Computer code for tests of spatial pattern

Adrian Baddeley

This document contains computer code in the R language for generating the plots and tables in the paper

On tests of spatial pattern based on simulation envelopes
Baddeley, Diggle, Hardegen, Lawrence, Milne and Nair
Submitted to *Ecology*.

Please note that the results obtained from this code may not be identical to the results in the paper, because the procedures involve randomisation.

Computation times are shown for some of the longer calculations: these timings were measured on a 3 GHz laptop running Linux.

1 Note about files

The original source of this document is a file `baddeley.Rnw` written in the markup language Sweave. This is a "literate programming" document that contains fragments of R code to be executed, surrounded by explanatory text written in LaTeX.

The output file `baddeley.pdf` shows the results of executing the R code (including graphics) and typesetting the surrounding text.

The file `baddeley.R` contains only the R language commands.

2 Setup

Create a directory/folder to save graphics files.

```
> if(!file.exists("baddeley-figures")) dir.create("baddeley-figures")
```

Load the required software.

```
> library(spatstat)
> if(versionstring.spatstat() < "1.32") {
+   stop("spatstat version 1.32-0 or later is required")
+ }
```

If the variable `recompute` is set to `TRUE`, all values will be computed from scratch. If it is `FALSE`, the results of some of the longer computations will be reloaded from files, if available.

```
> recompute <- FALSE
```

Set the state of the random number generator, so that the results will be repeatable, at least on the same computer with the same version of R.

```
> set.seed(54321)
```

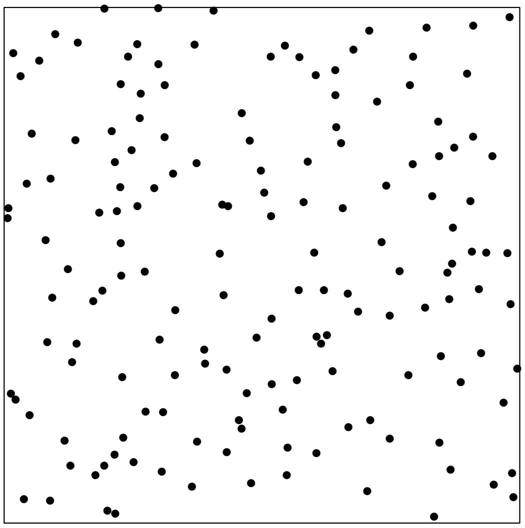
3 Synthetic data example

First we generate some synthetic data in a 100-metre square.

```
> Y <- rStrauss(beta=250/(100^2), gamma=0.5, R=4, W=square(100))
> unitname(Y) <- c("metre", "metres")
```

Plot the data.

```
> plot(Y, pch=16, main="")
```

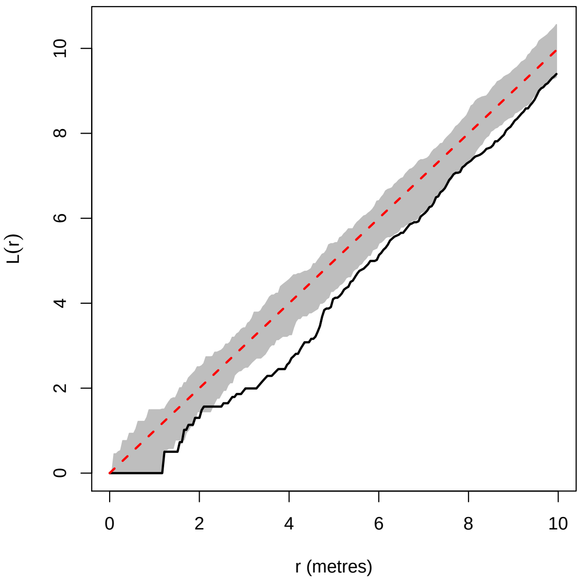


4 Pointwise envelope

Plot the pointwise envelope of the L-function for CSR.

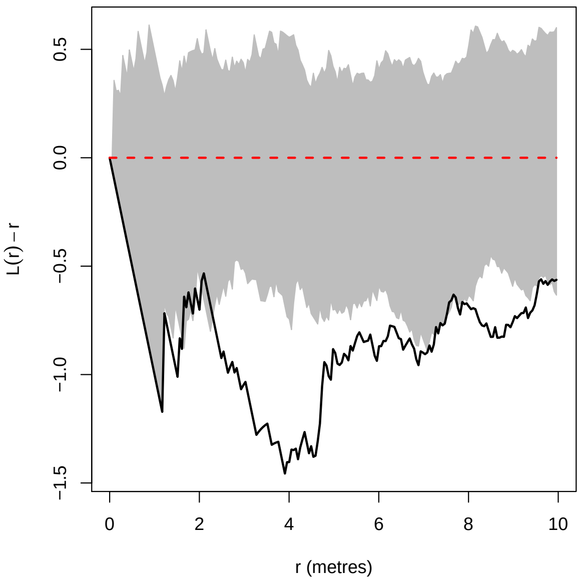
```
> ELp <- envelope(Y, Lest, nsim=39, verbose=FALSE)

> plot(ELp, xlim=c(0,10), main="", lwd=2, legend=FALSE)
```



Plot the envelope of the centred L-function $L(r) - r$ against r. This uses the "plot formula" `. - r ~ r` to indicate the transformation.

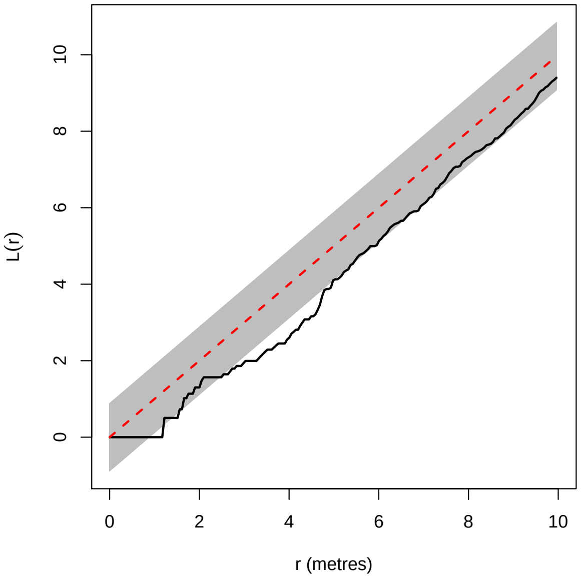
```
> plot(ELp, . - r ~ r, xlim=c(0,10), main="", lwd=2, legend=FALSE)
```



5 Global envelope

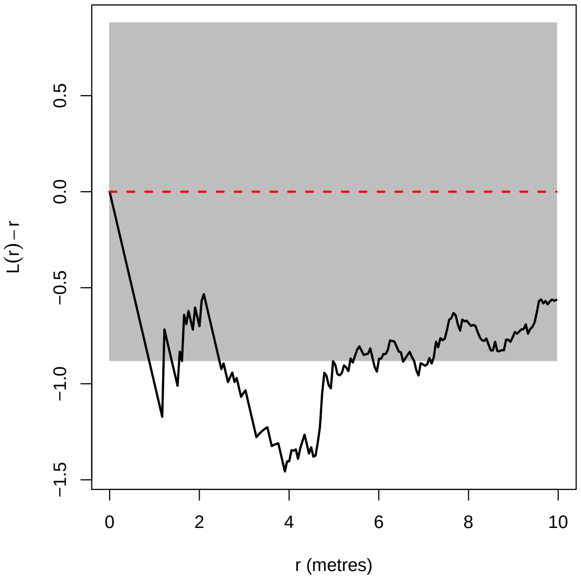
Plot the global envelope of the L-function over the interval from 0 to 10 metres.

```
> ELg <- envelope(Y, Lest, nsim=19, global=TRUE, ginterval=c(0,10), verbose=FALSE)
> plot(ELg, xlim=c(0,10), main="", lwd=2, legend=FALSE)
```



Plot the corresponding envelope of the centred L-function.

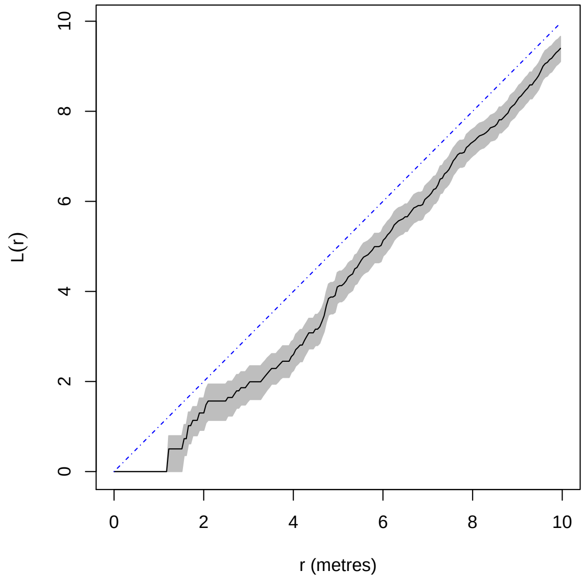
```
> plot(ELg, . - r ~ r, xlim=c(0,10), main="", lwd=2, legend=FALSE)
```



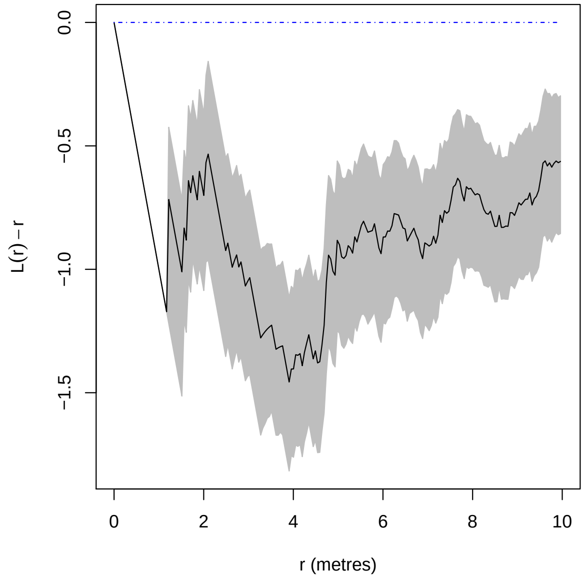
6 Confidence interval

Compute confidence interval for K function using Loh's bootstrap method.

```
> LCI <- lohboot(Y, "Lest", nsim=2000, confidence=0.95)
> plot(LCI, main="", legend=FALSE, xlim=c(0,10))
```



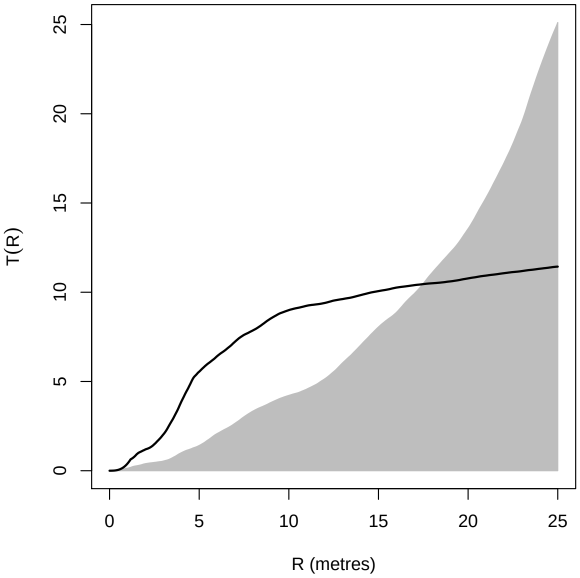
```
> plot(LCI, . - r ~ r, main="", legend=FALSE, xlim=c(0,10))
```



7 Envelope representations of tests

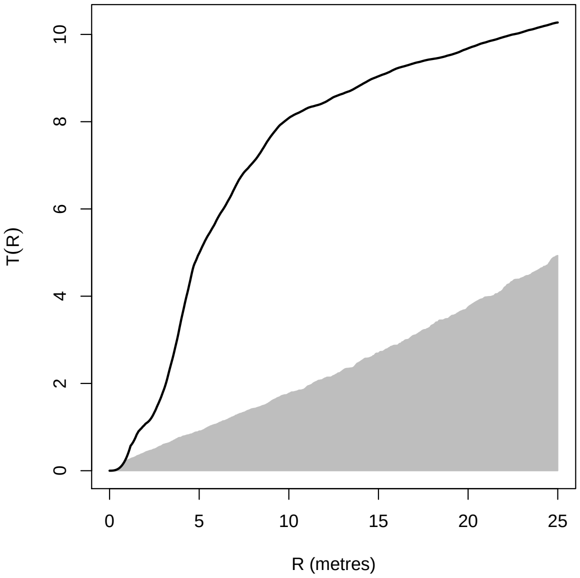
Compute and plot an envelope representation of the DCLF test of size $\alpha = 0.05$.
First, based on 19 simulated values.

```
> erep19 <- dclf.progress(Y, Lest, nsim=19, verbose=FALSE)
> plot(erep19, main="", xlim=c(0,25), lwd=2, legend=FALSE)
```



Second, based on 1999 simulated values (critical value is the 100th largest).

```
> erep1999 <- dclf.progress(Y, Lest, nsim=1999, nrank=100, verbose=FALSE)
> plot(erep1999, main="", xlim=c(0,25), lwd=2, legend=FALSE)
```



8 Power curves

Here we compute power curves for the tests of CSR against two alternatives. The following parameter determines the number of simulations that will be used to estimate the power. It should be set to a large number if you want accurate values.

```
> Ntest <- 1000
```

Total computation time on a laptop is about 5 seconds for each simulation so setting `Ntest` equal to 1000 will take about 85 minutes.

8.1 Additional code

Define the stabilised G-function, $G^{\dagger}(r) = \arcsin\sqrt{G(r)}$.

```
> Gstab <- function(X, ...) {
+   GX <- Gest(X, ...)
+   eval.fv(asin(sqrt(GX)))
+ }
```

Here is a function that can be used to measure power against any specified alternative.

```
> powersim <- function(expr, N, fun=Lest, test=c("dclf", "mad"), nsim=19,
+                      rmax=25, nr=512) {
+   # validate arguments
+   stopifnot(is.expression(expr))
+   test <- match.arg(test)
+   progfun <- switch(test, dclf=dclf.progress, mad=mad.progress)
+   # create space to store results
+   reject <- matrix(FALSE, nr, N)
+   # distance values 'r' (this is not normally required but saves time)
+   r <- seq(0, rmax, length=nr)
+   # run loop
+   for(i in 1:N) {
+     # generate realisation from alternative
+     X <- eval(expr)
+     # compute test statistic and critical value for each [0,R]
+     pr <- progfun(X, fun, nsim=19, r=r, verbose=FALSE)
+     # determine outcome of test for each [0,R]
+     reject[,i] <- with(pr, obs >= crit)
+   }
+   # compute power
+   power <- rowMeans(reject, na.rm=TRUE)
+   return(list(x=r, y=power))
+ }
```

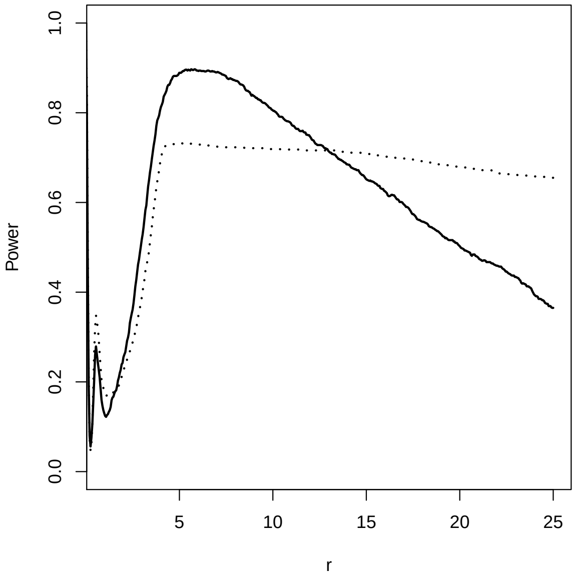
8.2 Spatial inhibition

Now we use the function `powersim` to evaluate the power curves for a model of spatial inhibition.

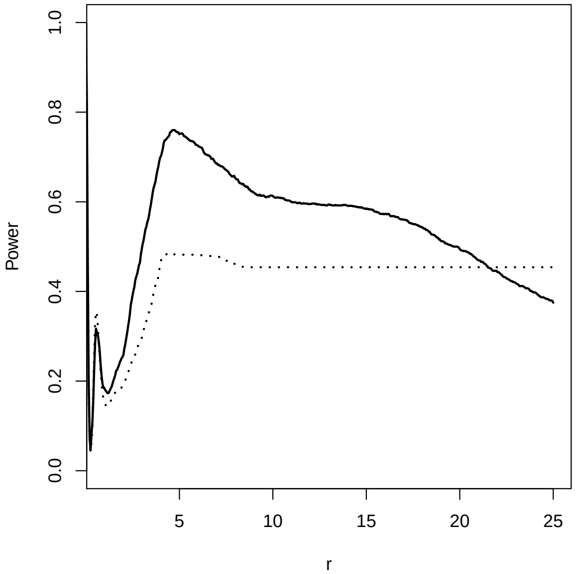
```
> if(recompute || !file.exists("powerinhib.rda")) {
+   W <- square(100)
+   expr <- expression(rStrauss(beta=0.025, gamma=0.5, R=4, W=W))
+   powerIDL <- powersim(expr, Ntest, fun=Lest, test="dclf")
+   powerIDG <- powersim(expr, Ntest, fun=Gstab, test="dclf")
+   powerIML <- powersim(expr, Ntest, fun=Lest, test="mad")
+   powerIMG <- powersim(expr, Ntest, fun=Gstab, test="mad")
+   save(powerIDL, powerIDG, powerIML, powerIMG, file="powerinhib.rda")
+ } else load("powerinhib.rda")
```

Plot curves

```
> plot(powerIDL, type="l",
+      main="", xlab="r", ylab="Power",
+      ylim=c(0,1), xlim=c(1,25),
+      lwd=2)
> lines(powerIML, lwd=2, lty=3)
```



```
> plot(powerIDG, type="l",
+      main="", xlab="r", ylab="Power",
+      ylim=c(0,1), xlim=c(1,25), lwd=2)
> lines(powerIMG, lwd=2, lty=3)
```



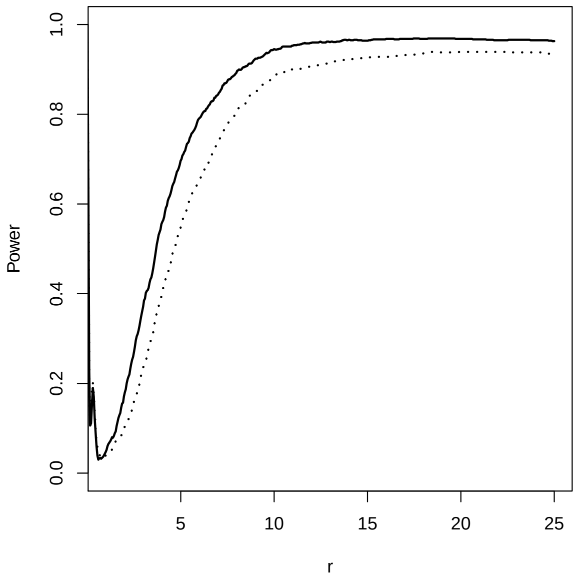
8.3 Spatial clustering

Now we use the function `powersim` to evaluate the power curves for a model of spatial clustering.

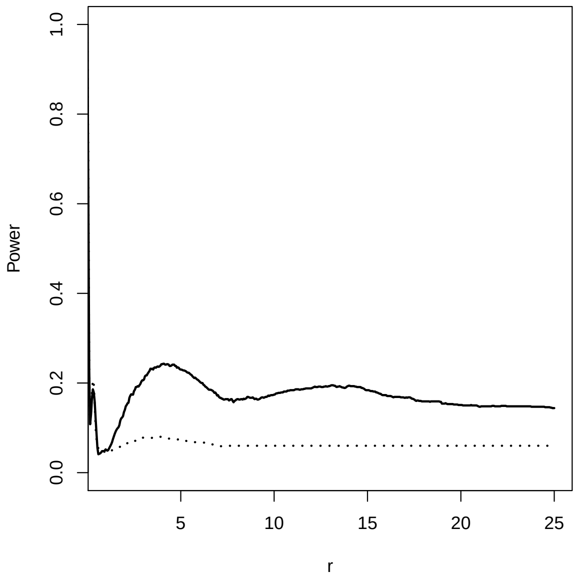
```
> if(recompute || !file.exists("powerclust.rda")) {
+   W <- square(100)
+   expr <- expression(rMatClust(kappa=0.005, r=14, mu=5, win=W))
+   powerCDL <- powersim(expr, Ntest, fun=Lest, test="dclf")
+   powerCDG <- powersim(expr, Ntest, fun=Gstab, test="dclf")
+   powerCML <- powersim(expr, Ntest, fun=Lest, test="mad")
+   powerCMG <- powersim(expr, Ntest, fun=Gstab, test="mad")
+   save(powerCDL, powerCDG, powerCML, powerCMG, file="powerclust.rda")
+ } else load("powerclust.rda")
```

Plot curves

```
> plot(powerCDL, type="l",
+      main="", xlab="r", ylab="Power",
+      ylim=c(0,1), xlim=c(1,25), lwd=2)
> lines(powerCML, lwd=2, lty=3)
```



```
> plot(powerCDG, type="l",
+      main="", xlab="r", ylab="Power",
+      ylim=c(0,1), xlim=c(1,25), lwd=2)
> lines(powerCMG, lwd=2, lty=3)
```



Note that, when several different tests are to be compared, there are more efficient ways to measure power, by re-using the simulations. It is also possible to run simulations on several computers and pool the results afterward. For simplicity, these techniques are not covered here. See the `spatstat` help files for `envelope`, `pool.envelope` and `dclf.progress` for more information.

9 Exact K function for Hard Core process

Next we compute, by extensive simulation, the *true* K-function of the Hard Core process.

Here is a simulation routine that estimates the K function of any point process, assuming that realisations of the process are generated by evaluating the expression `expr`.

We'll need a lot more than 100 simulations to get an accurate estimate!

```
> Kmcmc <- function(expr, nsim=100) {
+   X <- eval(expr)
+   Y <- envelope(X, Kest, nsim=nsim,
+                 simulate=expr,
+                 weights=npairs, VARIANCE=TRUE)
+   # reformat
+   Y <- vanilla.fv(Y)
+   fvnames(Y, ".y") <- "mmean"
+   fvnames(Y, ".s") <- c("loCI", "hiCI")
+   discard <- names(Y) %in% c("obs", "lo", "hi")
+   Y <- Y[, !discard]
+   return(Y)
+ }
> npairs <- function(X) {
+   n <- npoints(X)
+   return(n * (n-1))
+ }
```

The following code prescribes how to simulate a realisation of the Hard Core process.

```
> beta <- 0.06
> R <- 4
> W    <- square(100)
> Wbig <- grow.rectangle(W, 2 * R)
> simexpr <- expression(rHardcore(beta=beta, R=R, W=Wbig)[W])
```

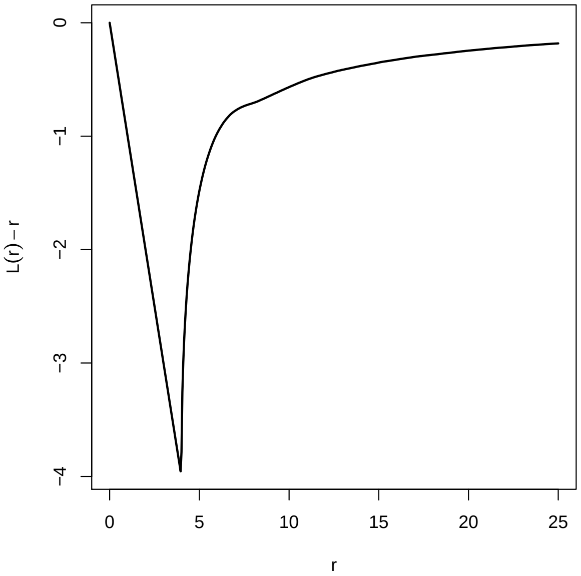
Perform `Nsim` realisations, or load the precomputed results from a file.

The results stored in the file `"Khard.rda"` were computed using 10,000 realisations, which took 7 hours (about 2.5 seconds per realisation). For safety we have set `Nsim = 100` which takes about 4 minutes to recompute.

```
> Nsim <- 100
> if(recompute || !file.exists("Khard.rda")) {
+   Khard <- Kmcmc(simexpr, Nsim)
+   save(Khard, file="Khard.rda")
+ } else load("Khard.rda")
```

Plot the centred L-function.

```
> plot(Khard,
+      sqrt(cbind(mmean, loCI, hiCI)/pi) -r ~ r,
+      ylab=expression(L(r)-r),
+      main="", legend=FALSE, lwd=2)
```

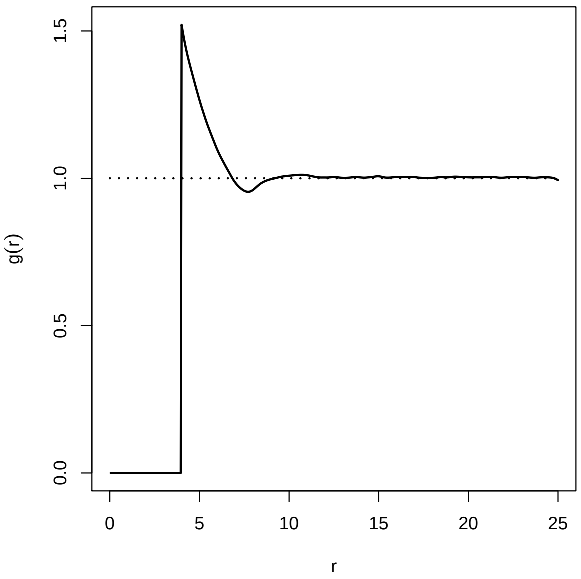


Compute the pair correlation function by numerical differentiation of the K-function, assuming there is a discontinuity in the derivative at $r = 4$ metres.

```
> DKhard <- deriv(Khard, which="mmean", kinks=4, spar=0.4)
```

Compute $g(r) = K'(r)/(2\pi r)$ and plot.

```
> plot(DKhard,
+      cbind(mmean/(2 * pi * r), 1) ~ r,
+      ylab=expression(g(r)),
+      col=1, lty=c(1,3), lwd=2,
+      main="", legend=FALSE)
```



10 Conservative tests

10.1 Tests of CSR

Here we estimate the true significance level (probability of Type I error) for the DCLF test of CSR with nominal significance 0.05, with or without conditioning on the number of points. This was reported in Table 2 of the paper.

First we write a function to perform the simulations. To save space, we store the ranks rather than the Monte Carlo p-values.

```
> CSR.ranks.dclf <- function(Npatterns=1000,
+                            ...,
+                            nmc=19,
+                            lambda=0.05,
+                            W = square(100),
+                            rmaxK=25,
+                            rmaxG=10,
+                            verbose=TRUE) {
+   mKpois <- mGpois <- mKcond <- mGcond <-
+     mLpois <- mLcond <- mHpois <- mHcond <- integer(Npatterns)
+   # 'H' denotes asin(sqrt(G))
+   KtoL <- expression(sqrt(./pi))
+   GtoH <- expression(asin(sqrt(.)))
+   #
+   rK <- seq(0, rmaxK, length=512)
+   rG <- seq(0, rmaxG, length=512)
+   areaW <- area.owin(W)
+   #
+   for(i in 1:Npatterns) {
+     X <- rpoispp(lambda, win=W)
+     nX <- npoints(X)
+     lamX <- nX/areaW
+     # generate common simulations
+     XsimP <- XsimC <- vector(mode="list", length=nmc)
+     for(j in 1:nmc) {
+       XsimP[[j]] <- rpoispp(lamX, win=W)
+       XsimC[[j]] <- runifpoint(nX, win=W)
+     }
+     # .......... Poisson simulations ....................
+     # compute K functions and apply DCLF test
+     dcKpois <- dclf.test(X, fun=Kest, r=rK, simulate=XsimP,
+                          savefuns=TRUE,
+                          nsim=nmc, verbose=FALSE)
+     mKpois[i] <- as.integer(dcKpois$statistic$rank)
+     # transform to L, apply DCLF test
+     dcLpois <- dclf.test(dcKpois, transform=KtoL,
+                          nsim=nmc, verbose=FALSE)
+     mLpois[i] <- as.integer(dcLpois$statistic$rank)
+     # compute G functions and apply DCLF test
+     dcGpois <- dclf.test(X, fun=Gest, r=rG, simulate=XsimP,
+                          savefuns=TRUE,
+                          nsim=nmc, verbose=FALSE)
+     mGpois[i] <- as.integer(dcGpois$statistic$rank)
+     # stabilise variance, apply DCLF test
+     dcHpois <- dclf.test(dcGpois, transform=GtoH,
+                          nsim=nmc, verbose=FALSE)
+     mHpois[i] <- as.integer(dcHpois$statistic$rank)
+
+     # .......... conditional simulations ....................
+     # compute K functions and apply DCLF test
+     dcKcond <- dclf.test(X, fun=Kest, r=rK, simulate=XsimC,
+                          savefuns=TRUE,
+                          nsim=nmc, verbose=FALSE)
+     mKcond[i] <- as.integer(dcKcond$statistic$rank)
+     # transform to L, apply DCLF test
+     dcLcond <- dclf.test(dcKcond, transform=KtoL,
+                          nsim=nmc, verbose=FALSE)
+     mLcond[i] <- as.integer(dcLcond$statistic$rank)
+     # compute G functions and apply DCLF test
+     dcGcond <- dclf.test(X, fun=Gest, r=rG, simulate=XsimC,
+                          savefuns=TRUE,
+                          nsim=nmc, verbose=FALSE)
+     mGcond[i] <- as.integer(dcGcond$statistic$rank)
+     # stabilise variance, apply DCLF test
+     dcHcond <- dclf.test(dcGcond, transform=GtoH,
+                          nsim=nmc, verbose=FALSE)
+     mHcond[i] <- as.integer(dcHcond$statistic$rank)
+     #
+     if(verbose)
+       progressreport(i, Npatterns)
+   }
+   return(as.matrix(data.frame(mKpois, mKcond,
+                               mLpois, mLcond,
+                               mGpois, mGcond,
+                               mHpois, mHcond
+                               )))
+ }
```

Now execute, or load the result of 100,000 simulations from the file. Computation time is about 3–4 seconds per simulation, so 1000 simulations would take about an hour.

```
> if(recompute || !file.exists("CSRranks.rda")) {
+   CSRranks <- CSR.ranks.dclf(1000, verbose=FALSE)
+   save(CSRranks, file="CSRranks.rda")
+ } else load("CSRranks.rda")
```

Compute the estimated probability of Type I error. Present as in Table 1 of the paper.

```
> tab1 <- colMeans(CSRranks == 1)
> tab1

 mKpois  mKcond  mLpois  mLcond  mGpois  mGcond  mHpois  mHcond
0.04332 0.04980 0.03902 0.04980 0.02387 0.05040 0.02570 0.05023

> tab1 <- matrix(tab1, ncol=2, byrow=TRUE)
> dimnames(tab1) <- list(c("K", "L", "G", "H"), c("Uncond", "Cond"))
> tab1

   Uncond    Cond
K 0.04332 0.04980
L 0.03902 0.04980
G 0.02387 0.05040
H 0.02570 0.05023
```

10.2 Tests of fitted model

Here we estimate the true significance level (probability of Type I error) for the DCLF test of a fitted model.

This code contains additional safety checks to deal with problems encountered in fitting the model or in simulating the fitted model.

```
> MatClust.ranks.dclf <- function(Npatterns=1000,
+                                  ...,
+                                  nmc=19,
+                                  kappa=0.005, r=14, mu=5, W=square(100),
+                                  rmaxK=25,
+                                  rmaxG=10,
+                                  verbose=TRUE) {
+   mK <- mG <- mL <- mH <- rep(NA_integer_, Npatterns)
+   # 'H' denotes asin(sqrt(G))
+   KtoL <- expression(sqrt(./pi))
+   GtoH <- expression(asin(sqrt(.)))
+   #
+   rK <- seq(0, rmaxK, length=512)
+   rG <- seq(0, rmaxG, length=512)
+   #
+   for(i in 1:Npatterns) {
+     # generate realisation of Matern Cluster process
+     X <- rMatClust(kappa=kappa, r=r, mu=mu, win=W)
+     # fit Matern Cluster model to X by minimum contrast
+     fit <- kppm(X, ~1, clusters="MatClust")
+     # generate simulations from fitted 'null' model for use in tests
+     Xsim <- simulate(fit, nsim=nmc, verbose=FALSE, retry=0)
+     # check there are no NULL outcomes i.e. simulation failures
+     if(!any(unlist(lapply(Xsim, is.null)))) {
+       # compute K functions and apply DCLF test
+       dcK <- try(dclf.test(X, fun=Kest, r=rK, simulate=Xsim,
+                            savefuns=TRUE,
+                            nsim=nmc, verbose=FALSE))
+       if(!inherits(dcK, "try-error")) {
+         mK[i] <- as.integer(dcK$statistic$rank)
+         # transform to L, apply DCLF test
+         dcL <- dclf.test(dcK, transform=KtoL,
+                          nsim=nmc, verbose=FALSE)
+         mL[i] <- as.integer(dcL$statistic$rank)
+       }
+       # compute G functions and apply DCLF test
+       dcG <- try(dclf.test(X, fun=Gest, r=rG, simulate=Xsim,
+                            savefuns=TRUE,
+                            nsim=nmc, verbose=FALSE))
+       if(!inherits(dcG, "try-error")) {
+         mG[i] <- as.integer(dcG$statistic$rank)
+         # stabilise variance, apply DCLF test
+         dcH <- dclf.test(dcG, transform=GtoH,
+                          nsim=nmc, verbose=FALSE)
+         mH[i] <- as.integer(dcH$statistic$rank)
+       }
+     }
+     #
+     if(verbose)
+       progressreport(i, Npatterns)
+   }
+   return(as.matrix(data.frame(mK, mL, mG, mH)))
+ }
```

Now execute 5000 simulations, or load the result of 50,000 simulations from the file. Computation time is only 1–2 seconds per simulation so 5000 simulations takes up to 2 hours.

```
> if(recompute || !file.exists("MatClustranks.rda")) {
+   MatClustranks <- MatClust.ranks.dclf(5000, verbose=FALSE)
+   save(MatClustranks, file="MatClustranks.rda")
+ } else load("MatClustranks.rda")
```

Compute the estimated probability of Type I error. This is Table 2.

```
> colMeans(MatClustranks == 1, na.rm=TRUE)
```

```
         mK          mL          mG          mH 
0.011497272 0.010284906 0.001778137 0.002768236 
```