point1: point pattern set up and basic analyses

Philip Dixon

3/26/2020

Use of R for spatial point patterns

The main package is spatstat. This provides everything we will need for spatial point patterns. The older package is splancs. A few years ago, most of splancs was integrated into spatstat. (The developers of the two packages are very good friends.) Two years ago, the features in splancs but not in spatstat were for space-time analysis. Spatstat now supports space-time point patterns, but I don't (yet) know what analyses are provided.

spatstat requires its own data formats, because (as you'll see) analysis of a point pattern requires both the locations and the boundary of the study area. It is now possible to convert to and from sp objects (what we've used to store geostatistical and areal data). Those functions are in the maptools library. A spatstat vignette explains how to do this conversion. library(spatstat); vignette('shapefiles') will give you all the details.

spatstat is extremely well documented. There are multiple vignettes. vignette('getstart') is an introduction to the package and its capabilities. There is a huge textbook written by the authors of the package: Baddeley et al, 2016. Spatial Point Patterns: Methodology and Applications with R.

spatstat has many capabilities. We don't discuss all of them (that would take more than a semester by itself). Lecture focuses on the most important and currently most used analyses. Here is my summary of what you need to know to do that subset.

point pattern data

They are the locations (x,y) of events and may include additional information (the marks). spatstat assumes that locations are provided in a rectangular coordinate system, so Euclidean distance is the appropriate distance. Most point pattern data sets are in a relatively small spatial domain (e.g. 1 ha for the cypress data, and I've analyzed some patterns in ca 1000 square miles of central Iowa). Conceptually, great circle distances across large spatial domains are just as useful for point pattern analyses. It's just that spatstat isn't set up to deal with great circle distances.

Take home "watch out": If you're importing locations in longlat coordinates, create an SpatialPoints object with a longlat projection and use spTransform() to convert to UTM first.

All the data I provide will be in a Euclidean coordinate system, usually something local, so (0,0) is one corner of the study area.

spatstat point pattern data sets (.ppp objects)

A point pattern data set in spatstat contains two separate pieces of information: the locations, and the boundary of the sampling area. The boundary is used to adjust for edge effects. The boundary can be an arbitrary polygon, and that can be read in from an ARC shapefile. See vignette('shapefiles') if you need to do this).

Most study area boundaries are rectangles with the edges aligned with the coordinate axes. For example, the swamp tree data set (see next section) has locations within a 50m x 200m rectangle. The corners of the study area are (0,0), (0, 200), (50, 200), (50,0). spatstat provides an especially convenient way to specify such a boundary. This "simplest way" is to list the min and max x coordinate and the min and max y coordinate. So the swamp boundary can be specified as W=c(0,50, 0, 200). If the rectangle was 200m x 50m, so the range of x values is (0, 200), the boundary would be W=c(0,200, 0, 50).

spatstat stores the boundary information internally as an owin object. owin objects are printed in an especially nice way.

There are other ways to specify the boundary. The as.owin() creates the boundary (the Observation WINdow) from several different types of data. See ?as.owin, specifically the Details about the argument W, to see all the options. An alternative is to use the owin() function. See ?owin for information about that.

I will let you know how to set up plot boundaries if something more complicated than c(xmin, xmax, ymin, ymax) is necessary.

The swamp tree data set

The data are in swamp.csv. This contains locations of all tree stems in a single 50m x 200m (= 1ha) plot of the Savannah River Swamp, SC. The four variables are x, y: the location of the stem, live: 1 if live, 0 if standing dead, and sp: the tree species using a two letter code derived (in most cases) from the scientific name. The species codes are:

- TD: bald cypress, Taxodium distichum
- NS: black tupelo, Nyssa sylvatica
- NX: swamp tupelo, Nyssa aquatica
- FX: ash, Fraxinus, multiple species
- OT: other, much less frequent species, including maples You may ask, why isn't Nyssa aquatic labelled NA? I used that code when I first started analyzing these data and wondered why all the swamp tupelo information disappeared. NA is the missing value code! So, NX it is.

Creating a spatstat point pattern (.ppp) data set

as.ppp() creates a .ppp object from the two required pieces of information: locations, specified as a 2 column matrix or data frame, and the boundary. The first column of the matrix (or data frame) is considered the x coordinate; the second the y coordinate. Doesn't matter what variable names those columns have. To see this, look at cypress.pppw below. That flips the plot so it is horizontal, not vertical. My practice is to explicitly subset the desired columns when passing the locations into as.ppp.

We will (for now) only look at the locations of the live cypress trees, so we read the full data set, then extract the live cypresses, then create the ppp object.

```
## Warning: package 'spatstat' was built under R version 3.6.3
```

```
## Loading required package: spatstat.data
```

```
## Warning: package 'spatstat.data' was built under R version 3.6.3
```

```
## Loading required package: nlme
```

```
## Loading required package: rpart
```

```
##
## spatstat 1.63-3       (nickname: 'Wet paint')
## For an introduction to spatstat, type 'beginner'
```

You get a warning when there are locations outside the study area. This is very useful if you mix up your coordinates so the study area boundary coordinates don't correspond to the data coordinates.

```
bad <- as.ppp(cypress[,c('x','y')], W=c(0,200, 0, 50))
```

```
## Warning: 61 points were rejected as lying outside the specified window
```

Printing a ppp object gives you a short summary; summary(ppp object) gives you more information.

```
cypress.ppp
```

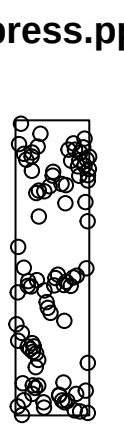
```
## Planar point pattern: 91 points
## window: rectangle = [0, 50] x [0, 200] units
```

```
summary(cypress.ppp)
```

```
## Planar point pattern:  91 points
## Average intensity 0.0091 points per square unit
##
## Coordinates are given to 1 decimal place
## i.e. rounded to the nearest multiple of 0.1 units
##
## Window: rectangle = [0, 50] x [0, 200] units
## Window area = 10000 square units
```

Plotting a ppp object shows you the boundary and locations. The plot is isometrically sized. 10m in the X direction is the same physical distance as 10m in the Y direction. You want plots like this when X and Y are Euclidean coordinates.

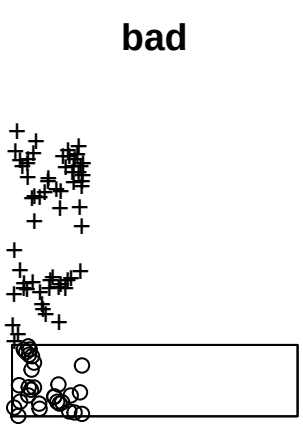
```
plot(cypress.ppp)
```



Plotting something with points outside the boundary shows you those bad points using a different symbol.

```
plot(bad)
```

```
## Warning in plot.ppp(bad): 61 illegal points also plotted
```



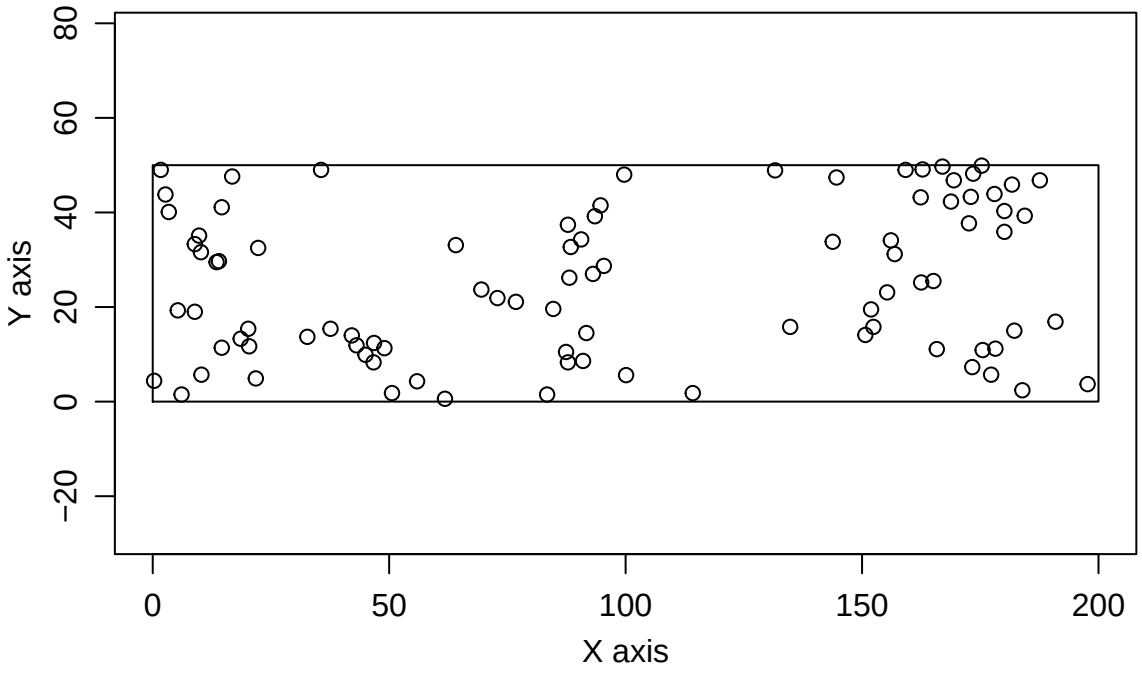
Modifying plot characteristics

This is a base graphics plot, so par() and options to plot() can be used to change plotting characteristics. Some commonly used features are:

- To reduce the white space around the plot: I commonly use the par() statement below to reduce the top and right margins (the mar= bit) and move text in towards the plot (the mgp= bit). When there are x and y axes and axis labels, I use 3.2 lines for the bottom and left margins and 0.2 lines for the top and right margins. If you want minimal space, use mar=rep(0.2, 4). You probably want to suppress the title (see below) if you use minimal space.

- Adding axes: add axes=T to the plot command and leave at least 2 lines in the bottom and left margins
- Adding axis labels: add ann=T, xlab='X axis label', ylab='Y axis label' to the plot command. Also leave at least 3 lines in the bottom and left margins. See below for an example mar= that does that.
- Suppressing the object name: This is automatically printed because plot.ppp() sets main=. If you don't want the object name printed, add main=" to the plot command.

```
par(mar=c(3,3,0,0)+0.2, mgp=c(2,0.8,0))
plot(cypress.pppw, main='', ann=T, xlab='X axis', ylab='Y axis',
  axes=T)
```



Descriptive summaries and pointwise confidence intervals

spatstat has a similar structure for all the different summary functions. One function computes the observed summary statistic. There is one function for each of the various summary statistics. The envelope() function then computes the randomization envelope for any of the summary functions.

The results of any spatstat summary function is a "function value", fv, object. This is a dataframe containing the x values, one or more results, and information about how to plot and label the results. See ?fv if you want to know more.

fv objects enforce common-sense restrictions on combining two components. For example, with data frames, there is nothing to stop you merging one with G(x) evaluated at x=1, 2, and 3 and one evaluated at x=0.5, 1, and 2. That operates row-by-row. cbind() on fv objects makes sure only values with the same x get merged.

The complication is that arithmetic operations have to be done differently for fv objects. I'll demonstrate how to do these as we need them.

Point-point distances:

The Gest() function estimates G(x), the cdf of point-point distances. The required argument is the point pattern. Optionally, you can specify the distances at which G(x) should be evaluated. I rarely do this because the default is usually reasonable.

The result is an fv object with 7 columns, 3 of which are different versions of edge-corrected estimators of G(x).

- r: the distance, what I called x above
- theo: theoretical G(r) for Poisson process (CSR)
- han: Hanisch estimator of G(r)
- rs: reduced-sample = border corrected estimator
- km: Kaplan-Meier estimator - I recommend this
- hazard: KM estimate of the hazard function
- theohaz: theoretical hazard function for a Poisson process

We will not talk about or use hazard functions, other than to say that they are another way to think about event data. Hazard is related to G(x) in the following way. G(x) is P[NN distance < x]. Hazard is P[NN distance = x | NN distance >= x]. When the event is death, hazard is very interpretable. h(x) = P[die at age x | alive at start of age x].

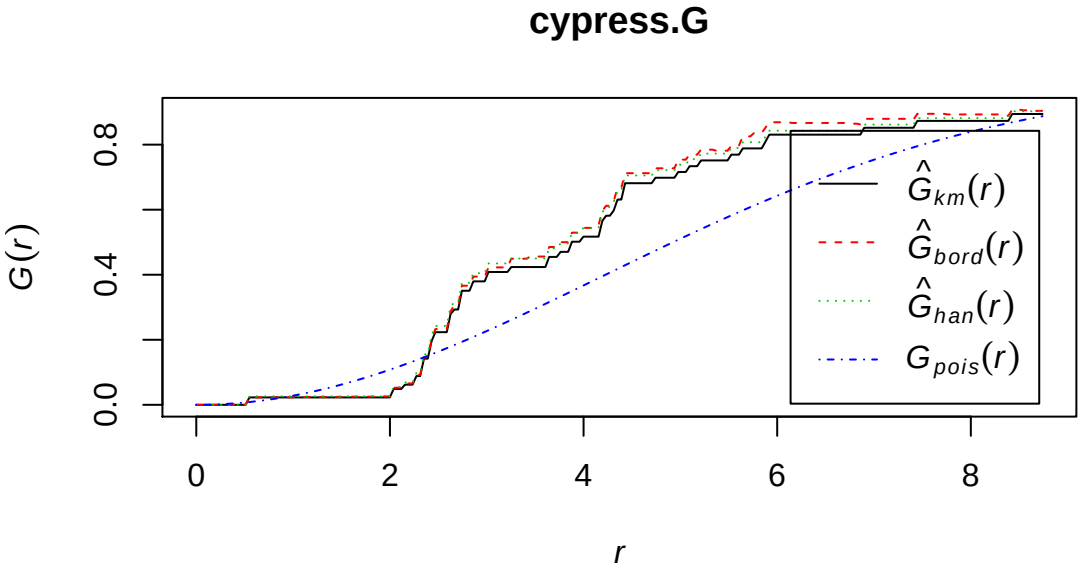
printing an fv object tells you the names of the variables, the nice math expression of them and a short description.

plot() applied to an fv object calls plot.fv(). ?plot.fv is the relevant help file. This does a lot of very reasonable things: draws the relevant curves, labels the axes and nicely labels the lines. Note that parts of hat(G)han turn into a superscript and a subscript.

```
cypress.G <- Gest(cypress.ppp)
cypress.G
```

```
## Function value object (class 'fv')
## for the function r -> G(r)
## ......................................................................
##         Math.label      Description
## r       r               distance argument r
## theo    G[pois](r)      theoretical Poisson G(r)
## han     hat(G)[han](r)  Hanisch estimate of G(r)
## rs      hat(G)[bord](r) border corrected estimate of G(r)
## km      hat(G)[km](r)   Kaplan-Meier estimate of G(r)
## hazard  hat(h)[km](r)   Kaplan-Meier estimate of hazard function h(r)
## theohaz h[pois](r)      theoretical Poisson hazard function h(r)
## ......................................................................
## Default plot formula:  .~r
## where "." stands for 'km', 'rs', 'han', 'theo'
## Recommended range of argument r: [0, 8.7404]
## Available range of argument r: [0, 20.068]
```

```
plot(cypress.G)
```



Customizing the plot

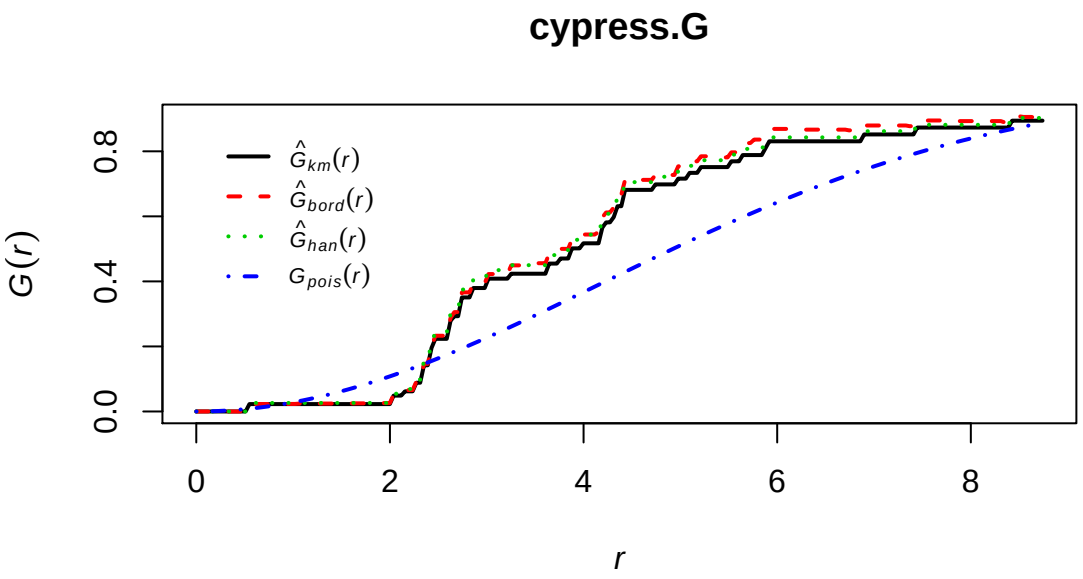
Because plot.fv() uses base graphics, all the base graphics customizations can be used (e.g. mar and mgp to shrink the margins, main=" to omit the title, xlab= and ylab= to relabel the axes).

There are many other useful customizations. All used in the plot command. Each can take a vector (one value for each line) or a scalar (single value used for all lines). The ones I most commonly use are:

- lwd= line width.
- lty= line type. 1=solid, 2 = dashed, 3 = dotted, 4 = dash dot, and more
- col= colors, either a character vector of names or a numeric vector.
  I often find the legend box too big, and most of the time, I prefer to not show the box around the legend. Those customizations are done by legendargs=list( ). The things inside the list() are arguments that could be given to legend() if you were manually adding the legend. Two of the useful of these are:
- cex= expands or shrinks the legend text. cex=0.8 is often sufficiently small
- bty='n' omits the box around the legend

The following will plot the estimated G(x) functions, with double-thick lines, and modifies the legend by shrinking the text and omitting the box.

```
plot(cypress.G, lwd=2, legendargs=list(cex=0.7, bty='n') )
```



Computing simulation envelopes

The envelope() function computes simulation envelopes for any summary function. The two required arguments are the point pattern and the name of the summary function (not in quotes). By default, this does 99 simulations and returns the largest and smallest. This gives a 98% pointwise coverage envelope. Because it's only 99 simulations, it's relatively fast. For research publications, I usually use nsim=999 simulations and return the 2.5% and 97.5% quantiles, which give me a 95% pointwise confidence envelope. For nsim=999, this is specified by nrank = 25. The 25'th value, counting up from the smallest, is the 0.025 quantile = 2.5 percentile of (nsim+1) values. If you've got a really fast computer, and not too large a data set, nsim=4999, nrank=125 gives 95% pointwise envelopes with less Monte-Carlo error.

Why nsim+1? If you remember our earlier discussion of randomization tests, the observed data gives one summary function. Each of the nsim simulations gives another. Total is nsim+1.

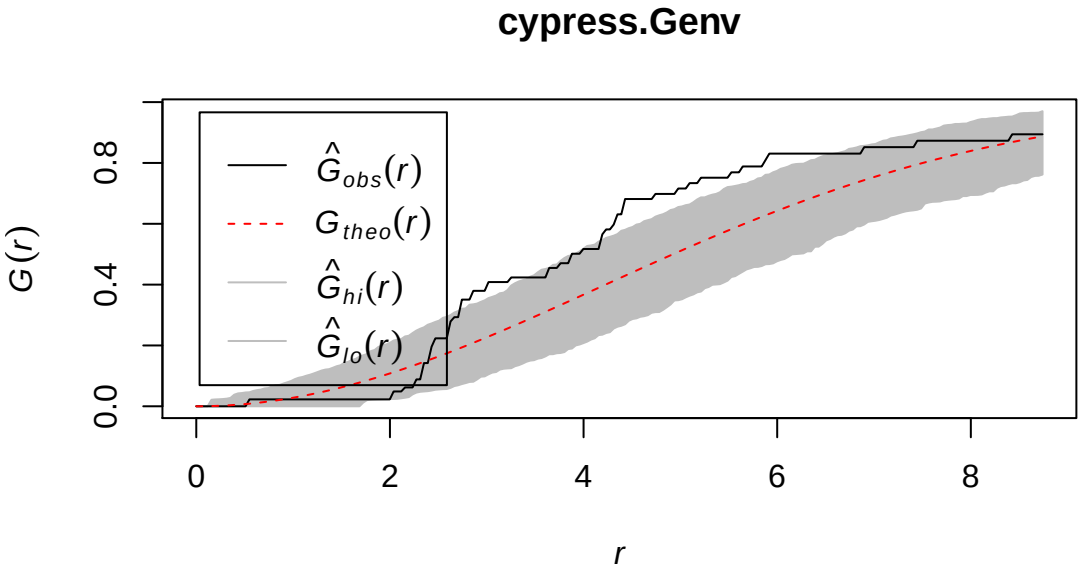
Note: this can be slow or very slow if you ask for lots of simulations of a large data set. You do see a status bar.

The result is an fv object with the estimated G(r), theoretical G(r) and lower and upper bounds of the pointwise envelope around the theoretical curve. Printing the result gives you this summary. Plotting the result gives you a nice comparison of the observed and theoretical curves.

```
cypress.Genv
```

```
## Pointwise critical envelopes for G(r)
## and observed value for 'cypress.ppp'
## Edge correction: "km"
## Obtained from 999 simulations of CSR
## Alternative: two.sided
## Significance level of pointwise Monte Carlo test: 50/1000 = 0.05
## ......................................................................
##      Math.label     Description
## r    r              distance argument r
## obs  hat(G)[obs](r) observed value of G(r) for data pattern
## theo G[theo](r)     theoretical value of G(r) for CSR
## lo   hat(G)[lo](r)  lower pointwise envelope of G(r) from simulations
## hi   hat(G)[hi](r)  upper pointwise envelope of G(r) from simulations
## ......................................................................
## Default plot formula:  .~r
## where "." stands for 'obs', 'theo', 'hi', 'lo'
## Columns 'lo' and 'hi' will be plotted as shading (by default)
## Recommended range of argument r: [0, 8.7404]
## Available range of argument r: [0, 20.068]
```

```
plot(cypress.Genv)
```



You only get one of the possible edge corrections. The default choice for Gest is KM (the one I recommend).

envelope() has a huge number of options if you want to modify some aspect of the simulation. Almost all are only for an advanced user who knows what she is doing or what specifically he wants.

Choosing variables to plot

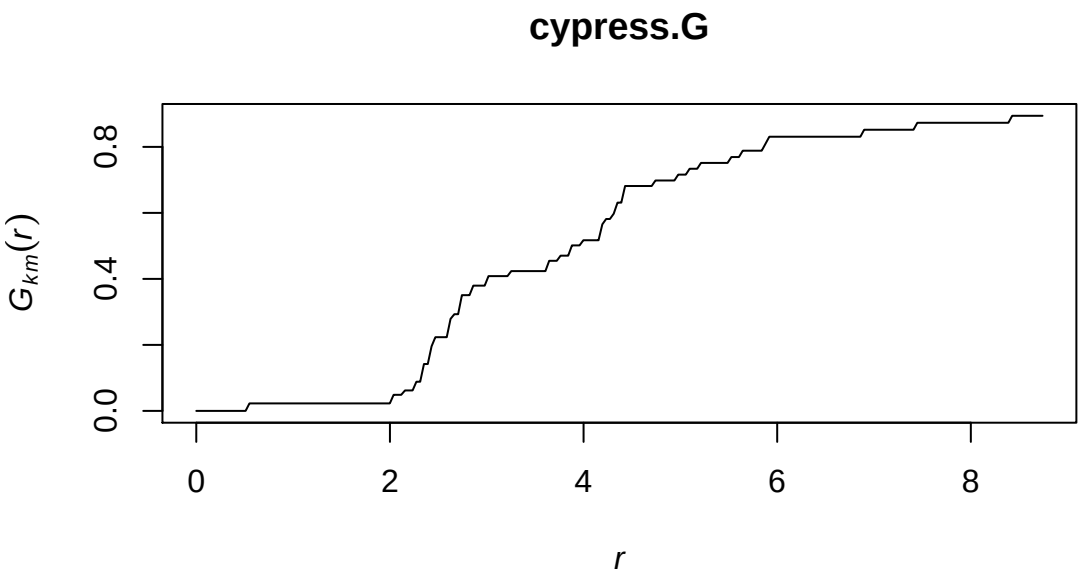
The default plots are often what you need. Many times, you want something slightly different. Most alternative plots are really easy to draw "on the fly".

The logic of fv objects is that each comes with a default plot specification. For example, the default for a Gest result is to plot theo, han, rs and km vs r.

You override the default plot by specifying a formula as the second argument to plot. That formula specifies what is to be plotted on the Y axis (before the ~) and what is to go on the X axis (after the ~). These can be variables or results of computations involving variables in the fv object. These are best described by illustration:

To plot just one variable:

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



The variable names you need are shown when you print the object. Or, you can print them directly using fvnames(). The default is to give you the names of variables plotted by default. Adding '.a' as the second argument gives you all variable names that go (usually) on the Y axis. Adding '.x' gives you the variable name(s) that goes on the X axis.

```
fvnames(cypress.G)
```

```
## [1] "km"   "rs"   "han"  "theo"
```

```
fvnames(cypress.G, '.a')
```

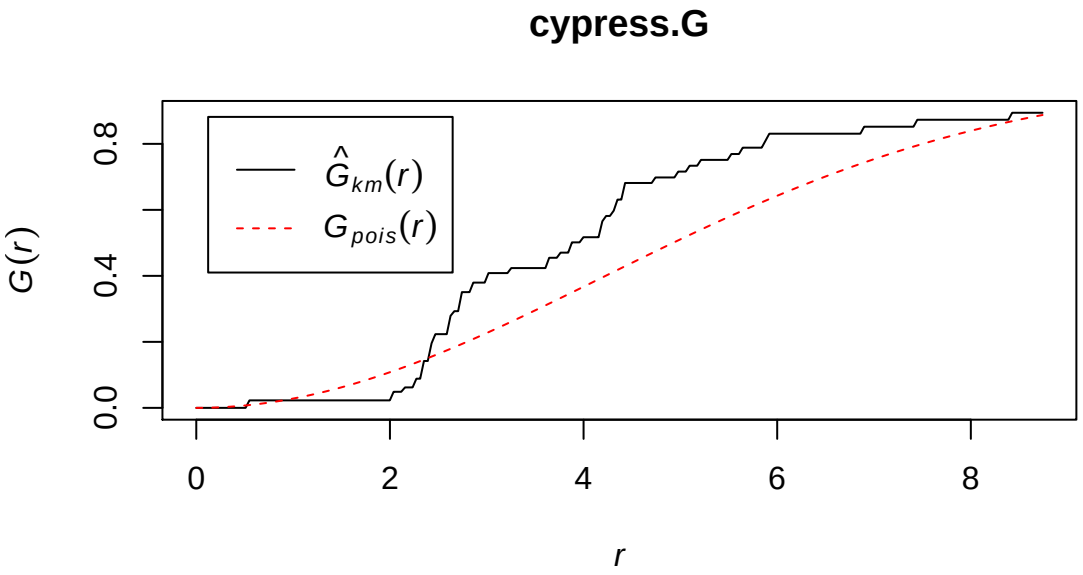
```
## [1] "theo"    "han"     "rs"      "km"      "hazard"  "theohaz"
```

```
fvnames(cypress.G, '.x')
```

```
## [1] "r"
```

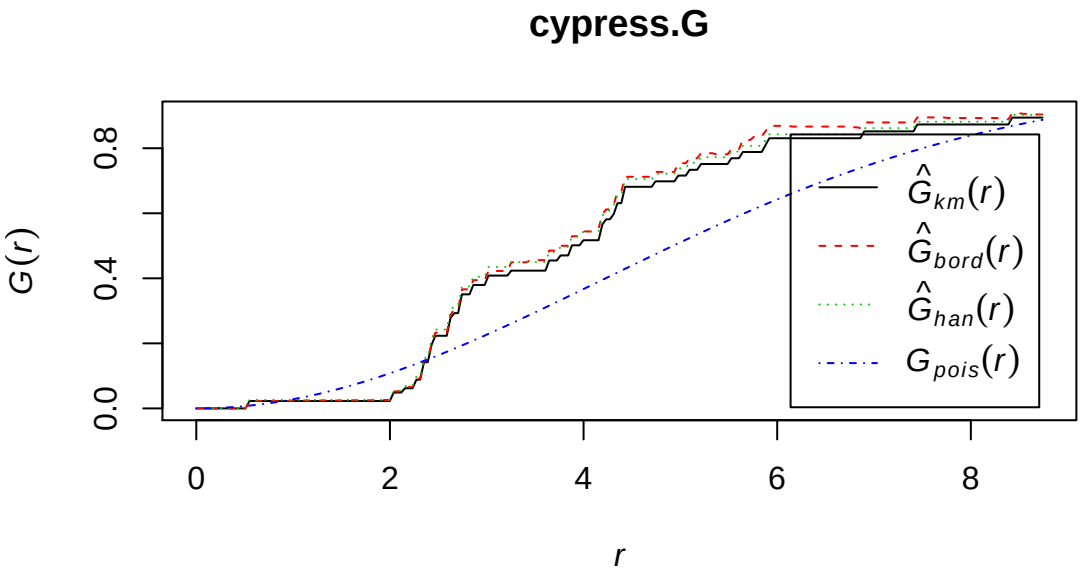
To plot multiple variables:

```
plot(cypress.G, cbind(km, theo) ~ r)
```



To plot all the default variables (usually different estimators of the specified function and not derivatives or hazard functions):

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



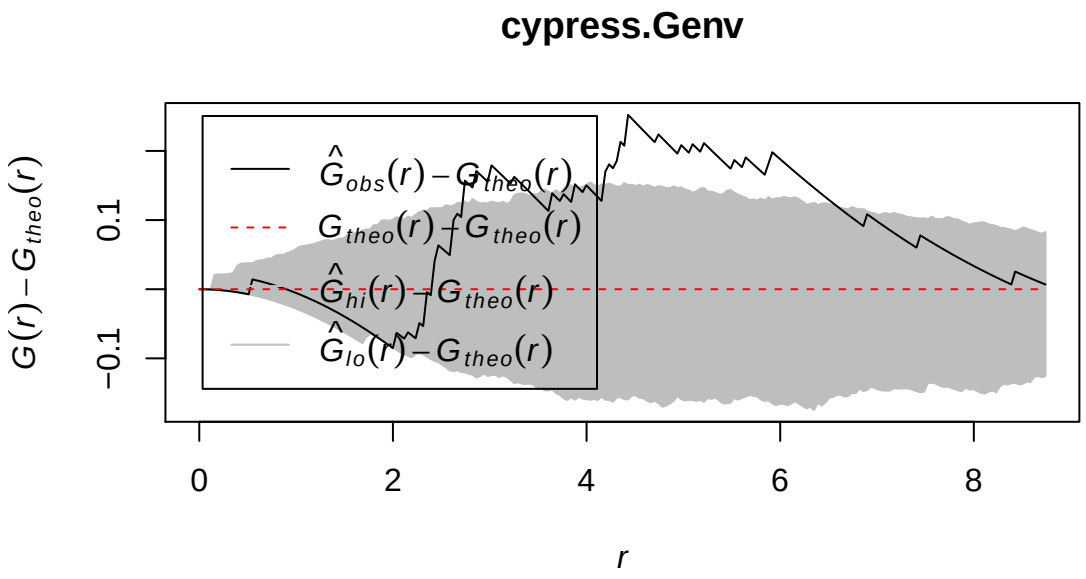
This is the default plot of a Gest result.

Doing computations with fv objects

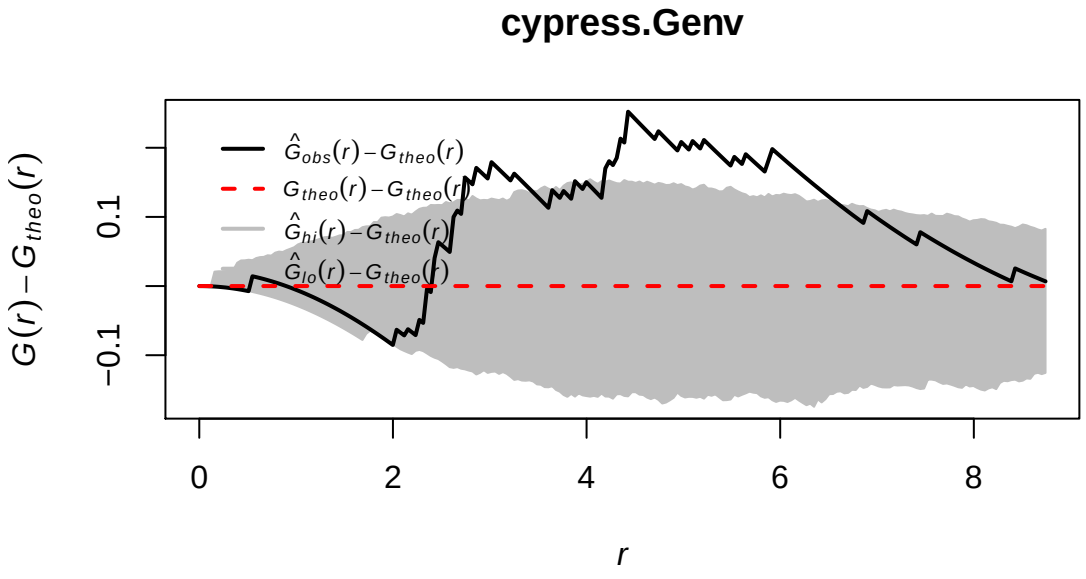
There are two ways to do computations with fv objects. The most common use is a computation inside a plot request. The second is as a stand-alone computation. Computation inside plots is described first.

Although the earlier plot of the estimated G(r) and simulation envelopes is the usual way you will see these plotted, there is a graphical artifact. You are interested in deviations above or below the envelope, but the dominant feature in the plot is the increase of all curves from left to right. That visually minimizes differences from the theoretical value. One way to "flatten" the curve to emphasize differences is to subtract the theoretical value from all curves.

```
plot(cypress.Genv, .-theo ~ r)
```




```
plot(cypress.Genv, .-theo ~ r, lwd=2, legendargs=list(cex=0.7, bty='n'))
```



In the first version, the legend overlaps part of the envelope. Shrinking the legend and removing the box avoids that. Doubling the line width makes the data more obvious.

The fv class is a generalization of a data.frame. That means you can use the usual ways to access and save variables in a data frame. This is advised only when you are working with columns of a single data frame. Why is explained a bit later.

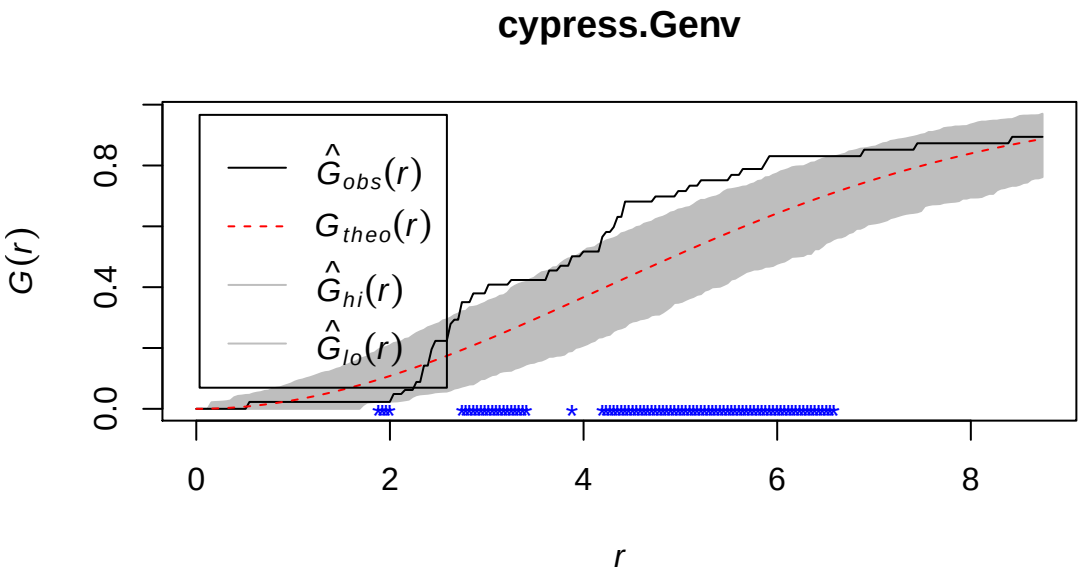
One useful question is "When does G(x) first go abovee the upper side of the simulation envelope". cypress.Genv includes obs (observed G(x)), theo (Poisson theoretical value), hi (upper simulation bound), and lo (lower simulation bound). Conceptually, we need to find the smallest r for which obs > hi. Here's how I do that.

```
temp <- with( cypress.Genv, r[obs > hi] )
min(temp)
```

```
## [1] 2.743628
```

temp contains all the values of r for which obs > hi. We then report the smallest one. Similar operations can tell you whenever obs outside the envelope (obs > hi OR obs < lo), which you can then use to decorate the plot. Spatstat uses base graphics so additions are done using points(), lines(), text() or something related. Here's how to add a blue * along the bottom of the usual G(x) plot whenever obs is outside the envelope. The logical OR operation is the vertical bar, |, character.

```
plot(cypress.Genv)
temp <- with(cypress.Genv, r[ (obs > hi) | (obs < lo) ] )
points(temp, rep(0, length(temp)) , pch='*', col=4)
```



The eval.fv() function does stand-alone computations on fv objects.
The rest of the summary functions

Everything we've done with G(x) = Gest() can be repeated with any summary function offered by spatstat. These include:

- Fest(): point - event distance = F(x)
- Jest(): Baddeley's J(x) function
- Kest(): Ripley's K(x) function (more below)
- Lest(): and the L(x) transformation = sqrt(K(x)/pi) (more below)

- pcf(): pair correlation function = g(x) (more below)

Ripley's K(x) and the related L(x) and g(x) functions

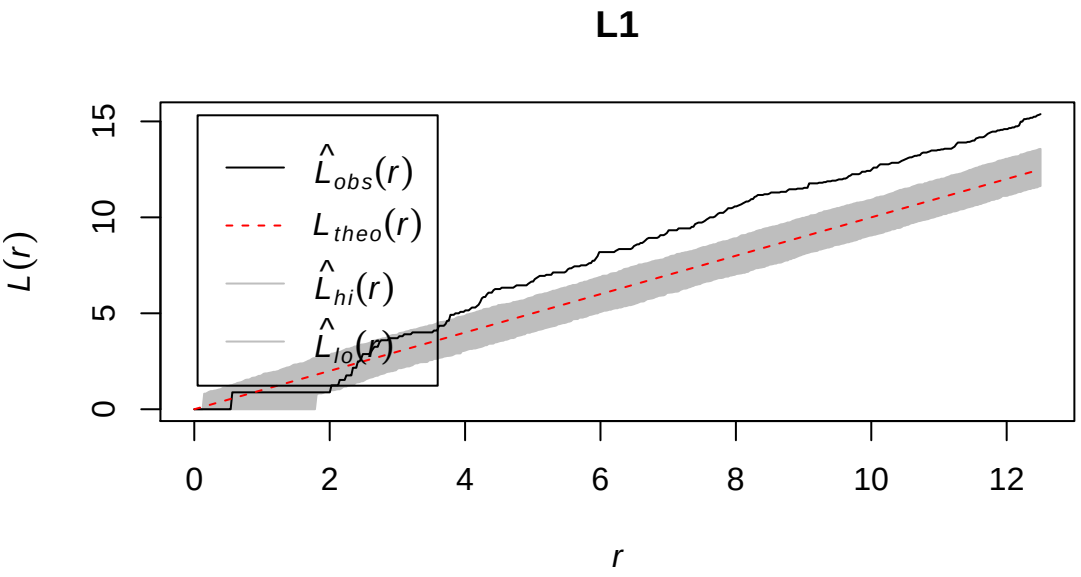
This is the default for envelope(), so if you omit the function (the second argument), envelope uses Kest by default. This is appropriate because K(x) is the most commonly used summary function.

spatstat provides 3 edge corrected estimators of K(x). border is reduced sample estimator and is the equivalent of the rs method for Gest. trans is a translation-based correction. iso is the Ripley isotropic correction. My preference is for the iso method. iso is also the default correction used in envelope.

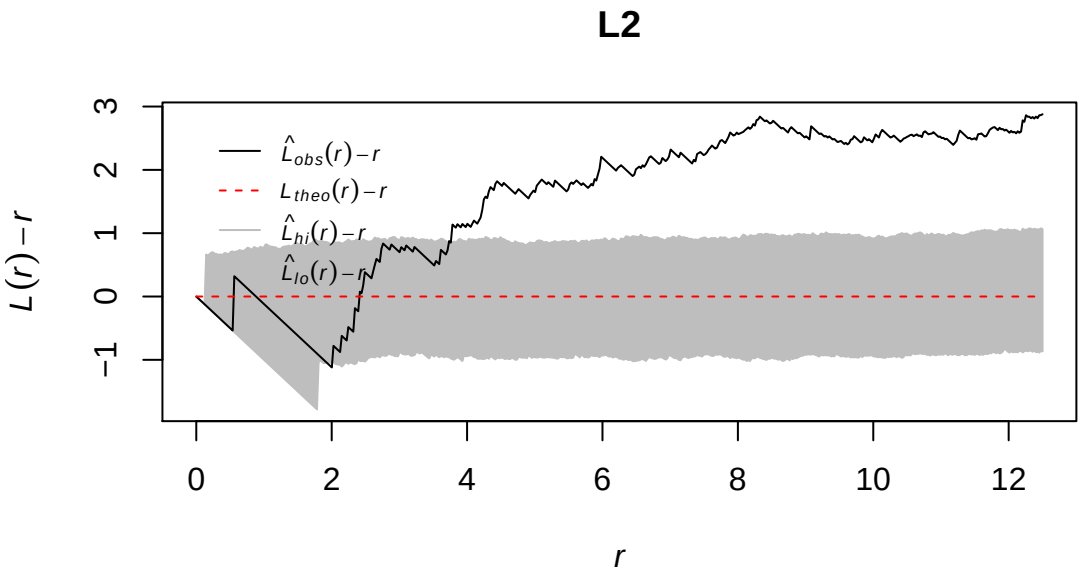
Lest() computes sqrt(K(x) / pi). This is the L1(x) function of Wiegand and Moloney. It has approximately constant variance, but the expected value = x, so differences from the expected value or the envelope can be hard to spot. Because this is computed from K(x), there are the same three edge correction versions. Again, iso is both my recommendation and the default correction for envelope().

I prefer the L2(x) = L1(x) - x function, which has expected value = 0 for all x. This is easy to compute and plot by using the information earlier about computing with fv objects. Here is an example:

```
plot(cypress.lenv, main='L1')
```




```
plot(cypress.lenv, .-r ~ r, main='L2',
  legendargs=list(cex=0.7, bty='n'))
```

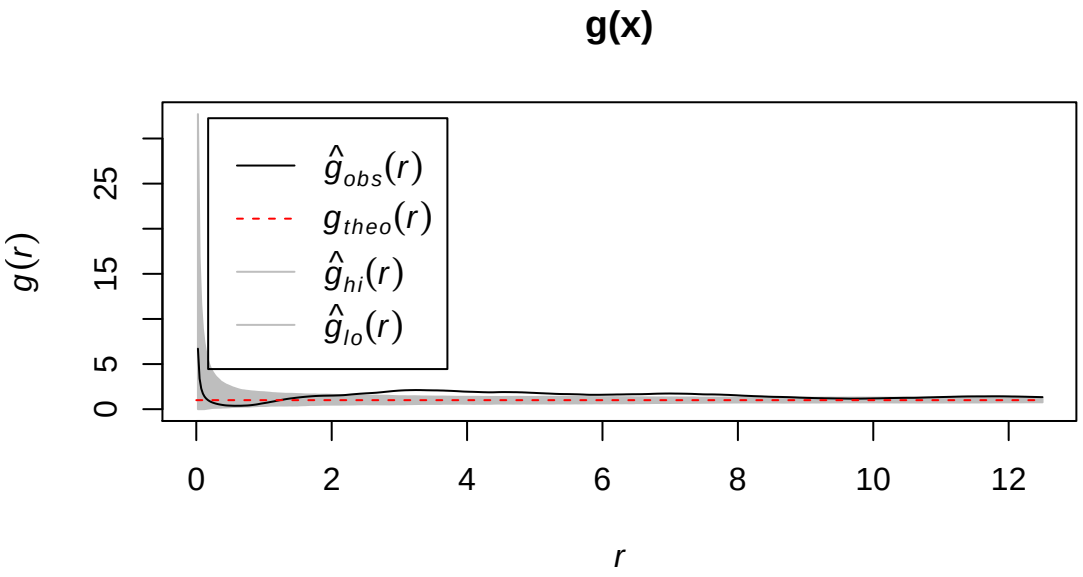


The second plot request takes all variables (the .), subtracts r (the -r) and plots on the Y axis (before the ~) with r on the X axis. If you only wanted to plot some of the variables, use cbind(var1, var2, ...) - r on the left-hand side.

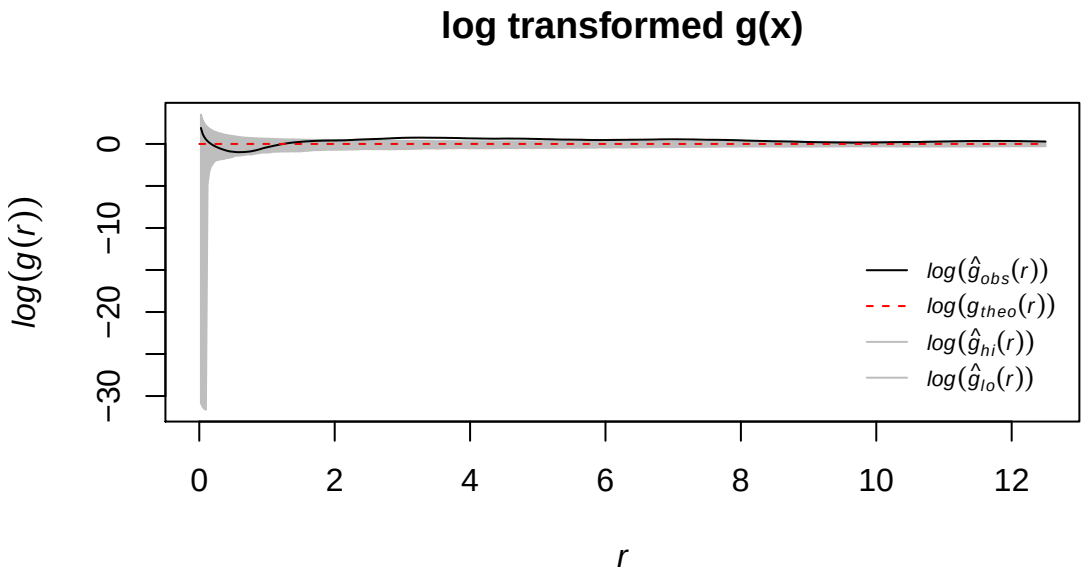
The pair correlation function returns results from 0 to infinity, with a value of 1 for CSR. Hence, it is common to plot log transformed g(x). Here's how to do that:

That's sometimes not very helpful, as here because there are large extremes at very short distances in both plots.

```
plot(cypress.genv, main='g(x)')
```

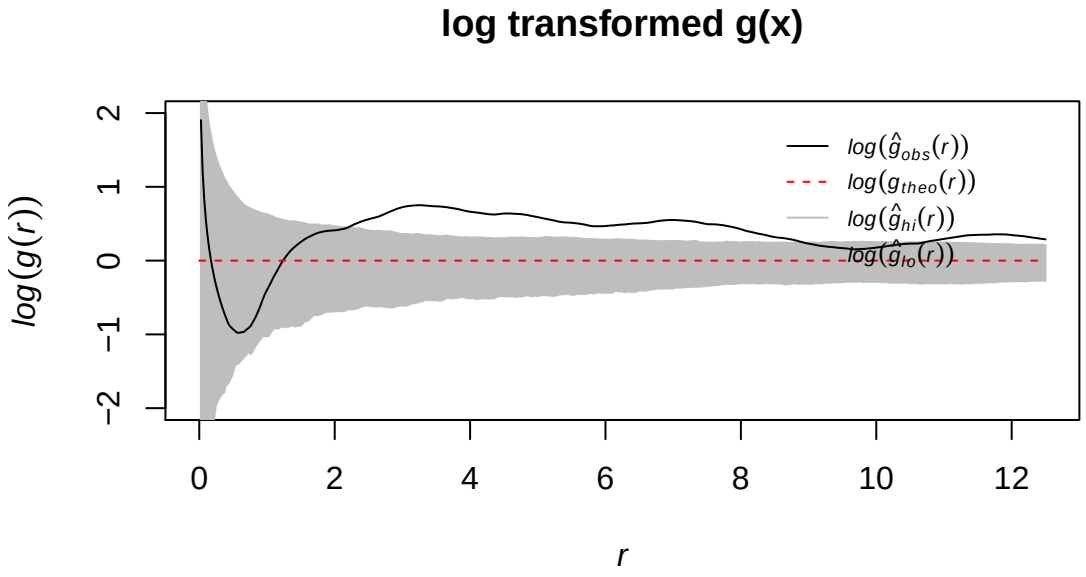


```
plot(cypress.genv, log(.) ~ r, main='log transformed g(x)',
  legendargs=list(cex=0.7, bty='n'))
```



You can brute force fix things by restricting the range of plotted Y values. That is done by specifying ylim=c(min, max).

```
plot(cypress.genv, log(.) ~ r, main='log transformed g(x)',
  legendargs=list(cex=0.7, bty='n'), ylim=c(-2, 2))
```



It sometimes takes some playing with the min and max to plot to get a plot that provides information in the interesting range.