```
#############################################################################
# Examples of dynamics on spatial graphs with different metacommunity
perspectives
#
# By: Dominique Gravel (dominique_gravel@uqar.ca)
# April 2013
#
#############################################################################
################################
# General parameters
S = 100
n = 25
r = 0.3
nsteps = 100

########################################
########################################
# Patch dynamics example
source("spatial_graphs.R")
source("patch_model.R")
c = 0.2
e = 0.1
spatial_graph = geograph(n, r)
results = patch_model(c, e, S, spatial_graph, nsteps)

# Figure 1: time series
x11(height = 5.5, width = 6)
plot(c(1:nsteps),apply(results[[2]][,2:(S+1)],1,sum),xlab = "Time", ylab
= "Average local species richness",cex.lab = 1.5, cex.axis = 1.25, type =
"l", ylim = c(0,S))

# Figure 2: local species richness
s = apply(results[[1]],1,sum)
vec.col = numeric(length(s))
RK = rank(s)
for(i in 1:n) vec.col[i] = rainbow(n,start = 0, end = 0.7)[RK[i]]
plot_spatial(spatial_graph, vec.col)

########################################
########################################
# Neutral example
source("spatial_graphs.R")
source("lottery_model.R")
# Default parameters
m = 0.2
M = 0.01
k = 0.1
J = 100
sdN = Inf
sdE = 5
spatial_graph = geograph_fn(n,r)
results = lottery_model(m,M,k,S,J,sdE,sdN,spatial_graph,nsteps)

# Figure 1: time series
x11(height = 5.5, width = 6)
plot(c(1:nsteps),apply(results[[2]][,2:(S+1)],1,sum),xlab = "Time", ylab
= "Average local species richness",cex.lab = 1.5, cex.axis = 1.25, type =
"l", ylim = c(0,S))

# Figure 2: local species richness
```

```
s = apply(results[[1]],1,sum)
vec.col = numeric(length(s))
RK = rank(s)
for(i in 1:n) vec.col[i] = rainbow(n,start = 0, end = 0.7)[RK[i]]
plot_spatial(spatial_graph, vec.col)


########################################
########################################
# Species sorting example
source("spatial_graphs.R")
source("lottery_model.R")
# Default parameters
m = 0.2
M = 0.01
k = 0.1
J = 100
sdN = 15
sdE = 5
spatial_graph = geograph_fn(n,r)
results = lottery_model(m,M,k,S,J,sdE,sdN,spatial_graph,nsteps)

# Figure 1: time series
x11(height = 5.5, width = 6)
plot(c(1:nsteps),apply(results[[2]][,2:(S+1)],1,sum),xlab = "Time", ylab
= "Average local species richness",cex.lab = 1.5, cex.axis = 1.25, type =
"l", ylim = c(0,S))

# Figure 2: local species richness
s = apply(results[[1]],1,sum)
vec.col = numeric(length(s))
RK = rank(s)
for(i in 1:n) vec.col[i] = rainbow(n,start = 0, end = 0.7)[RK[i]]
plot_spatial(spatial_graph, vec.col)
```

```
#########################################################################
# Individual based lottery model with spatially explicit
# dispersal and environmental heterogeneity
#
# The model collapse to a neutral model when the niche breadth tends to
infinity
#
# Store the results in a time series of occupancy, a presence-absence
matrix
# and a local abundance matrix
#
# By: Dominique Gravel (dominique_gravel@uqar.ca)
# April 2013
#########################################################################

# Main function
lottery_model = function(m,M,d,S,J,sdE,sdN,spatial_graph,nsteps) {
      # Args:
      #     m: immigration probability from the neighbourhood
      #     M: immigration probability from outside the metacommunity
      #     k: local death rate
      #     S: number of species
      #     J: local community size
      #     sdE: within patch standard deviation of the environment
      #     sdN: niche breadth
      #     spatial_graph: a spatial graph object
      #     nsteps: number of time steps to run the simulation
      #
      # Returns:
      #     A list with a time series of occupancy for each species, a
site presence-absence matrix
      #     and a site-abundance matrix from the last time step
      #
      ########################################
      # Prepare the simulation

      ####################
      # Adjency matrix
      adjMat = spatial_graph[[2]] # The original matrix
      n = nrow(adjMat)                  # Number of nodes
      degrees = apply(adjMat,2,sum)     # Number of degrees for each node
      w = adjMat*matrix(degrees^-1,nr = n,nc=n, byrow=T) # Weighted
adjancy matrix

      ####################
      # Local environmental conditions
      Env = matrix(nr=n, nc = J)
      for(i in 1:n) Env[i,] = rnorm(J,mean = runif(1,0,100), sd = sdE)
      # For a random distribution of environments
#     for(i in 1:n) Env[i,] = rnorm(J,mean = i/n*100, sd = 0)
      # For a uniform distribution of environments
      ####################
      # Niche optimums
      u = runif(S, 0, 100)   # For a random distribution of optimums
#     u = 100*c(1:S)/S       # For a uniform distribution of optimums

      ####################
      # Initialization of the metacommunity
      # Starts with uniform abundance at each location
      localC = list()
```

```
      P = matrix(0,nr = n, nc = S)
      for(i in 1:n) {
            localC[[i]] = t(rmultinom(n = J,size = 1, prob =
numeric(S)+J^-1))
            P[i,] = apply(localC[[i]],2,sum)/J
      }

      ####################
      # Matrix in which we record the occupancy over time
      Series = matrix(nr=nsteps,nc=S+1)

      ####################
      # Loop over all time steps
      for(time in 1:nsteps) {

            # Calculate the weighted relative abundance in the neighboring
communities
            wP = w%*%P/apply(w,1,sum)

            ####################
            # Loop across the patches
            for(i in 1:n) {
                  # Kill individuals at random
                  rand = runif(J,0,1)
                  localC[[i]][rand < k,] = 0

                  ####################
                  # Calculate recruitment probability
                  # Relative abundance in the seed rain
                  rel_seed = M*S^-1 + m*wP[i,] + (1-m-
M)*apply(localC[[i]],2,sum)/sum(localC[[i]])

                  # Weighting by local environmental conditions
                  surv = exp(-(matrix(Env[i,],nr = J, nc = S, byrow = F) -
matrix(u,nr = J, nc = S, byrow = T))^2/2/sdN^2)
                  recruitProb = surv*matrix(rel_seed,nr = J,nc = S,byrow =
T)/apply(surv*matrix(rel_seed,nr = J,nc = S,byrow = T),1,sum)

                  ####################
                  # Replace dead individuals
                  recruit = t(apply(recruitProb,1,recruit_fn))
                  localC[[i]][rand<d,] = recruit[rand<d,]

                  # Record local relative abundance
                  P[i,] = apply(localC[[i]],2,sum)/sum(localC[[i]])
            }

      ####################
      # Transform abundance in presence/absence
      pres = matrix(0,nr = n, nc = S)
      pres[P>0] = 1
      occ = apply(pres,2,mean)

      # Record occupancy
      Series[time,] = c(time,occ)
      }

      # Output
      return(list(pres,Series,P))
}
```

```
# Convenient function used in the simulation
recruit_fn = function(prob) rmultinom(n = 1, size = 1, prob = prob)
```

```
#########################################################################
# Multi-species discrite-time Levins metapopulation model
# Spatially explicit local dispersal
# Spatially uniform environment
#
# Store the results in a time series of occupancy
# and a presence absence matrix
#
# By: Dominique Gravel (dominique_gravel@uqar.ca)
# April 2013
#########################################################################

patch_model = function(c, e, S, spatial_graph, nsteps) {
      # Runs a muti-species Levins model on a spatial graph
      #
      # Args:
      #     c: colonization probability
      #     e: extinction probability
      #     S: number of species
      #     spatial_graph: a spatial graph object
      #     nsteps: number of time steps to run the simulation
      #
      # Returns:
      #     A list with a time series of occupancy for each species and a
presence-absence
      #     matrix from the last time step
      #
      ########################################
      # Prepare the simulation

      ####################
      # Adjency matrix
      adjMat = spatial_graph[[2]] # The original matrix
      n = nrow(adjMat) # Number of nodes
      degrees = apply(adjMat,2,sum) # Number of degrees for each node
      ColProb =  c*adjMat*matrix(degrees^-1,nr = n,nc=n, byrow=T) #
Colonization probability weighted by the number of out degrees

      ####################
      # Initialization of the metacommunity in the presence-absence
matrix pres
      # Global distribution at the beginning of the run
      pres = matrix(1, nr = n, nc = S)

      ####################
      # Matrix in which we record the occupancy over time
      Series = matrix(nr=nsteps,nc=S+1)

      ########################################
      # Loop over all time steps
      for(time in 1:nsteps) {

            ####################
            # Test if there is extinction
            ExtMat = matrix(0,nr=n,nc=S)
            randExt = matrix(runif(n*S,0,1),nr=n,nc=S)
            ExtMat[pres == 1 & randExt < e] = -1

            ####################
            # Test if there is colonization
```

```
            ColMat = matrix(0,nr=n,nc=S)
            randCol = matrix(runif(n*S,0,1),nr=n,nc=S)

            # Calculate how many populations are connected to the focal
patch
            ConPop = ColProb%*%pres

            # Perform the test
            ColMat[pres == 0 & randCol < ConPop] = 1

            ####################
            # Apply changes in presence-absence
            pres = pres + ExtMat + ColMat

            # Record occupancy
            Series[time,] = c(time, apply(pres,2,sum)/n)

      }

      return(list(pres,Series))
}
```

```
#########################################################################
# Source code for the figures 1-4 in the paper
# Gravel, Poisot and Desjardins. 2013. Using neutral theory to reveal the
contribution of dispersal to community assembly in complex landscapes. J.
Limnology.
#
# By: Dominique Gravel (dominique_gravel@uqar.ca)
# April 2013
#
#########################################################################
source("spatial_graphs.R")
source("patch_model.R")
source("lottery_model.R")
library(vegan)

###############################
# Useful functions
alpha_fn = function(pres) {
      s = apply(pres,1,sum)
      vec.col = numeric(length(s))
      RK = rank(s)
      for(i in 1:n) vec.col[i] = rainbow(n,start = 0, end = 0.7)[RK[i]]
      return(vec.col)
}

plot_alpha = function(spatial_graph, vec.col) {
      # Plots a spatial graph
      #
      # Args:
      #   spatia_graph: the output of one of the spatial graphs
      #
      par(mar=c(1,1,1,1))
      XY = spatial_graph[[1]]
      adjMat = spatial_graph[[2]]

      plot(XY[,1],XY[,2],xlab = "", ylab = "",cex = 1.5,labels = F)
      adjVec = stack(as.data.frame(adjMat))[,1]
      XX = expand.grid(XY[,1],XY[,1])
      YY = expand.grid(XY[,2],XY[,2])
      XX = subset(XX,adjVec==1)
      YY = subset(YY,adjVec==1)
      arrows(x0 = XX[,1],x1=XX[,2],y0 = YY[,1], y1 = YY[,2], length =
0,lwd = 0.2, col = "darkgrey")
      points(XY[,1],XY[,2],pch=21,bg=vec.col,cex = 1.5)
}

###############################
# General parameters
S = 100
n = 25
r = 0.3

###############################
# Draw the four landscapes
connected = connected_fn(n)
lattice = lattice_fn(n)
geograph = geograph_fn(n,r)
geotree = geotree_fn(n,r)

# Geographic distances among plots
```

```
distConnected = as.matrix(dist(connected[[1]], upper = T, diag = T))
distLattice = as.matrix(dist(lattice[[1]], upper = T, diag = T))
distGeograph = as.matrix(dist(geograph[[1]], upper = T, diag = T))
distGeotree = as.matrix(dist(geotree[[1]], upper = T, diag = T))

# Topologic distances among plots
topoLattice = spm(lattice[[2]])
topoGeograph = spm(geograph[[2]])
topoGeotree = spm(geotree[[2]])

# Degree centrality
degGeograph = deg_cen(geograph)
degGeotree = deg_cen(geotree)

# Eigen centrality
eigGeograph = eig_cen(geograph)
eigGeotree = eig_cen(geotree)

# Closeness centrality
clsGeograph = cls_cen(geograph)
clsGeotree = cls_cen(geotree)

################################
# Run the three models with the four landscapes

###############
# Patch dynamics
# Specific parameters
c = 0.4
e = 0.1
nsteps = 1000

# Simulations:
patchConnected = patch_model(c, e, S, connected, nsteps)
patchLattice = patch_model(c, e, S, lattice, nsteps)
patchGeograph = patch_model(c, e, S, geograph, nsteps)
patchGeotree = patch_model(c, e, S, geotree, nsteps)

# Alpha diversity
alphaPatchConnected = alpha_fn(patchConnected[[1]])
alphaPatchLattice = alpha_fn(patchLattice[[1]])
alphaPatchGeograph = alpha_fn(patchGeograph[[1]])
alphaPatchGeotree = alpha_fn(patchGeotree[[1]])

# Beta diversity
betaPatchConnected = as.matrix(vegdist(patchConnected[[1]],method =
"bray",diag=T,upper=T))
betaPatchLattice = as.matrix(vegdist(patchLattice[[1]],method =
"bray",diag=T,upper=T))
betaPatchGeograph = as.matrix(vegdist(patchGeograph[[1]],method =
"bray",diag=T,upper=T))
betaPatchGeotree = as.matrix(vegdist(patchGeotree[[1]],method =
"bray",diag=T,upper=T))

###############
# Neutral dynamics
# Specific parameters
m = 0.2
M = 0.01
k = 0.1
```

```
J = 100
sdN = Inf
sdE = 5
nsteps = 1000

# Simulations
neutralConnected = lottery_model(m,M,k,S,J,sdE,sdN,connected,nsteps)
neutralLattice = lottery_model(m,M,k,S,J,sdE,sdN,lattice,nsteps)
neutralGeograph = lottery_model(m,M,k,S,J,sdE,sdN,geograph,nsteps)
neutralGeotree = lottery_model(m,M,k,S,J,sdE,sdN,geotree,nsteps)

# Alpha diversity
alphaNeutralConnected = alpha_fn(neutralConnected[[1]])
alphaNeutralLattice = alpha_fn(neutralLattice[[1]])
alphaNeutralGeograph = alpha_fn(neutralGeograph[[1]])
alphaNeutralGeotree = alpha_fn(neutralGeotree[[1]])

# Beta diversity
betaNeutralConnected = as.matrix(vegdist(neutralConnected[[1]],method =
"bray",diag=T,upper=T))
betaNeutralLattice = as.matrix(vegdist(neutralLattice[[1]],method =
"bray",diag=T,upper=T))
betaNeutralGeograph = as.matrix(vegdist(neutralGeograph[[1]],method =
"bray",diag=T,upper=T))
betaNeutralGeotree = as.matrix(vegdist(neutralGeotree[[1]],method =
"bray",diag=T,upper=T))

###############
# Species sorting dynamics
# Specific parameters
m = 0.2
M = 0.01
k = 0.1
J = 100
sdN = 15
sdE = 5
nsteps = 1000

# Simulations
ssConnected = lottery_model(m,M,k,S,J,sdE,sdN,connected,nsteps)
ssLattice = lottery_model(m,M,k,S,J,sdE,sdN,lattice,nsteps)
ssGeograph = lottery_model(m,M,k,S,J,sdE,sdN,geograph,nsteps)
ssGeotree = lottery_model(m,M,k,S,J,sdE,sdN,geotree,nsteps)

# Alpha diversity
alphaSSConnected = alpha_fn(ssConnected[[1]])
alphaSSLattice = alpha_fn(ssLattice[[1]])
alphaSSGeograph = alpha_fn(ssGeograph[[1]])
alphaSSGeotree = alpha_fn(ssGeotree[[1]])

# Beta diversity
betaSSConnected = as.matrix(vegdist(ssConnected[[1]],method =
"bray",diag=T,upper=T))
betaSSLattice = as.matrix(vegdist(ssLattice[[1]],method =
"bray",diag=T,upper=T))
betaSSGeograph = as.matrix(vegdist(ssGeograph[[1]],method =
"bray",diag=T,upper=T))
betaSSGeotree = as.matrix(vegdist(ssGeotree[[1]],method =
"bray",diag=T,upper=T))
```

```
################################
# Figure 1
quartz(height = 6.5, width = 6.5/4)
t = matrix(c(1:4),nr = 4, nc = 1, byrow = T)
layout(t)
layout.show(4)

#plot_alpha(connected,alphaPatchConnected)
plot_alpha(connected,alphaNeutralConnected)
#plot_alpha(connected,alphaSSConnected)

#plot_alpha(lattice,alphaPatchLattice)
plot_alpha(lattice,alphaNeutralLattice)
#plot_alpha(lattice,alphaSSLattice)

#plot_alpha(geograph,alphaPatchGeograph)
plot_alpha(geograph,alphaNeutralGeograph)
#plot_alpha(geograph,alphaSSGeograph)

#plot_alpha(geotree,alphaPatchGeotree)
plot_alpha(geotree,alphaNeutralGeotree)
#plot_alpha(geotree,alphaSSGeotree)

dev.copy2pdf(file = "Networks.pdf")

################################
# Figure 2
quartz(height = 6, width = 6)
t = matrix(c(1:4),nr = 2, nc = 2, byrow = T)
layout(t)
layout.show(4)

apply(patchGeograph[[1]],2,sum)

maxPatchGeograph = max(apply(patchGeograph[[1]],1,sum))
maxNeutralGeograph = max(apply(neutralGeograph[[1]],1,sum))
maxSSGeograph = max(apply(ssGeograph[[1]],1,sum))

maxPatchGeotree = max(apply(patchGeotree[[1]],1,sum))
maxNeutralGeotree = max(apply(neutralGeotree[[1]],1,sum))
maxSSGeotree = max(apply(ssGeotree[[1]],1,sum))

par(mar=c(3,6,4,1))
plot(degGeograph,apply(patchGeograph[[1]],1,sum)/maxPatchGeograph,pch =
19, xlab = "", ylab = "Species richness
(scaled)", ylim = c(0,1),cex.lab = 1.5, cex.axis = 1.25)
points(degGeograph,apply(neutralGeograph[[1]],1,sum)/maxNeutralGeograph,p
ch = 21, bg = "red")
points(degGeograph,apply(ssGeograph[[1]],1,sum)/maxSSGeograph,pch = 21,
bg = "blue")

par(mar=c(3,4,4,3))
plot(eigGeograph,apply(patchGeograph[[1]],1,sum)/maxPatchGeograph,pch =
19, xlab = "", ylab = "", ylim = c(0,1),cex.lab = 1.5, cex.axis = 1.25)
points(eigGeograph,apply(neutralGeograph[[1]],1,sum)/maxNeutralGeograph,p
ch = 21, bg = "red")
points(eigGeograph,apply(ssGeotree[[1]],1,sum)/maxSSGeograph,pch = 21, bg
= "blue")

par(mar=c(6,6,1,1))
```

```
plot(degGeotree,apply(patchGeotree[[1]],1,sum)/maxPatchGeotree,pch = 19,
xlab = "Degree centrality", ylab = "Species richness
(scaled)", ylim = c(0,1),cex.lab = 1.5, cex.axis = 1.25)
points(degGeotree,apply(neutralGeotree[[1]],1,sum)/maxNeutralGeotree,pch
= 21, bg = "red")
points(degGeotree,apply(ssGeotree[[1]],1,sum)/maxSSGeotree,pch = 21, bg =
"blue")

par(mar=c(6,4,1,3))
plot(eigGeotree,apply(patchGeotree[[1]],1,sum)/maxPatchGeotree,pch = 19,
xlab = "Eigen centrality", ylab = "", ylim = c(0,1),cex.lab = 1.5,
cex.axis = 1.25)
points(eigGeotree,apply(neutralGeotree[[1]],1,sum)/maxNeutralGeotree,pch
= 21, bg = "red")
points(eigGeotree,apply(ssGeotree[[1]],1,sum)/maxSSGeotree,pch = 21, bg =
"blue")

dev.copy2pdf(file = "Centrality.pdf")

################################
# Figure 3

quartz(height = 4.5, width = 6)
t = matrix(c(1:12),nr = 3, nc = 3, byrow = T)
layout(t)
layout.show(9)

par(mar=c(1,1,1,1))
plot(distLattice,betaPatchLattice,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(distGeograph,betaPatchGeograph,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(distGeotree,betaPatchGeotree,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)

plot(distLattice,betaNeutralLattice,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(distGeograph,betaNeutralGeograph,pch = 19,labels = F,xlab = "",ylab
= "",cex = 0.8)
plot(distGeotree,betaNeutralGeotree,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)

plot(distLattice,betaSSLattice,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(distGeograph,betaSSGeograph,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(distGeotree,betaSSGeotree,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)

dev.copy2pdf(file = "BetaGeoDist.pdf")

################################
# Figure 4

quartz(height = 4.5, width = 6)
t = matrix(c(1:12),nr = 3, nc = 3, byrow = T)
layout(t)
layout.show(9)

par(mar=c(1,1,1,1))
```

```
plot(topoLattice,betaPatchLattice,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(topoGeograph,betaPatchGeograph,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(topoGeotree,betaPatchGeotree,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)

plot(topoLattice,betaNeutralLattice,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(topoGeograph,betaNeutralGeograph,pch = 19,labels = F,xlab = "",ylab
= "",cex = 0.8)
plot(topoGeotree,betaNeutralGeotree,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)

plot(topoLattice,betaSSLattice,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(topoGeograph,betaSSGeograph,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)
plot(topoGeotree,betaSSGeotree,pch = 19,labels = F,xlab = "",ylab =
"",cex = 0.8)

dev.copy2pdf(file = "BetaTopoDist.pdf")
```

```
#############################################################################
# Functions to generate networks ("graphs") for spatial ecology.
#
# Stores the graph in a adjacency matrix
#
# Functions to generate graphs:
# - geograph_fn: returns a random geometric graph.
# - geotree_fn: returns a random geometric tree (of doom!).
# - lattice_fn: returns a lattice type of spatial graph.
# - connected_fn: returns a fully connected graph.
#
# Useful functions:
# - plot_spatial: custom function to plot spatial graphs.
# - deg_cen: computes degree centrality.
# - cls_cen: computes closeness centrality.
# - eig_cen: computers eigen-centrality.
#
# Associated functions:
# - connected: tests if the graph is fully connected.
# - test_con: recursive function used by 'connected'.
# - spm: Generates the shortest path distance matrix.
# - spd: Generates the shortest path distances starting with a given
source.
# - spt: Generates the shortest path tree as an adjacency matrix.
#
# Examples:
# > plot_spatial(geograph_fn(n = 25, r = 0.3))
# > plot_spatial(geotree_fn(n = 25, r = 0.3))
# > plot_spatial(connected_fn(n = 25))
# > plot_spatial(lattice_fn(n = 25))
#
# By: Dominique Gravel (dominique_gravel@uqar.ca)
#     Philippe Desjardins-Proulx (philippe.d.proulx@gmail.com)
#
# May 2013
#############################################################################

library(igraph)

geograph_fn = function(n = 64, r = 0.32) {
  # Generates a random geometric graph.
  #
  # Args:
  #   n: Number of vertices.
  #   r: Threshold distance to connect vertices.
  #
  # Returns:
  #   A list with the xy coordinates and the adjacency matrix.
  # Note: the algorithm tests if the graph is connected
  repeat {
    xy = cbind(runif(n), runif(n))
    distMat = as.matrix(dist(xy, method = 'euclidean', upper = T, diag =
T))
    adjMat = matrix(0, nr = n, nc = n)
    adjMat[distMat < r] = 1
    diag(adjMat) = 0
    if (isconnected(adjMat)) {
      return(list(xy, adjMat))
    }
  }
```

```
}

##############################
geotree_fn = function(n = 64, r = 0.32) {
  # Generates a random geometric tree.
  #
  # Args:
  #   n: Number of vertices.
  #   r: Threshold distance to connect vertices.
  #
  # Returns:
  #   A list with the xy coordinatesa nd the adjacency matrix
  source = sample(1:n, 1)
  g = geograph_fn(n, r)
  return(list(g[[1]], spt(g[[2]], source)))
}

##############################
lattice_fn = function(n) {
  # Generates a lattice type of graph
  #
  # Args:
  #   n: number of cells in the lattice
  #
  # Returns:
  #   A list with the xy coordinates and the adjacency matrix
  ns = sqrt(n)
  X = seq(0,1,by = 1/(ns-1))
  Y = seq(0,1,by = 1/(ns-1))
  XY = expand.grid(X,Y)
  distMat = as.matrix(dist(XY,method = "euclidean", upper = T, diag = T))
  adjMat = matrix(0, nr=n, nc=n)
  adjMat[distMat <= 1/(ns-1)*(1+1e-10)] = 1
  diag(adjMat) = 0
  return(list(XY,adjMat))
}

##############################
connected_fn = function(n) {
  # Generates a connected graph
  #
  # Args:
  #   n: number of cells in the graph
  #
  # Returns:
  #   A list with the xy coordinates and the adjacency matrix
  X = sin(c(0:(n-1))*2*pi/(n-1))
  Y = cos(c(0:(n-1))*2*pi/(n-1))
  adjMat = matrix(1,nr = n, nc = n)
  diag(adjMat) = 0
  XY = cbind(X,Y)
  return(list(XY,adjMat))
}

##############################
plot_spatial = function(spatial_graph, vec.col) {
  # Plots a spatial graph
  #
  # Args:
  #   spatia_graph: the output of one of the spatial graphs
```

```
  #
  x11(height = 5.5, width = 6)
  par(mar=c(5,6,2,1))
  XY = spatial_graph[[1]]
  adjMat = spatial_graph[[2]]
  plot(XY[,1],XY[,2],xlab = "X", ylab = "Y",cex.lab = 1.5, cex.axis =
1.25)
  adjVec = stack(as.data.frame(adjMat))[,1]
  XX = expand.grid(XY[,1],XY[,1])
  YY = expand.grid(XY[,2],XY[,2])
  XX = subset(XX,adjVec==1)
  YY = subset(YY,adjVec==1)
  arrows(x0 = XX[,1],x1=XX[,2],y0 = YY[,1], y1 = YY[,2], length = 0,lwd =
0.1, col = "grey")
  points(XY[,1],XY[,2],pch=21,bg=vec.col)
}

################################
deg_cen = function(g) {
  # Computes degree centrality.
  #
  # Args:
  #   g: A graph object as generated by geograph_fn, geotree_fn, etc...
  #
  # Returns:
  #   A vector with the degree centrality of each vectex.
  return(rowSums(g[[2]]))
}

################################
eig_cen = function(g) {
  # Computes eigen centrality.
  #
  # Args:
  #   g: A graph object as generated by geograph_fn, geotree_fn, etc...
  #
  # Returns:
  #   A vector with the eigen centrality of each vertex.
  ig = graph.adjacency(g[[2]], mode = 'undirected', weighted = NULL, diag
= F)
  return(evcent(ig)[[1]])
}

################################
cls_cen = function(g) {
  # Computes closeness centrality, the average distance between a
  # vertex and all other vertices (compute with the shortest path
  # algorithm).
  #
  # Args:
  #   g: A graph object as generated by geograph_fn, geotree_fn, etc...
  #
  # Returns:
  #   A vector with the closeness centrality of each vectex.
  ig = graph.adjacency(g[[2]], mode = 'undirected', weighted = NULL, diag
= F)
  return(closeness(ig))
}

################################
```

```
isconnected = function(adjMat) {
  # Tests if the graph is fully connected (i.e.: there is a path between
all nodes).
  #
  # Args:
  #   adjMat: An adjacency matrix (made of boolean values).
  #
  # Returns:
  #   TRUE if the graph is connected, FALSE otherwise.
  diag(adjMat) = 0
  n = nrow(adjMat)
  for (v in 1:n) {
    in_path = vector('logical', n)
    in_path[v] = T
    in_path = test_con(adjMat, in_path, v)
    if (sum(in_path == T) < n) {
      return(F)
    }
  }
  return(T)
}

################################
test_con = function(adjMat, in_path, v) {
  # Helper recursive function for 'isConnected'.
  for (i in 1:nrow(adjMat)) {
    if (adjMat[v, i] && in_path[i] == F) {
      in_path[i] = T
      in_path = test_con(adjMat, in_path, i)
    }
  }
  return(in_path)
}

################################
spm = function(adjMat) {
  # Generates the shortest path distance matrix for unweighted graphs
using
  # the Dijkstra algorithm.
  #
  # Args:
  #   adjMat: The adjacency matrix for the unweighted graph (should be
  #           filled with 0s and 1s)
  #
  # Returns:
  #   A matrix with the distance between all pair of vertices.
  m = as.matrix(spd(adjMat, 1)[[2]])
  for (i in 2:nrow(adjMat)) m = cbind(m, spd(adjMat, i)[[2]])
  return(m)
}

################################
spd = function(adjMat, source) {
  # Generates the shortest path distances starting with a given source.
  #
  # Args:
  #   adjMat: The adjacency matrix for the graph.
  #   source: The starting vertex for the algorithm.
  #
  # Returns:
```

```
  #   A vector with the shortest path distance.
  n = nrow(adjMat)
  distance = rep(Inf, n)
  distance[source] = 0
  previous = rep(0, n)
  q = 1:n

  while (length(q) != 0) {
    u = q[1]
    for (i in q) {
      if (distance[i] < distance[u]) {
        u = i
      }
    }
    q = setdiff(q, u)
    for (i in q) {
      x = if (adjMat[u, i] == 1) distance[u] + 1 else Inf
      if (x < distance[i]) {
        distance[i] = x
        previous[i] = u
      }
    }
  }
  return(list(previous, distance))
}

################################
spt = function(adjMat, source) {
  # Generates the shortest path tree as an adjacency matrix using
  # the shortest path distance function 'spd'.
  #
  # Args:
  #   adjMat: The adjacency matrix for the graph.
  #   source: The starting vertex for the algorithm.
  #
  # Returns:
  #   The shortest path tree as an adjacency matrix.
  prev = spd(adjMat, source)[[1]]
  m = matrix(0, nr = n, nc = n)
  for (i in 1:n) {
    m[i, prev[i]] = 1
    m[prev[i], i] = 1
  }
  return(m)
}
```