```
####################################
# M function and confidence intervals
# E. Marcon, Eric.Marcon@ecofog.gf
# Version 15/07/2012
####################################

require("spatstat")

################
# M.r
#   Estimates the M function
# Arguments
#   X : a point pattern (ppp object), marks must be a dataframe with two columns
#     PointType : labels, as factors
#     PointWeight : weights
#   r : A vector of distances
#   ReferenceType : one of the point types
#   NeighborType : one of the point types
#   CaseControl : if TRUE, the case-control version of M is computed.
#                 ReferenceType points are cases, NeighborType points are controls
# Value
#   A vector containing M values for all distances
################
M.r <-
function(X, r, ReferenceType, NeighborType, CaseControl=FALSE) {
  # Eliminate erroneous configurations
  if (CaseControl & (ReferenceType==NeighborType)) {
    warning("Cases and controls are identical.")
    return(rep(1,length(r)))
  }

  # Compute the matrix of distances (squared to save time)
  Dist <- pairdist.ppp(X, squared=TRUE)

  # Vectors to recognize point types
  IsReferenceType <- X$marks$PointType==ReferenceType
  IsNeighborType <- X$marks$PointType==NeighborType

  # Global ratio
  if (ReferenceType==NeighborType | CaseControl) {
    WrMinusReferencePoint <- sum(X$marks$PointWeight[IsReferenceType])-
X$marks$PointWeight
    Wn <- WrMinusReferencePoint[IsReferenceType]
  } else {
    Wn <- sum(X$marks$PointWeight[IsNeighborType])
  }
  if (CaseControl) {
    Wa <- sum(X$marks$PointWeight[IsNeighborType])
  } else {
    WaMinusReferencePoint <- sum(X$marks$PointWeight)-X$marks$PointWeight
    Wa <- WaMinusReferencePoint[IsReferenceType]
  }
  GlobalRatio <- Wn/Wa

  # M.distance calculates M at a single distance
  M.distance <- function(distance) {
    # Select point pairs less than r apart
    IsCloseEnough <- (Dist <= distance)
    # Eliminate point pairs made of a single point
    diag(IsCloseEnough) <- FALSE
    # Calculate the weight of neighbors
    IsCloseEnough <- IsCloseEnough[IsReferenceType, ]
    if (CaseControl) {
      IsCloseEnoughAndCase <- t(t(IsCloseEnough) & IsReferenceType)
      NeighborTypeWeight <- IsCloseEnoughAndCase %*% X$marks$PointWeight
      IsCloseEnoughAndControl <- t(t(IsCloseEnough) & IsNeighborType)
      AllNeighborWeight <- IsCloseEnoughAndControl %*% X$marks$PointWeight
```

```
    } else {
      IsCloseEnoughAndNeighborType <- t(t(IsCloseEnough) & IsNeighborType)
      NeighborTypeWeight <- IsCloseEnoughAndNeighborType %*% X$marks$PointWeight
      AllNeighborWeight <- IsCloseEnough %*% X$marks$PointWeight
    }
    # Calculate the local ratio
    LocalRatio <- NeighborTypeWeight/AllNeighborWeight
    # Calculate M, eliminate undefined values (no neighbor of any type)

return(sum(LocalRatio[is.finite(LocalRatio)])/sum(GlobalRatio[is.finite(LocalRatio)]))
  }

  return(sapply(r*r, M.distance))
}

################
# SimulateM
#   Randomizes the point pattern and calculates the M function
# Arguments
#   dummy : a dummy value to allow sapply(1:NumberOfSimulations, SimulateM, ...)
#   X : a point pattern (ppp object), marks must be a dataframe with two columns
#     PointType : labels, as factors
#     PointWeight : weights
#   r : A vector of distances
#   ReferenceType : one of the point types
#   NeighborType : one of the point types
#   SimulationType : the null hypothesis, may be:
#     RandomLocation : points are redistributed on the actual locations
#     RandomLabeling : randomizes point types, keeping locations and weights unchanged
#     PopulationIndependence : keeps reference points unchanged, randomizes other point
locations
#   CaseControl : if TRUE, the case control version of M is computed.
#                 ReferenceType points are cases, NeighborType points are controls
# Value
#   A vector containing M values for all distances
################
SimulateM <-
function(dummy, X, r, ReferenceType, NeighborType, SimulationType="RandomLocation",
CaseControl=FALSE) {
  SimulatedPP <- switch (SimulationType,
    RandomLocation = rlabel(X),
    RandomLabeling = RandomLabeling.M(X),
    PopulationIndependence = PopulationIndependence.M(X, ReferenceType)
    )
  return(M.r(SimulatedPP, r, ReferenceType, NeighborType, CaseControl))
}

################
# MEnvelope
#   Calculates the confidence envelope of M for null hypotheses
# Arguments
#   NumberOfSimulations : the number of simulations to draw
#   Alpha : the risk level
#   X : a point pattern (ppp object), marks must be a dataframe with two columns
#     PointType : labels, as factors
#     PointWeight : weights
#   r : A vector of distances
#   ReferenceType : one of the point types
#   NeighborType : one of the point types
#   SimulationType : the null hypothesis, may be:
#     RandomLocation : points are redistributed on the actual locations
#     RandomLabeling : randomizes point types, keeping locations and weights unchanged
#     PopulationIndependence : keeps reference points unchanged, randomizes other point
locations
#   CaseControl : if TRUE, the case control version of M is computed.
#                 ReferenceType points are cases, NeighborType points are controls
# Value
```

```
#   A list:
#     Simulations: the matrix of simulated values
#     Min and Max: two vectors containing lower and upper bounds of the confidence
interval
################
MEnvelope <-
function(NumberOfSimulations, Alpha = .05, X, r, ReferenceType, NeighborType,
SimulationType = "RandomLocation", CaseControl=FALSE) {
  # Warning for erroneous configurations
  if (CaseControl & (SimulationType != "RandomLocation")) {
    warning(paste("The null hypothesis for case-control M should be RandomLocation. The
argument used is", SimulationType))
  }
  # Compute simulations
  Msims <- t(sapply(1:NumberOfSimulations, SimulateM, X, r, ReferenceType,
NeighborType, SimulationType, CaseControl))
  # Compute the local confidence envelope
  Envelope <- sapply(1:length(r), CriticalValues, Msims, Alpha)
  # Return the simulations and the envelope
  return(list(Simulations=Msims, Min=Envelope[1, ], Max=Envelope[2, ]))
}

################
# GoFtest
#   Returns a p-value for the test of a distace based measure of concentration against
the null hypothesis
# Arguments
#   ActualValues : the M (or other function) vector for all distances
#   SimulatedValues : the matrix of simulated M (or other function) values (one line
per simulation)
#   r : the vector of distances
# Value
#   A p-value
################
GoFtest <- function(ActualValues, SimulatedValues, r) {
  NumberOfSimulations <- dim(SimulatedValues)[1]
  AverageSimulatedValues <- apply(SimulatedValues, 2, sum)/(NumberOfSimulations-1)
  rIncrements <- (r-c(0,r)[1:length(r)])[2:length(r)]

  # Ui calculate the statistic for a simulation
  Ui <- function(SimulationNumber) {
    Departure <- (SimulatedValues[SimulationNumber, ]-
AverageSimulatedValues)[1:length(r)-1]
    WeightedDeparture <-
(Departure[!is.nan(Departure)])^2*rIncrements[!is.nan(Departure)]
    return(sum(WeightedDeparture))
  }

  # Calculate the Ui statistic for all simulations
  SimulatedU <- sapply(1:NumberOfSimulations, Ui)

  # Calculate the statistic for the actual value
  RecenteredValues <- (ActualValues-AverageSimulatedValues)[1:length(r)-1]
  WeightedRecenteredValues <-
(RecenteredValues[!is.nan(RecenteredValues)])^2*rIncrements[!is.nan(RecenteredValues)]
  ActualU <- sum(WeightedRecenteredValues)

  # Return the rank
  return(mean(ActualU<SimulatedU))
}

################
# PlotResults
#   Plots a function
# Arguments
#   r : the vector of distances
#   ActualValues : the M (or other function) vector for all distances
```

```
#   LocalCI : envelope. List of two vectors $Min and $Max
#   xlab, ylab : labels of axes.
#   ReferenceValue : the reference value of the function, represented by a horizontal
line
################
PlotResults <-
function(r, ActualValues, LocalCI, xlab="r", ylab="", ReferenceValue=NA) {
  AllValues <- unlist(c(ActualValues, LocalCI$Min, LocalCI$Max))
  plot(r, ActualValues, type="l", col="red", xlab=xlab, ylab=ylab,
ylim=c(min(AllValues, na.rm = TRUE), max(AllValues, na.rm = TRUE)))
  lines(r, LocalCI$Min, lty=3)
  lines(r, LocalCI$Max, lty=3)
  abline(h=ReferenceValue, lty=2)
}

################
# Randomizations of point patterns
#

# RandomLabeling.M randomizes point types, keeping locations and weights unchanged.
# rlabel(X) randomizes the marks, types and weights together.
RandomLabeling.M <- function(X) {
  RandomizedX <- rlabel(X)
  RandomizedX$marks$PointWeight <- X$marks$PointWeight
  return(RandomizedX)
}

# PopulationIndependence.M keeps reference points unchanged, randomizes other point
locations.
# rlabel randomizes marks (equivalently radomizes locations).
PopulationIndependence.M <- function(X, ReferenceType) {
  ReferencePP <- X[X$marks$PointType==ReferenceType]
  OtherPointsPP <- X[X$marks$PointType!=ReferenceType]
  RandomizedX <- superimpose(ReferencePP, rlabel(OtherPointsPP))
  return(RandomizedX)
}

# CriticalValues returns the quantiles (alpha, 1-alpha) of the column number rNumber in
Simulations matrix
# (columns are the values of the measure for each distance, lines are simulations)
CriticalValues <- function(rNumber, Simulations, Alpha) {
  return(quantile(Simulations[ , rNumber], probs=c(Alpha, 1-Alpha), na.rm=TRUE))
}


####################################
# Example
####################################

# Simulate a point pattern with two types
# Grey points are inhomogenous Poisson
Grey <- rpoispp(function(x,y) {2000 * exp(-3*x)}, 10000)
PointType   <- rep("Grey", Grey$n)
PointWeight <- runif(Grey$n, min=1, max=10)
Grey$marks <- data.frame(PointType, PointWeight)
# Black points are clumped (inhomogenous Matérn)
Black <- rMatClust(8, 0.05, function(x,y) {20 * exp(-3*x)})
PointType   <- rep("Black", Black$n)
PointWeight <- runif(Black$n, min=1, max=10)
Black$marks <- data.frame(PointType, PointWeight)
# Merge
X <- superimpose(Grey, Black)
plot(X)

# Calculate M
r <- seq(0, .5, .01)
ActualValues.X <- M.r(X, r, "Black", "Black")
```

```
# Calculate confidence envelope (takes time)
NumberOfSimulations <- 100
Alpha <- .05
LocalEnvelope.X <- MEnvelope(NumberOfSimulations, Alpha, X, r,  "Black", "Black")

# Plot
PlotResults(r, ActualValues.X, LocalEnvelope.X, ylab="M", ReferenceValue=1)

# pValue
GoFtest(ActualValues.X, LocalEnvelope.X$Simulations, r)
```