of the technique of Baddeley & Turner (2000) which is based on a quadrature device originated by Berman & Turner (1992). Complete control over the quadrature technique is possible.

Indeed the function `ppm()` prefers to be provided with a "quadrature scheme" as its first argument, although it will make do with a point pattern and calculate a default quadrature scheme.

A quadrature scheme is an object of class `"quad"` giving the locations of quadrature points and the weights attached to them. See `help(quad.object)` for more details. The usual way to create a quadrature scheme is to use `quadscheme()`. For example:

```
Q <- quadscheme(simdat,gridcentres(simdat,50,50),
                                nx=40,ny=40)
fit <- ppm(Q, ~polynom(x,y,3),Softcore(0.5),
                          correction="periodic")
```

Following are the most useful functions for manipulating quadrature schemes.

`quadscheme`	generate a Berman-Turner quadrature scheme for use by `ppm`
`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.quad`	print basic information
`summary.quad`	summary of a quadrature scheme

5.7 Observed spatial covariates

If you wish to model the dependence of a point pattern on a spatial covariate, there are several requirements.

- the covariate must be a quantity $Z(u)$ observable (in principle) at each location u in the window (e.g. altitude, soil pH, or distance to another spatial pattern). There may be several such covariates, and they may be continuous valued or factors.
- the values $Z(x_i)$ of Z at each point of the data point pattern must be available.
- the values $Z(u)$ at *some* other points u in the window must be available.

Thus, it is not enough simply to observe the covariate values at the points of the data point pattern. In order to fit a model involving covariates, `ppm` must know the values of these covariates at every *quadrature* point.

The argument `covariates` to the function `ppm()` specifies the values of the spatial covariates. It may be either a data frame or a list of images.

(a) If `covariates` is a data frame, then the ith row of the data frame is expected to contain the covariate values for the ith quadrature point. The column names of the data frame should be the names of the covariates used in the model formula when you call `ppm`.

(b) If `covariates` is a list of images, then each image is assumed to contain the values of a spatial covariate at a fine grid of spatial locations.

The software will look up the values of these images at the quadrature points. The names of the list entries should be the names of the covariates used in the model formula when you call `ppm`.

Covariates in a data frame

Typically you would use the data frame format (a) if the values of the spatial covariates can only be observed at a few locations. You need to force `ppm()` to use these locations to fit the model. To do this, you will need to construct a quadrature scheme based on the spatial locations where the covariate Z has been observed. Then the values of the covariate at these locations are passed to `ppm()` through the argument `data`.

For example, suppose that `X` is the observed point pattern and we are trying to model the effect of soil acidity (pH). Suppose we have measured the values of soil pH at the points x_i of the point pattern, and stored them in a vector `XpH`. Suppose we have measured soil pH at some other locations u in the window, and stored the results in a data frame `U` with columns `x, y, pH`. Then do as follows:

```
Q <- quadscheme(data=X, dummy=list(x=U$x, y=U$y))
df <- data.frame(pH=c(XpH, U$pH))
```

Then the rows of the data frame `df` correspond to the quadrature points in the quadrature scheme `Q`. To fit just the effect of pH, type

```
ppm(Q, ~ pH, Poisson(), covariates=df)
```

where the term `pH` in the formula `~ pH` agrees with the column label `pH` in the argument `covariates = df`. This will fit an inhomogeneous Poisson process with intensity that is a loglinear function of soil pH. You can also try (say)

```
ppm(Q, ~ pH, Strauss(r=1), covariates=df)
ppm(Q, ~ factor(pH > 7), Poisson(), covariates=df)
ppm(Q, ~ polynom(x, 2) * factor(pH > 7), covariates=df)
```

Covariates in a list of images

The alternative format (b), in which `covariates` is a list of images, would typically be used when the covariate values are computed from other data.

For example, suppose we have a spatial epidemiological dataset containing a point pattern `X` of disease cases, and another point pattern `Y` of controls.

We want to model X as a point process with intensity proportional to the local density ρ of the susceptible population. We estimate ρ by taking a kernel-smoothed estimate of the intensity of Y. Thus

```
rho.hat <- ksmooth.ppp(Y, sigma=1.234)
ppm(X, ~offset(log(rho)), covariates=list(rho=rho.hat))
```

The first line computes the values of the kernel-smoothed intensity estimate at a fine grid of pixels, and stores them in the image object `rho.hat`. The second line fits the Poisson process model with intensity

$$\lambda(u) = \mu \, \rho(u)$$

Note that `covariates` must be a list of images, even though there is only one covariate. The variable name `rho` in the model formula must match the name `rho` in the list.

6 Diagnostics for a fitted model

Diagnostic plots are available for checking a fitted point process model.

`diagnose.ppm`	diagnostic plots for spatial trend
`qqplot.ppm`	diagnostic plot for interpoint interaction

For example, suppose we fit a Strauss process model to the Swedish Pines data:

```
data(swedishpines)
fit <- ppm(swedishpines, ~1, Strauss(7), rbord=7)
```

Then the adequacy of the trend in the fitted model can be assessed by calling

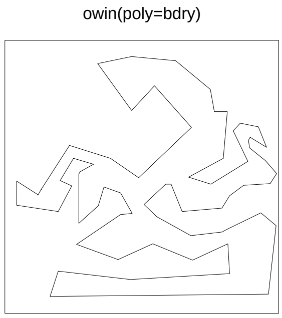
```
diagnose.ppm(fit)
qqplot.ppm(fit)
```

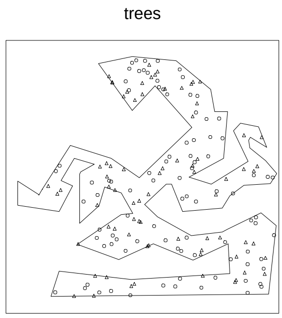
and inspecting these plots. See the help files for further information, or type `demo(diagnose)` for a demonstration of the diagnostics features.

7 Worked example

Suppose we have a data file `trees.tab` containing a table of x,y coordinates and species names for all trees in a paddock. The paddock has an irregular polygonal boundary whose vertex coordinates are stored in the file `paddock`. The following code will read in these data, plot the polygonal boundary, create the point pattern object and plot the point pattern.

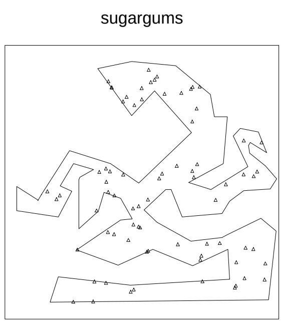
```
tab <- read.table("trees.tab", header=TRUE)
bdry <- scan("paddock", what=list(x=0,y=0))
plot(owin(poly=bdry))
trees <- ppp(tab$x, tab$y, poly=bdry, marks=factor(tab$species))
plot(trees)
```





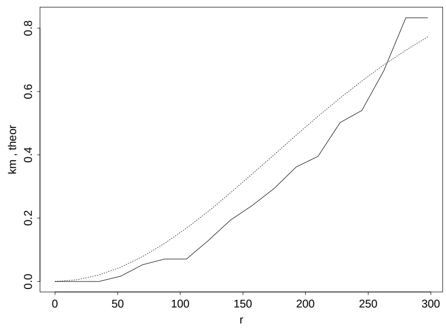
Next we inspect the pattern of sugar gums only, using the subset operation `"["` for point patterns:

```
sugargums <- trees[ trees$marks == "sugargum"]
plot(sugargums)
```



Next we compute and plot the cross-type G function between sugar gums and red box:

```
G <- Gcross(trees, "sugargum", "redbox")
plot(G)
```



Next we fit a nonstationary Poisson process, with a separate log-cubic spatial trend for each species of tree:

```
fitsep <- ppm(trees, ~ marks * polynom(x,y,3), Poisson())
```

We also fit the sub-model in which the species trends are all proportional:

```
fitprop <- ppm(trees, ~ marks + polynom(x,y,3), Poisson())
```

and fit the stationary model in which each species has constant intensity:

```
fit0 <- ppm(trees, ~ marks, Poisson())
```

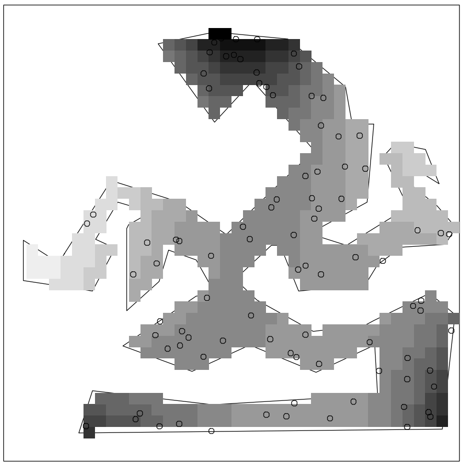
We plot the fitted trend surfaces for each tree:

```
plot(fitsep)
```

Finally we fit a nonstationary multitype Strauss / hard core process with a hard core operating between trees of the same species:

```
mymodel <- MultiStraussHard( levels(trees$marks),
               iradii=matrix(c(150,100,100,200), 2,2),
               hradii=matrix(c(50,NA,NA,50), 2,2))
fit <- ppm(trees, ~ marks * polynom(x,y,3), mymodel, rbord=20)
plot(fit, cif=FALSE)
```

mark = redbox



References

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

[2] A.J. Baddeley and R.D. Gill. Kaplan-Meier estimators for interpoint distance distributions of spatial point processes. *Annals of Statistics* **25** (1997) 263–292.

[3] M. Berman and T.R. Turner. Approximating point process likelihoods with GLIM. *Applied Statistics*, 41:31–38, 1992.

[4] N.A.C. Cressie. *Statistics for spatial data.* John Wiley and Sons, New York, 1991.

[5] P.J. Diggle. *Statistical analysis of spatial point patterns.* Academic Press, London, 1983.

[6] M.B. Hansen, R.D. Gill and A.J. Baddeley. Kaplan-Meier type estimators for linear contact distributions. *Scandinavian Journal of Statistics* **23** (1996) 129–155.

[7] M.B. Hansen, A.J. Baddeley and R.D. Gill. First contact distributions for spatial patterns: regularity and estimation. *Advances in Applied Probability (SGSA)* **31** (1999) 15–33.

[8] M.N.M. van Lieshout and A.J. Baddeley. A non-parametric measure of spatial interaction in point patterns. *Statistica Neerlandica* **50** (1996) 344–361.

[9] M.N.M. van Lieshout and A.J. Baddeley. Indices of dependence between types in multivariate point patterns. *Scandinavian Journal of Statistics* **26** (1999) 511–532.

[10] B.D. Ripley. *Spatial statistics.* John Wiley and Sons, New York, 1981.

[11] D. Stoyan, W.S. Kendall, and J. Mecke. *Stochastic Geometry and its Applications.* John Wiley and Sons, Chichester, second edition, 1995.