Supplementary Code:

R code described in the text: (1) Code to generate null models from real species distributions.

```
# Define some initial functions that are called by the main
# function:

calcCondProbs <- function (pa, neighLists = neighbourLists) {
### This function calculates the conditional probability
### of presence within a cell, given presence in neighboring
### cells. Its arguments are pa, a vector indicating presence or
### absence within the each square (values 0 or 1) and neighLists,
### a list of lists defining adjacency structure within the map.
### neighLists should have one list for each square of the map in
### the same order as pa, each list containing the indices of all
### squares with centre falling within a specified distance
### class of the centre of the focal square. For all the analyses
### presented here, 10 different distance classes were selected:
### 0 – 95km, 96 – 145km, 146 – 196km, etc. In this case
### neighLists is 10 items long, with each item a list the length
### of pa. The function is used in the calculate of the structural
### match between a simulated distribution and the real
### distribution.

    pOnly    <- pa == 1
    degree   <- length (neighLists)
    condProb <- numeric (degree)
    for (i in 1:degree) {
        condProb[i] <- mean(pa[unlist(neighLists[[i]][pOnly])])
    }
    condProb
}

#####################################################################

calcHists <- function (pa, neighLists = neighbourLists) {
### This function calculates binned probabilities of presence
### in the different distance classes for a vector of presence
### and absence, pa. It is used in the main function to
### identify candidate squares for swapping. Arguments are as
### above.

    nnlist   <- lapply(neighbourLists, function(lst) {
                    mapply(length, lst)})   # Count neighbours
                                            # within distance
                                            # classes
    pOnly    <- pa == 1    # Identify cells with presence
    degree   <- length (neighLists)  # calculate number of
                                     # distance bins.
    presSums <- vector("list", degree)
```

```
    probs    <- presSums

    # For each distance bin…
    for (i in 1: degree) {
        # identify squares with presence:
        nnlist[[i]]   <- nnlist[[i]][pOnly]
        # count squares with presence:
        presSums[[i]] <- mapply(function(vec) sum(pa[vec]),
                                neighLists[[i]][pOnly])
        # Calculate mean probability of presence within each
        # distance class and generate bins of the frequency of
        # each probability:
        probs[[i]]    <- hist(presSums[[i]] / nnlist[[i]],
                              breaks = seq(0, 1, length = 9),
                              plot = FALSE)
    }
    probs
}

#########################################################################

calcHistErrs <- function (actual = realHists, sim = sim1) {
### This function calculated the root mean squared error of the
### binned probabilities of presence between the real and
### simulated distribution. It is used in the main function to
### identify candidate squares for swapping. Its arguments are
### actual: the output of calcHists for the real distribution; and
### sim: the vector of presence/absence in the simulated
### distribution.

    degree    <- length(realHists)  # Count distance categories
    simHists  <- calcHists(sim)     # Calculate bins for
                                    # simulation.
    histDiffs <- vector("list", length = degree)
    for (i in 1: degree) {
        # For each distance category calculate the difference
        # between the real and simulated bins:
        histDiffs[[i]] <- actual[[i]][[2]] - simHists[[i]][[2]]
    }
    # calculate root mean square error:
    histErrs  <- sum( mapply (function(X) { sqrt(mean((X)^2)) },
                 histDiffs))
}


#########################################################################

calcCondProbErrs <- function (actualCPs = realProbs, sim = sim1) {
### This function calculates the difference between the
### conditional probabilities of the real and simulated
### distributions. Its arguments are actualCPs: the output of
```

```
### calcConProbs for the real distribution, and sim, the simulated
### distribution.

    # Calculate the conditional probabilities for the simulated
    # distribution:
    simProbs      <- calcCondProbs(sim)
    # Calculate the error:
    condProbDiffs <- actualCPs - simProbs
    condProbErr   <- sqrt(mean((condProbDiffs*1000)^2))
    # If the match is perfect, set error to 0:
    if (is.na(condProbErr)) condProbErr <- 0
    condProbErr
}

#########################################################################

calcAllSums <- function (pa, neighLists = neighbourLists) {
### This function calculates one part of the earlier function:
### the number of squares with presence within each distance
### category. Its arguments are as for calcCondProbs.

    nnlist   <- lapply(neighbourLists, function(lst) {
                    mapply(length, lst)})   # Count neighbours
                                            # within distance
                                            # classes
    degree <- length (neighLists)
    presSums <- vector("list", degree)
    for (i in 1: degree) {
         # count squares with presence:
         presSums[[i]] <- mapply(function(vec) sum(pa[vec]),
                              neighLists[[i]][pOnly])
    }
    presSums
}

#########################################################################

calcOneSum <- function (pa, neighLists =
                        neighbourLists[[mostWrongDegree]]) {
### This function calculates the number of neighbours with
### presence in the required distance category, but for only one
### distance category, defined by one element of neighLists as
### before.

    presSums <- mapply(function(vec) sum(pa[vec]), neighLists)
}

#########################################################################

calcSimErrs <- function (realSimBins = realSimBinned, sim = sim1,
                         bins = binVec, plot.it = FALSE, dists =
```

```
                        seq(0, 2900, length = 16)) {
### This function calculates the error between two simulated
### distributions based on semivariograms. Its arguments are:
###   realSimBins: a data.frame containing the sum of the
###     absolute difference between all pairs of squares
###     aggregated by the distance classes defined in dists with
###     column 1 defining the distance, column 2 the sum, and
###     column 3 the count of pairs within each distance class.
###     Each row represents one distance class.
###   sim: a vector of 0 and 1 indicating presence/absence in the
###     simulation.
###   bins: a vector identifying the distance classes (as in
###     dists) for all pairs of squares in the distribution.
###   plot.it: logical
###   dists: a vector of 16 distances identifying the bins

    # the absolute difference between all pairs of squares:
    simSimVec <- matrix(abs(sim[rep(1:nlocs, times = nlocs)] –
                 sim[rep(1: nlocs, each = nlocs)]), ncol = nlocs)
    simSimVec <- simSimVec[col(simSimVec) < row(simSimVec)]
    #  aggregate these differences by the binned distances:
    sum1sim1  <- aggregate(c(simSimVec), by = list(bins), sum)
    # calculate the mean absolute difference for real and
    # simulated distributions in each bin:
    realMeans <- realSimBins[1:15,2] / realSimBins[1:15,3]
    simMeans  <- sum1sim1[1:15,2] / realSimBins[1:15,3]

    # If required, plot the variogram:
    if (plot.it) {
        # Calculate confidence intervals:
        realCIs <- binomCI(realMeans, totals[1:15])
        simCIs  <- binomCI(simMeans, totals[1:15])
        # Plot the real distribution variogram:
        plot (dists, realMeans, ylim =  c(0, max(sum1sim1[,2] /
                  totals[1:length(realSimBins[,2])],
                  realSimBins[,2] /totals[1 : length(
                  realSimBins[,2])])), xlab = "distance bin",
                  ylab = "mean difference", pch = 20, cex = 2,
                  main = "Binned Variogram")
        # Add confidence intervals:
        for (i in 1:15) {
            lines(rep(dists[i],2), c(realCIs[[1]][i],
            realCIs[[2]][i]))
        }
        # Add the simulated values and confidence intervals in
        # red:
        points(dists, simMeans, col = "red", pch = 20, cex = 2)
        for (i in 1:15) {
            lines(c(dists[i],dists[i]), c(simCIs[[1]][i],
                simCIs[[2]][i]), col = "red")
        }
```

```
    }
    # Return the root mean square error:
    err1 <- sqrt(mean(((realMeans - simMeans)*1000)^2))
}

#########################################################################

binomCI <- function (p, n = nlocs) {
### Calculates normal approximation to binomial confidence
### intervals (used only for plots).
    upper <- p + 1.96 * sqrt((p*(1-p))/n)
    lower <- p - 1.96 * sqrt((p*(1-p))/n)
    return (list( upper, lower))
}

#########################################################################

initMap <- function(pa, neighLists = neighbourLists) {
### This function generates an initial map with the required
### number of presences and absences and conditional probability
### of presence in first order neighbours that approximates that
### of the real map. This map is later refined by iteration. Its
### arguments are identical to those of calcCondProbs.

    neighbours <- neighLists[[1]] # Identify first order
                                  # neighbours
    # Identify neighbours in the middle distance category:
    distNeighs <- neighLists[[round(length(neighLists)/2)]]
    nPres      <- sum (pa, na.rm = T)  # count prevalence

    # Calculate conditional probability within the first
    # distance class:
    condProb   <- calcCondProbs(pa)[1]
    # select a focal square at random:
    startloc   <- sample(1:nlocs, 1)

    # Set up some parameters:
    nNeighs    <- 0                # Number of neighbours
    simPres    <- 0                # pa in simulation
    simDist    <- numeric(nlocs)   # the simulated distribution
    tested     <- numeric(nlocs)   # whether the square has been
                                   # tested before

    while (simPres < nPres) {
        # While prevalence in the simulated distribution is
        # less than that of the real distribution:

        # count the number of untested neighbours of the focal
        # square:
        nNeighs <- sum(tested[neighbours[[startloc]]] == 0)
        looplim <- 0 # define a counter for the loop
```

```
# while choosing a square with no untested neighbours:
while (nNeighs == 0) {

    looplim <- looplim + 1 # increment counter
    # if the loop is stuck, chose a new focal square:
    if (looplim > 300) startloc <- sample (1:nlocs,1)
    # select a distant neighbour of the focal square:
    startloc <- sample(distNeighs[[startloc]], 1)
    # Set the simulated distribution at the focal square
    # to 1:
    simDist[startloc] <- 1
    # Identify the focal square as tested:
    tested[startloc] <- 1
    # count the number of untested neighbours of the
    # focal square:
    nNeighs <- sum(tested[neighbours[[startloc]]] == 0)
}

# identify the untested neighbours of the focal square:
neighs <- neighbours[[startloc]][tested[
            neighbours[[startloc]]] == 0]
# Define presence / absence in the neighbours of the focal
# square based on the conditional probability:
simDist[neighs] <- sample (0:1, nNeighs, replace = T,
                           prob = c(1-condProb, condProb))
# Identify test locations as tested:
tested[neighs] <- 1

# If focal square has neighbours with presence, select one
# of these squares as the new focal square:
if (sum (simDist[neighs]) > 0) {
    if (sum (simDist[neighs]) == 1) {
        startloc <- neighs[simDist[neighs] == 1]
    } else {
        startloc <- sample (neighs[simDist[neighs] ==
                            1], 1)
    }
# Otherwise select any neighbour:
} else {
    if (length (neighs) == 1) {
        startloc <- neighs
    } else {
        startloc <- sample (neighs, 1)
    }
}

simPres <- sum(simDist)    # Count number of presences
# if > 90% of squares have already been tested, reset
# tested to 0 and continue:
if (sum (tested) > nPres * 0.9) tested <- numeric(nlocs)
```

```
    }

    # Check that there are not too many simulated presences:
    tooMany <- simPres - nPres
    if (tooMany != 0) {
        # if there are any, set any extra presences to 0:
        simDist[sample(which(simDist == 1), tooMany)] <- 0
    }
    return (simDist)
}

#####################################################################

plotFinal <- function(actual = pa, simP = bestSim, realSimBin =
              realSimBinned, binVecP = binVec, realCP, coords) {
### This function plots a real distribution, a final simulation
### and some summaries of the match between the two. Its arguments
### are:
###    actual: a vector of presence / absence in the real
###            distribution.
###    simP: a vector of presence / absence in the simulated
###            distribution.
###    realSimBin: a data.frame as defined in calcSimErrs
###    binVecP: a vector of distance bins for all pairs of
###            squares.
###    realCP: conditional probabilities for the real distribution
###            as calculated by calcCondProbs.
###    coords: a 2-column matrix or data.frame with x and y
###            coordinates for each square.

    # Count the distance classes:
    degree    <- length(neighbourLists)
    # Calculate the errors using the variogram:
    simErr    <- calcSimErrs(realSimBins = realSimBin, bins =
                 binVecP, sim = simP)
    # Calculate the errors using conditional probability:
    condProbP <- calcCondProbErrs(actualCPs = realCP, sim = simP)
    # Define the plot layout:
    layout(matrix(c(1,1,2,2,3,3,4,4,4,5,5,5,4,4,4,5,5,5),
                  ncol = 6, byrow = T))
    # Plot the conditional probabilities for the simulation (red).
    plot(1:degree, calcCondProbs(simP), xlab = "order", ylab =
        "probability", pch = 20, col = "red", cex = 2.2, ylim =
        c(0,1), main = "Probability | presence")
    # Plot the conditional probabilities of the real distribution:
    points (1:degree, calcCondProbs(actual), pch = 20, cex = 2.2)
    # add some text summarizing the statistics:
    plot(1,1, type = "n", axes = FALSE, ylab = "", main =
       "Fit Statistics", cex.main = 2, xlab = "")
    mtext(bquote(paste(plain('Conditional Probability Error ') ==
        .(round(condProbP, 2)), sep = "")), line = -5)
```

```
    mtext(bquote(paste(plain('Binned Variog. Error ') ==
        .(round(simErr, 2)), sep = "")), line = -3)
    # Plot the variograms of the real distribution:
    simErr <- calcSimErrs(realSimBins = realSimBin, bins =
              binVecP, sim = simP, plot.it = T)
    # Plot the two distributions:
    plot(coords[,2], coords[,1], col = actual+1, pch = 20, cex =
        2, xlab = "long", ylab = "lat", main = "Real Dist")
    plot(coords[,2], coords[,1], col = simP+1, pch = 20, cex = 2,
        xlab = "long", ylab = "lat", main = "Simulated Dist")
}

#########################################################################
### ----------------------------------------------------------- ###
#########################################################################

#### The main function:

randDists <- function(pa, nsims = 99, maxiters = 10000,
                      plot.final = TRUE, binVecs, neighbourList) {
### This function produces (very slowly) a null distribution with
### equal prevalence and similar conditional probability to a
### given distribution. Its arguments are:
### pa: a vector of 0 and 1 indicating presence/absence in each
###     square.
### nsims: the number of simulations required.
### maxiters: the maximum number of iterations required if
###     convergence is not reached.
### plot.final: logical, describing whether to plot the final
###     distribution of each null model with summary statistics.
### binVecs: a numeric vector defining the distance classes of
### each pairwise comparison of points.
### neighbourList: a list of lists as defined above in
###     calcCondProbs.

    # First calculate summary statistics for the real
    # distribution:
    binVec         <- binVecs
    neighbourLists <- neighbourList
    realSim        <- matrix (abs (pa [rep (1:nlocs, times =
                       nlocs)] - pa[rep(1: nlocs, each = nlocs)]),
                       nrow = nlocs)
    realSim        <- realSim[col(realSim) < row(realSim)]
    realSimBinned  <- aggregate(realSim, by = list(binVec), sum)
    realHists      <- calcHists(pa)
    realProbs      <- calcCondProbs(pa)
    realNeighSums  <- calcAllSums(pa)

    # Set up results matrix (if required):
    if (nsims > 1) out <- matrix(NA, ncol = nsims, nrow = nlocs)
```

```
# For each simulation required:
for (jj in 1:nsims) {

    # Generate initial map:
    sim1         <- initMap(pa)
    # save map as bestSim:
    bestSim      <- sim1
    # Calculate summary statistics for simulation:
    simErr       <- calcSimErrs(sim = sim1, realSimBins =
                          realSimBinned)
    simHists     <- calcHists(sim1)
    simProbs     <- calcCondProbs(sim1)
    simNeighSums <- calcAllSums(sim1)
    degree       <- length(realHists)
    HistErr      <- calcHistErrs(actual = realHists, sim =
                         sim1)
    CPErr        <- calcCondProbErrs(actualCPs = realProbs,
                         sim = sim1)
    oldSimErr    <- CPErr
    # Define counter:
    loop <- 0

    while (!(CPErr < 5 & loop > 100) & loop <= maxiters) {

        # While the error is greater than the tolerance or
        # the process has run for fewer than 100 iterations
        # and less than the maximum, do pairwise swapping of
        # presence/absence:
        loop           <- loop + 1   # Increment loop
        # Identify the distance class with probabilities
        # differing most between the real and simulated
        # distribution:
        simProbs       <- calcCondProbs(sim1)
        condProbDiffs  <- realProbs - simProbs
        condProbErr    <- sqrt(mean((condProbDiffs)^2))
        mostWrongDegree <- which (abs(condProbDiffs) ==
                              max(abs(condProbDiffs)))[1]
        pOnly          <- sim1 == 1

        # Identify neighbours for all squares within this most
        # differing class and estimate distance classess with
        # most error:
        nnlist         <- nNeighList[[mostWrongDegree]][pOnly]
        presSums       <- mapply(function(vec) sum(sim1[vec]),
                          neighbourLists[[mostWrongDegree]][
                          pOnly])
        simHist        <- hist(presSums / nnlist, breaks =
                          seq(0, 1, length = 9), plot = FALSE)
        histDiffs      <- realHists[[mostWrongDegree]][[2]] –
                          simHist[[2]]
```

```
if(length(histDiffs) == 0) histDiffs <- 0 # if perfect
mostWrongBin   <- which(abs(histDiffs) == max (abs(
                  histDiffs)))[1]
# Identify whether the most strongly differing class
# is a result of over or under simulating within this
# class:
signOfWrong    <- sign (histDiffs[mostWrongBin])[1]

if(is.na(signOfWrong) | signOfWrong == 0) {
   signOfWrong <- -1
}
# Set up parameters for this most differing class:
simNeighSums   <- calcOneSum(sim1,
                  neighbourLists[[mostWrongDegree]])
bins           <- seq(0, 1, length = 9)

# identify squares whose of presence / absence is most
# likely result in an improvement:
if (signOfWrong == -1) {
    # If simulation is an overestimate:
    # Identify the bin that is most overestimated:
    mostOppositeBin <- which(histDiffs == max (
                       histDiffs))[1]
    # Identify squares with neighbours matching these
    # requirements and select one from the top 50,
    # biasing selection probability to the best
    # candidates:
    candidates <- simNeighSums[sim1 == 1] /
              nNeighList[[mostWrongDegree]][sim1 == 1]
    candidates <- (1:nlocs)[sim1 == 1][order( abs(
                  candidates - mean(
                  bins[mostWrongBin],
                  bins[mostWrongBin + 1])),
                  decreasing = FALSE)]
    selectedloc <- sample( candidates[1 : (min (sum (
                   pa), 50))], 1, prob = seq(.1, .01,
                   length = min(sum(pa),50)))
    #Set simulation to 0 in this location:
    sim1[selectedloc] <- 0
    # Repeat process to identify a location to swap
    # distribution to 1.
    candidates <- simNeighSums[sim1 != 1] /
                  nNeighList[[mostWrongDegree]][sim1
                  != 1]
    candidates <- (1:nlocs)[sim1 != 1][order( abs(
                  candidates - mean(
                  bins[mostOppositeBin],
                  bins[mostOppositeBin + 1])),
                  decreasing = FALSE)]
    selectedloc <- sample( candidates[1 : (min (sum(
                   pa), 50))], 1, prob = seq(.1, .01,
```

```
                    length = (min(sum(pa),50))))
        sim1[selectedloc] <- 1

    } else {
        # Repeat above procedure if simulation was
        # underestimated in the most wrong distance class:
        mostOppositeBin <- which(histDiffs == min (
                           histDiffs))[1]
        candidates <- simNeighSums[sim1 != 1] /
                      nNeighList[[mostWrongDegree]][sim1
                      != 1]
        candidates <- (1:nlocs)[sim1 != 1][order( abs(
                      candidates - mean(
                      bins[mostWrongBin],
                      bins[mostWrongBin + 1])),
                      decreasing = FALSE)]
        selectedloc <- sample( candidates[1 : (min (sum(
                       pa),50))], 1, prob = seq(.1, .01,
                       length = min(sum(pa),50)))
        sim1[selectedloc] <- 1
        candidates <- simNeighSums[sim1 == 1] /
                      nNeighList[[mostWrongDegree]][sim1
                      == 1]
        candidates <- (1:nlocs)[sim1 == 1][order (abs (
                      candidates - mean(
                      bins[mostOppositeBin],
                      bins[mostOppositeBin + 1])),
                      decreasing = FALSE)]
        selectedloc <- sample( candidates[1 : (min (sum(
                       pa),50))], 1, prob = seq( .1, .01,
                       length = (min( sum( pa), 50))))
        sim1[selectedloc] <- 0
    }

    if (loop %% 20 == 0) {
        # Every 20 loops calculate the errors:
        CPErr  <- calcCondProbErrs(actualCPs = realProbs,
                  sim = sim1)
        simErr <- calcSimErrs(sim = sim1, realSimBins =
                  realSimBinned)
        if (loop == 100) {
            # Throw away first 100 iterations
            bestSim <- sim1
            oldSimErr <- CPErr
        }
        # Redefine errors with new distribution:
        oldSimErr <- min( oldSimErr, CPErr)
        print( paste ("simErr = ", round(simErr,1), ",
               CPErr = ", round(CPErr,1), sep = ""))
        if(oldSimErr == (CPErr)) {
            # If the error of the current simulation is
```

```
                    # lower than that stored, save the current
                    # simulation.
                    bestSim <- sim1
                }
            }
        }
        if (nsims > 1) {
            # If multiple simulations are required, store the best
            # simulation in the appropriate column of out.
            out[,jj] <- bestSim
            print(paste("Simulation", jj, "compete"))
        }
    }
    # If required, plot summaries and final distributions:
    if (plot.final) plotFinal(pa, bestSim, realSimBin =
        realSimBinned, binVecP = binVec, realCP = realProbs)
    # Return final null distributions:
    if (nsims > 1) return(out) else return (bestSim)
}

#########################################################################
###-----------------------------------------------------------------###
#########################################################################
```

R code described in the text: (2) Code used to fit climate envelopes:

```
fitEnvelope <- function (organismData, weatherData, method,
               outGrid = NULL, do.cv = TRUE, do.var.imp= FALSE,
               outputPredictions = TRUE, outputFile = NULL) {

### This function fits climate envelopes to datasets using BIOMOD
### defaults, outputting predictions and models. It requires
### libraries nnet, verification, mgcv. Its arguments are:
### organismData: a vector of 0 and 1 indicating presence and
###         absence in each square.
### weatherData: a numeric matrix or data.frame containing the
###         current climate conditions: rows represent squares in
###         the same order as organismData, each column represents
###         one weather variable.
### method: a character string identifying the fitting method to
###         use. Currently, one of “nnet”, “gam” or “glm”.
### outGrid: if present, a matrix of the same form as weatherData
###         defining the future climate scenarios at each square.
### do.cv: logical, defining whether the root mean squared error
###         for the model should be measured using 10-fold cross
###         validation.
### do.var.imp: logical, indicating whether leave-one-out cross
###         validation should be used to assess the importance of
###         each weather variable.
### outputPredictions: logical – is the predicted distribution
###         required?
### outputFile: an optional character vector defining a text file
###         to be created to hold output (useful when runing
###         batches).

    # Load required packages:
    require (nnet)
    require (verification)

    # Define variables:
    dat1            <- organismData
    weather         <- weatherData
    weather         <- as.matrix (weather)
    nRows           <- length (dat1)

    # If only one weather variable, rename:
    if(dim(weather)[2] == 1) {
        colnames(weather) <- "weather[randSel,]"
    }

    # If future scenarios are given, format and define output
    # matrix:
    if(!is.null(outGrid)) {
        outGrid     <- as.matrix(outGrid)
        outRows     <- dim(outGrid)[1]
```

```
    outPreds    <- matrix (nrow = outRows, ncol = 10)
}

# Select 70% of indices including at least 2 presences:
randSel         <- sample (1:nRows, ceiling((nRows * 7) / 10))
while (sum (dat1[randSel]) < 2)   randSel <- sample (1:nRows,
       ceiling((nRows*7)/10))

# Define output matrices:
trainingPreds   <- matrix(nrow = length (randSel), ncol = 10)
validationPreds <- matrix(nrow = nRows – length(randSel), ncol
                   = 10)
finalPreds      <- matrix(nrow = nRows, ncol = 10)

if (method == "nnet") {
    # If neural networks are required...

   for (j in 1:10) {
   # fit 10 neural networks using BIOMOD defaults:
        assign (paste ("fullnet", j, sep = ""), nnet (x =
                weather[randSel,], y = dat1[randSel], size =
                7, decay = 0.03, maxit = 1000, trace = F))

        # Do predictions for each of the training data, the
        # validation data, the complete dataset and, if
        # required, the scenarios:
        trainingPreds[,j]   <- get (paste ("fullnet", j, sep =
                               ""))$fitted.values
        validationPreds[,j] <- predict (get (paste ("fullnet",
                               j, sep = "")), newdata =
                               weather[-randSel,], type =
                               "raw")
        finalPreds[,j]      <- predict (get (paste ("fullnet",
                               j, sep = "")), newdata =
                               weather, type = "raw")
        if(!is.null(outGrid)) {
            outPreds[,j]    <- predict (get (paste ("fullnet",
                               j, sep = "")), newdata =
                               outGrid, type = "raw")
        }
    }
    # Calculate the mean predictions across the 10 neural
    # networks:
    trainingPreds   <- rowMeans(trainingPreds)
    validationPreds <- rowMeans(validationPreds)
    finalPreds      <- rowMeans(finalPreds)
    if(!is.null(outGrid)) outPreds <- rowMeans(outPreds)

} else if (method == "glm") {
# If fitting method is “glm”...
```

```
# Hide warnings:
def.opt <- options()
options(warn = -1)
# Identify climate variables:
vars <- colnames (weather)

# Calculate the number of two-way interactions to fit,
# based on the length of the dataset and number of
# variables and paste together an appropriate formula:
if(length(vars)>1) {
    nTwoWay <- factorial( length( vars)) / (2 * factorial(
               length( vars) - 2))
    if ((nTwoWay + 2 * length(vars) * 3) <
               length(weather[, 1])) {
        twoWayInts <- paste( rep( vars, each =
                      length( vars)), rep( vars, times =
                      length( vars)), sep = ":")
        ind <- rep(TRUE, length = length(vars)^2)
        for (ii in 1:length(vars)){
            for (jj in 1:length(vars)) {
                if (ii >= jj) {
               ind[(ii - 1) * length( vars) + jj] <- FALSE
                }
            }
        }
        form1 <- as.formula(paste("dat1[randSel] ~ ",
                 paste(paste(vars, collapse = " + "),
                 paste("I(", vars, "^2)", sep = "",
                 collapse = " + "), paste (
                 twoWayInts[ind], collapse = " + "), sep =
                 " + ")))
    }
} else {
    form1 <- as.formula(paste("dat1[randSel] ~ ", paste(
             paste( vars, collapse = " + "), paste("I(",
             vars, "^ 2)", sep = "", collapse = " + "),
             sep = " + ")))
}

# Fit the glm model:
normal.glm <- glm (form1, family = binomial, data =
              as.data.frame( weather[randSel, ]))
# Reset the options:
options(def.opt)
# Make predictions from the model for the training,
# validation, full and (if required) future climate
# datasets:
finalPreds      <- predict (normal.glm, newdata =
                   as.data.frame( weather), type =
                   "response")
validationPreds <- predict (normal.glm, newdata =
```

```
                    as.data.frame( weather[-randSel, ]),
                    type = "response")
    trainingPreds   <- normal.glm$fitted.values
    if(!is.null(outGrid)) {
        outPreds <- predict (normal.glm, newdata =
                    as.data.frame( outGrid), type =
                    "response")
    }

} else if (method == "gam") {
# If the fitting method required is “gam”...

    # Coerce climate data to suitable data.frames and rename
    # if required:
    weather     <- as.data.frame(weather)
    testWeather <- weather[randSel,]
    valWeather  <- weather[-randSel,]
    if(length(names(weather)) == 1) {
        names(weather)     <- "weath"
        names(testWeather) <- "weath"
        names(valWeather)  <- "weath"
    }

    # Turn warnings off and require package mgcv:
    def.opt <- options()
    options(warn = -1)
    require (mgcv)

    # Paste formula from names and fit gam model:
    normal.gam <- eval (parse(text = paste ("gam (
                  dat1[randSel] ~ s (", paste( colnames(
                  weather), collapse = ","), ", k = 8,",
                  "fx = FALSE), family = binomial, data = ",
                  "testWeather)")))

    # Reset options:
    options(def.opt)
    # Make predictions from the model for the training,
    # validation, full and (if required) future climate
    # datasets:
    finalPreds       <- predict (normal.gam, newdata =
                        weather, type = "response")
    validationPreds  <- predict (normal.gam, newdata =
                        valWeather,  type = "response")
    trainingPreds    <- normal.gam$fitted.values
    if(!is.null(outGrid)) {
        outGrid <- as.data.frame(outGrid)
        if(length(names(outGrid)) == 1) {
            names(outGrid) <- "weath"
        }
        outPreds <- predict (normal.gam, newdata = outGrid,
```

```
                    type = "response")
    }
}

# Calculate a matrix of the four corners of a contingency
# table comparing predicted and real presence/absence for the
# complete range of cutoff values in [0,1]:
cutoff <- seq (0, 1, by = 0.01)
result <- matrix (ncol = 101, nrow = 4)
for (j in 1:101) {
    result[1,j] <- sum (trainingPreds > cutoff[j] &
                        dat1[randSel] == 1)
    result[2,j] <- sum (trainingPreds > cutoff[j] &
                        dat1[randSel] == 0)
    result[3,j] <- sum (trainingPreds < cutoff[j] &
                        dat1[randSel] == 1)
    result[4,j] <- sum (trainingPreds < cutoff[j] &
                        dat1[randSel] == 0)
}

kappa <- function (a, b, c, d) {
# This functions calculates Cohen's Kappa.
    kappa <- (2 * ((a * d) - (b * c)))/(((a + b) * (b + d)) +
             ((c + d) * (a + c)))
}

# Generate from training data a vector of Kappa values for all
# possible cutoffs :
kappa.vals       <- kappa (a = result[1,], b = result[2,], c =
                           result[3,], d = result[4,])
# Find maximum Kappa value:
maxKappa         <- max(kappa.vals)
# Define optimal cutoff based on training data
optimal.cutoff   <- cutoff[max (which(kappa.vals == max (
                    kappa.vals)))]

# Calculate AUC for validation dataset:
AUC <- roc.area (dat1[-randSel], as.vector (
       validationPreds))$A

# Set predictions according to cutoff:
finalPreds[finalPreds > optimal.cutoff] <- 1
finalPreds[finalPreds < optimal.cutoff] <- 0

if(do.cv) {
# If 10-fold cross-validation is required...

    # Define variables to store results:
    MSE       <- numeric (10)
    CVerror   <- matrix (NA, ncol = dim (weather)[2], nrow =
                         10)
```

```
# Generate cross-validation subsets:
cvsubsets <- sample (rep(1:10, ceiling( length( randSel) /
                    10)))[1 : length( randSel)]
for (ii in 1:10) {
    assign (paste ("randSel", ii, sep = ""),
            randSel[cvsubsets != ii])
}

for (k in 1:10) {
# For each of the 10-fold cross validations...

    # Repeat above procedure on the required subset (code
    # is effectively a repeat of above with different
    # datasets):
    if (method == "nnet") {
        CVPreds <- matrix(nrow = length( randSel[cvsubsets
                   == k]), ncol = 10)
        for (j in 1:10) {

            assign (paste ("fullnetCV", j, sep = ""),
                    nnet( x =weather[get(paste("randSel",
                    k, sep = "")),], y = dat1[get( paste(
                    "randSel", k, sep = ""))], size = 7,
                    decay = 0.03, maxit = 1000, trace =
                    FALSE))

            CVPreds[,j] <- predict (get (paste (
                          "fullnetCV", j, sep = "")),
                          newdata =
                          weather[randSel[cvsubsets ==
                          k],], type = "raw")
        }
        CVPreds <- rowMeans(CVPreds)
        MSE[k]  <- sum(((dat1[randSel[cvsubsets == k]] –
                   CVPreds) ^ 2)) / length (CVPreds)

    } else if (method == "glm") {

        def.opt <- options()
        options(warn = -1)
        if ((nTwoWay + 2 * length( vars) * 3) < length(
             weather[,1])) {
            form1 <- as.formula( paste(
                    "dat1[randSel[cvsubsets != k]] ~ ",
                    paste( paste( vars, collapse = " +
                    "), paste("I(", vars, "^2)", sep =
                    "", collapse = " + "), paste (
                    twoWayInts[ind], collapse = " + "),
                    sep = " + ")))
        } else {
            form1 <- as.formula( paste(
```

```
                "dat1[randSel[cvsubsets != k]] ~ ",
                paste( paste( vars, collapse = " + "),
                paste("I(", vars, "^2)", sep = "",
                collapse = " + "), sep = " + ")))
    }
    cv.glm <- glm (form1, family = binomial, data =
              as.data.frame( weather[randSel[cvsubsets
              != k],]))
    options(def.opt)
    CVPreds <- predict (cv.glm, newdata =
               as.data.frame(
               weather[randSel[cvsubsets == k],]),
               type = "response")
    MSE[k] <- sum(((dat1[randSel[cvsubsets == k]] –
              CVPreds) ^ 2))/length (CVPreds)

} else if (method == "gam") {

    def.opt <- options()
    options(warn = -1)
    normal.gam <- eval (parse( text = paste ("gam (
                  dat1[randSel[cvsubsets !=k]]",
                  " ~ s (", paste(colnames(weather),
                  collapse = ","), ", k = 8,", "fx =
                  FALSE), family = binomial, data = ",
                  "as.data.frame(
                  weather[randSel[cvsubsets != k],
                  ]))")))
    options(def.opt)
    CVPreds <- predict (normal.gam, newdata =
               as.data.frame(
               weather[randSel[cvsubsets == k],]),
               type = "response")
    MSE[k]  <- sum(((dat1[randSel[cvsubsets == k]] –
               CVPreds) ^ 2)) / length (CVPreds)
}

if(do.var.imp) {
# If a leave-one-out cross validation of the
# importance of each parameter is also required...

    for (i in 1:dim(weather)[2]) {
    # For each climate variable repeat the cross-
    # validation procedure dropping each climate
    # variable in turn. Code is again a repeat of
    # above.

        if (method == "nnet") {
            CVSubPreds <- matrix(nrow = length(
                          randSel[cvsubsets == k]),
                          ncol = 10)
```

```
            for (j in 1:10) {
                assign (paste ("CVsubnet", j, sep =
                        ""), nnet (x =
                        weather[randSel[cvsubsets !=
                        k], -i], y =
                        dat1[randSel[cvsubsets != k]],
                        size = 7, decay = 0.03,
                        maxit = 1000, trace = F))

                CVSubPreds[,j] <- predict (get (paste
                        ("CVsubnet", j, sep = "")),
                        newdata =
                        weather[randSel[cvsubsets ==
                        k], -i], type = "raw")
            }
            CVSubPreds   <- rowMeans( CVSubPreds)
            CVerror[k,i] <- sum(
                          ((dat1[randSel[cvsubsets ==
                          k]] - CVSubPreds) ^ 2)) /
                          length (CVSubPreds)

        } else if (method == "glm") {
            def.opt <- options()
            options(warn = -1)
            vars <- colnames (weather[,-i])
            nTwoWay <- factorial(length( vars)) / (2 *
                       factorial (length(vars) - 2))
            if ((nTwoWay + 2 * length( vars)) * 3 <
            length( weather[,1])) {
                twoWayInts <- paste( rep( vars, each =
                              length( vars)), rep(
                              vars, times = length(
                              vars)), sep = ":")
                ind <- rep(TRUE, length = length(
                              vars)^2)
                for (ii in 1:length(vars)){
                    for (jj in 1:length(vars)) {
                        if (ii >= jj) {
                            ind[(ii - 1) * length(
                            vars) + jj] <- FALSE
                        }
                    }
                }
                form1 <- as.formula( paste(
                        "dat1[randSel[cvsubsets != k]]
                        ~ ", paste( paste( vars,
                        collapse = " + "), paste("I(",
                        vars, "^2)", sep = "",
                        collapse = " + "), paste (
                        twoWayInts[ind], collapse = "
```

```
                        + "), sep = " + ")))
                } else {
                    form1 <- as.formula( paste(
                            "dat1[randSel[cvsubsets != k]]
                            ~ ", paste(paste(vars,
                            collapse = " + "), paste("I(",
                            vars, "^2)", sep = "",
                            collapse = " + "), sep = " +
                            ")))
                }
                sub.glm <- glm (form1, family = binomial,
                          data = as.data.frame(
                          weather[randSel[cvsubsets !=
                          k], ]))
                options(def.opt)
                CVSubPreds <- predict (sub.glm, newdata =
                            as.data.frame(weather[randSel
                            [cvsubsets == k], ]),type =
                            "response")
                CVerror[k,i] <- sum( ((dat1[randSel
                             [cvsubsets == k]] –
                             CVSubPreds) ^ 2)) /
                             length (CVSubPreds)

            } else if (method == "gam") {

                def.opt <- options()
                options(warn = -1)
                sub.gam <- eval (parse(text = paste ("gam
                          (dat1[randSel[cvsubsets!=k]] ~ s
                          (", paste( colnames( weather[ ,
                          -i]), collapse = ","), ", k = 8,
                          fx = FALSE), family = binomial,
                          data = ", "as.data.frame(
                          weather[randSel [cvsubsets !=
                          k], ]))")))
                options(def.opt)

                CVSubPreds  <- predict (sub.gam, newdata =
                               as.data.frame(
                               weather[randSel [cvsubsets
                               == k], ]), type =
                              "response")
                CVerror[k, i] <- sum( ((dat1[randSel
                                 [cvsubsets == k]] –
                                 CVSubPreds) ^ 2)) /
                                 length (CVSubPreds)
            }
        }
    }
}
```

```
}

if(do.var.imp) {
# If leave-one-out cross validation of the importance of each
# parameter is required...

    # Define some objects to hold the results:
    diffMinusVarI <- matrix(NA, ncol = 3, nrow = dim(
                     weather)[2])
    rownames (diffMinusVarI)     <- colnames(weather)
    colnames (diffMinusVarI)     <- c("Kappa", "AUC", "CV")
    if (do.cv) diffMinusVarI[,3] <- colMeans(CVerror) –
                                    mean(MSE)

    for (i in 1:dim(weather)[2]) {
    # For each climate variable in turn, fit models and make
    # predictions as above, leaving the climate parameters out
    # sequentially.
        if (method == "nnet") {
            trainingSubPreds   <- matrix (nrow = length(
                                  randSel), ncol = 10)
            validationSubPreds <- matrix (nrow = nRows –
                                  length( randSel), ncol = 10)
            finalSubPreds      <- matrix(nrow = nRows, ncol =
                                  10)
            for (j in 1:10) {

                assign (paste ("subnet", j, sep = ""), nnet(x=
                        weather[randSel, -i], y =
                        dat1[randSel], size = 7, decay = 0.03,
                        maxit = 1000, trace = F))

                trainingSubPreds[,j] <- get (paste ("subnet",
                                        j, sep =
                                        ""))$fitted.values
                validationSubPreds[,j] <- predict (get (paste(
                                        "subnet", j, sep = "")),
                                        newdata = weather[
                                        -randSel, -i], type =
                                        "raw")
                finalSubPreds[,j] <- predict (get (paste (
                                        "subnet", j, sep = "")),
                                        newdata = weather[,-i],
                                        type = "raw")
            }
            trainingSubPreds   <- rowMeans(trainingSubPreds)
            validationSubPreds <- rowMeans(validationSubPreds)
            finalSubPreds      <- rowMeans(finalSubPreds)

        } else if (method == "glm") {
```

```
        def.opt <- options()
        options(warn = -1)
        vars <- colnames (weather[,-i])
        nTwoWay <- factorial( length( vars)) / (2 *
                   factorial (length(vars) - 2))
        if ((nTwoWay + 2 * length( vars)) * 3 <  length(
                  weather[ ,1])) {
            twoWayInts <- paste(rep(vars, each = length(
                          vars)), rep( vars, times =
                          length( vars)), sep = ":")
            ind <- rep(TRUE, length = length( vars) ^ 2)
            for (ii in 1:length(vars)){
                for (jj in 1:length(vars)) {
                    if (ii >= jj) {
                        ind[(ii - 1) * length( vars) +
                        jj] <- FALSE
                    }
                }
            }
            form1 <- as.formula( paste( "dat1[randSel] ~
                     ", paste( paste( vars, collapse = " +
                     "), paste( "I(", vars, " ^ 2)", sep =
                     "", collapse = " + "), paste (
                     twoWayInts[ind], collapse = " + "),
                     sep = " + ")))
        } else {
            form1 <- as.formula( paste( "dat1[randSel] ~
                     ", paste( paste( vars, collapse = " +
                     "), paste( "I(", vars, "^2)", sep =
                     "", collapse = " + "), sep = " + ")))
        }
        sub.glm <- glm (form1, family = binomial, data =
                   as.data.frame( weather[randSel, ]))
        options(def.opt)
        finalSubPreds      <- predict (sub.glm, newdata =
                              as.data.frame( weather),
                              type = "response")
        validationSubPreds <- predict (sub.glm, newdata =
                              as.data.frame( weather[
                              -randSel, ]), type =
                              "response")
        trainingSubPreds   <- sub.glm$fitted.values

    } else if (method == "gam") {

        def.opt <- options()
        options(warn = -1)
        sub.gam <- eval (parse( text = paste ("gam (
                   dat1[randSel] ~ s (", paste( colnames(
                   weather[, -i]), collapse = ","), ", k =
                   8,", "fx = FALSE), family = binomial,
```

```
                    data = ", "as.data.frame(
                    weather[randSel, ]))")))
            options(def.opt)

            finalSubPreds       <- predict (sub.gam, newdata =
                                   as.data.frame( weather),
                                   type = "response")
            validationSubPreds  <- predict (sub.gam, newdata =
                                   as.data.frame( weather[
                                   -randSel,]), type =
                                   "response")
            trainingSubPreds    <- sub.gam$fitted.values
        }

        # As before, define results matrix for the full range
        # of cut-off values and select optimal cutoff using
        # Kappa statistic:
        subresult <- matrix (ncol = 101, nrow = 4)
        for (j in 1:101) {
            subresult[1,j] <- sum (trainingSubPreds >
                              cutoff[j] & dat1[randSel] == 1)
            subresult[2,j] <- sum (trainingSubPreds >
                              cutoff[j] & dat1[randSel] == 0)
            subresult[3,j] <- sum (trainingSubPreds <
                              cutoff[j] & dat1[randSel] == 1)
            subresult[4,j] <- sum (trainingSubPreds <
                              cutoff[j] & dat1[randSel] == 0)
        }
        kappa.subvals <- kappa (a = subresult[1, ], b =
                         subresult[2, ], c = subresult[3, ],
                         d = result[4, ])
        maxSubKappa   <- max(kappa.subvals)
        subcutoff     <- cutoff[max (which( kappa.subvals ==
                         max (kappa.subvals)))]

        # Calculate AUC for the model without this climate
        # variable:
        subAUC        <- roc.area (dat1[-randSel], as.vector (
                         validationSubPreds))$A
        # Calculate difference in AUC and Kappa statistics
        # between full and reduced model:
        diffMinusVarI[i, 1] <- maxKappa - maxSubKappa
        diffMinusVarI[i, 2] <- AUC - subAUC
    }
}

# Return model predictions, model and fit statistics as
# appropriate to input arguments and method:
if (method == "glm") {
    mainResults <- list(method = method, Kappa = maxKappa,
                   AUC = AUC, glm.model = normal.glm)
```

```
    } else {
        mainResults <- list(method = method, Kappa =  maxKappa,
                        AUC = AUC)
    }
    if (do.cv) mainResults$MSE <- mean(MSE)
    if (do.var.imp) {
        mainResults$variableImportance <-diffMinusVarI
    }
    if (outputPredictions) mainResults$fittedDistrib <- finalPreds
    if (!is.null(outGrid)) {
        mainResults$predictedDistrib <- outPreds
        mainResults$optimalCut       <- optimal.cutoff
    }

    if (!is.null(outputFile)) {
    # If required, write the output to a text file:

        outputFile = file (outputFile, open = "a")
        writeLines (paste ("species:  ", paste( deparse(
                    substitute( organismData)))), outputFile)
        writeLines (paste("method:  ", method), outputFile)
        writeLines (paste("climate:  ", paste( deparse(
                    substitute( weatherData)))), outputFile)
        writeLines (paste(mainResults), outputFile)
        close (outputFile)
    }

    # Return the final results:
    return (mainResults)
}
```

R code described in the text : (3) Power analysis pattern generation

```
cutModel <- function (clim1, clim2, clim3, clim4, Climate =
                      currentClim) {
### This function defines presence/absence for a distribution
### based on user provided tolerances to four climate variables.
### It is called repeatedly by the main function, to determine the
### deterministic pattern used. Its arguments are:
### clims1 – clims4: numeric vectors of length 2 defining the
###       upper and lower tolerances on each of the climate axes:
###       they should be passed in the same order as the columns
###       in currentClim.
### Climate: a matrix or data.frame containing the climate
###       variables, one per column, each row defining the climate
###       in one distribution square.

    pa <- Climate[,3] < clim1[2] & Climate[,3] > clim1[1] &
          Climate[,4] < clim2[2] & Climate[,4] > clim2[1] &
          Climate[,5] < clim3[2] & Climate[,5] > clim3[1] &
          Climate[,6] < clim4[2] & Climate[,6] > clim4[1]
    pa <- as.numeric(pa)
}


allClim <- function(pa, climate, additionalProp = 0,
                    allowDispersal = TRUE, neighbourLists) {
### This function generates a deterministic distribution to be
### used in the power test. It can generate completely
### deterministic patterns, or patterns containing an element of
### noise. It must be called only in an environment already
### containing the functions documented in earlier sections. Its
### arguments are:
### pa: a vector of presence/absence for a real distribution: it
###     is used to determine the required prevalence in the
###     deterministic pattern, and if a noisy pattern is to be
###     generated, the spatial structure of this pattern will be
###     the target for erosion following dispersal.
### climate: a matrix of climate variables, one row for each
###     location in the distribution.
### additionalProp: numeric. If desired, the deterministic
###     distribution can be generated with higher prevalence than
###     the actual distribution.
### allowDispersal: Logical, if TRUE once the deterministic
###     pattern with the required prevalence is generated, the
###     'species' disperses into all neighbours of squares with
###     presence, generating false positives.
### neighbourLists: a list of lists describing the adjacency
###     structure of squares in the distribution. It has structure
###     identical to that described in earlier functions.

    # Rename climate:
```

```
currentClim <- climate
# Calculate target prevalence:
n1    <- sum(pa, na.rm = T)
# Calculate prevalence plus additional proportion:
n     <- ceiling(n1 * (1 + additionalProp))

# Select some initial climate tolerances falling within the
# range of each climate variable:
clim1 <- sort(runif(2, min(currentClim[,3], na.rm = T),
         max(currentClim[,3], na.rm = T)))
clim2 <- sort(runif(2, min(currentClim[,4], na.rm = T),
         max(currentClim[,4], na.rm = T)))
clim3 <- sort(runif(2, min(currentClim[,5], na.rm = T),
         max(currentClim[,5], na.rm = T)))
clim4 <- sort(runif(2, min(currentClim[,6], na.rm = T),
         max(currentClim[,6], na.rm = T)))

# Define the first distribution based on these tolerances:
mod1              <- cutModel( clim1, clim2, clim3, clim4)
mod1[is.na(mod1)] <- 0
# Define some parameters:
nlocs             <- length(mod1)
nmod              <- sum(mod1)
# Print the number of presences in the distribution:
print(paste("n = ", n, "nmod = ", nmod))
# Determine whether initial model is too small or too large:
size              <- ifelse(nmod < n, "small", "large")

while(nmod != n) {
# While the prevalence in the model is not the target
# prevalence:

    # Calculate the proportion of each climate axis tolerated
    # by the 'species'
    prClim1 <- (clim1[2] - clim1[1]) / (Rclim1[2] - Rclim1[1])
    prClim2 <- (clim2[2] - clim2[1]) / (Rclim2[2] - Rclim2[1])
    prClim3 <- (clim3[2] - clim3[1]) / (Rclim3[2] - Rclim3[1])
    prClim4 <- (clim4[2] - clim4[1]) / (Rclim4[2] - Rclim4[1])
    # Save the current climate tolerance and indication of
    # current match:
    oldClims <- list(clim1, clim2, clim3, clim4, size, up)
    # Randomly select 0 or 1 to indicate whether to start by
    # tolerance at the upper or lower limit:
    up <- rbinom(1,1,0.5)

    if (nmod < n) {
    # If there are too few presences in the distribution widen
    # the climatic tolerance on an selected axis:

        # Identify the current distribution as too small:
        size <- "small"
```

```
if (size != oldClims[[5]]) {
# If the distribution has too few presences but
# previously had too many, narrow the same climate
# variable previously widened:

    narrow <- wide
    up <- oldClims[[6]]

} else {
# otherwise identify the narrowest climate
# tolerance...

    narrow <- which.min( c( prClim1, prClim2, prClim3,
              prClim4))
}

# and increase it, based on the previous tolerance to
# this variable:
if (narrow == 1) {
# If the first climate variable is the narrowest,
# select new limits for this one:

    if (size != oldClims[[5]]) {
        if(up == 1) {
            clim1[2] <- runif(1, clim1[2],
                        oldClims[[1]][2])
        } else {
            clim1[1] <- runif(1, oldClims[[1]][1],
                        clim1[1])
        }
    } else {
        if(up == 1) {
            clim1[2] <- runif(1, clim1[2], max(
                        currentClim[,3], na.rm = T))
        } else {
            clim1[1] <- runif(1, min(currentClim[,3],
                        na.rm = T), clim1[1])
        }
    }

} else {
# Otherwise...

    if (narrow == 2) {
    # if the first climate variable is the narrowest,
    # select new limits for this one:

        if (size != oldClims[[5]]) {
            if(up == 1) {
                clim2[2] <- runif(1, clim2[2],
                            oldClims[[2]][2])
```

```
            } else {
                clim2[1] <- runif(1, oldClims[[2]][1],
                            clim2[1])
            }
        } else {
            if(up == 1) {
                clim2[2] <- runif(1, clim2[2], max(
                            currentClim[,4], na.rm =
                            TRUE))
            } else {
                clim2[1] <- runif(1, min(
                            currentClim[,4], na.rm =
                            TRUE), clim2[1])
            }
        }
    } else {
    # Etc.
        if (narrow == 3) {
            if (size != oldClims[[5]]) {
                if(up == 1) {
                    clim3[2] <- runif(1, clim3[2],
                                oldClims[[3]][2])
                } else {
                    clim3[1] <- runif(1,
                                oldClims[[3]][1],
                                clim3[1])
                }
            } else {
                if(up == 1) {
                    clim3[2] <- runif(1, clim3[2],
                                max(currentClim[,5],
                                na.rm = TRUE))
                } else {
                    clim3[1] <- runif(1, min(
                                currentClim[,5],
                                na.rm = T), clim3[1])
                }
            }
        } else {
            if (size != oldClims[[5]]) {
                if(up == 1) {
                    clim4[2] <- runif(1, clim4[2],
                                oldClims[[4]][2])
                } else {
                    clim4[1] <- runif(1,
                                oldClims[[4]][1],
                                clim4[1])
                }
            } else {
                if(up == 1) {
                    clim4[2] <- runif(1, clim4[2],
```

```
                                    max(currentClim[,6],
                                    na.rm = TRUE))
                    } else {
                        clim4[1] <- runif(1, min(
                                    currentClim[, 6],
                                    na.rm = T), clim4[1])
                    }
                }
            }
        }
        # Generate a new model distribution with the wider
        # tolerance:
        mod1 <- cutModel(clim1, clim2, clim3, clim4)
        mod1[is.na(mod1)] <- 0
        nmod <- sum(mod1)

    } else {
    # Or if the distribution has too many presences, do the
    # opposite of above to choose a parameter to narrow:

        if(nmod > n) {
            size <- "large"
            if (size != oldClims[[5]]) {
                wide <- narrow
                up <- oldClims[[6]]
            } else {
                wide <- which.min( c (prClim1, prClim2,
                         prClim3, prClim4))
            }
            if (wide == 1) {
                if (size != oldClims[[5]]) {
                    if(up == 1) {
                        clim1[2] <- runif(1, clim1[2],
                                    oldClims[[1]][2])
                    } else {
                        clim1[1] <- runif(1, oldClims[[1]][1],
                                    clim1[1])
                    }
                } else {
                    if(up == 1) {
                        clim1[2] <- runif(1, clim1[1],
                                    clim1[2])
                    } else {
                        clim1[1] <- runif(1, clim1[1],
                                    clim1[2])
                    }
                }
            } else {
                if (wide == 2) {
                    if (size != oldClims[[5]]) {
```

```
                if(up == 1) {
                    clim2[2] <- runif(1, clim2[2],
                                oldClims[[2]][2])
                } else {
                    clim2[1] <- runif(1,
                                oldClims[[2]][1],
                                clim2[1])
                }
            } else {
                if(up == 1) {
                    clim2[2] <- runif(1, clim2[1],
                                clim2[2])
                } else {
                    clim2[1] <- runif(1, clim2[1],
                                clim2[2])
                }
            }
        } else {
            if (wide == 3) {
                if (size != oldClims[[5]]) {
                    if(up == 1) {
                        clim3[2] <- runif(1, clim3[2],
                                    oldClims[[3]][2])
                    } else {
                        clim3[1] <- runif(1,
                                    oldClims[[3]][1],
                                    clim3[1])
                    }
                } else {
                    if(up == 1) {
                        clim3[2] <- runif(1, clim3[1],
                                    clim3[2])
                    } else {
                        clim3[1] <- runif(1, clim3[1],
                                     clim3[2])
                    }
                }
            } else {
                if (size != oldClims[[5]]) {
                    if(up == 1) {
                        clim4[2] <- runif(1, clim4[2],
                                    oldClims[[4]][2])
                    } else {
                        clim4[1] <- runif(1,
                                    oldClims[[4]][1],
                                    clim4[1])
                    }
                } else {
                    if(up == 1) {
                        clim4[2] <- runif(1, clim4[1],
                                    clim4[2])
```

```
                            } else {
                                clim4[1] <- runif(1, clim4[1],
                                            clim4[2])
                            }
                        }
                    }
                }
            }
        }
    }
    # Generate a new model distribution with the narrower
    # tolerance:
    mod1 <- cutModel(clim1, clim2, clim3, clim4)
    mod1[is.na(mod1)] <- 0
    nmod <- sum(mod1)
    print(paste("n = ", n, "nmod = ", nmod))
}

# Now the required prevalence has been achieved, store this
# pattern as the fully deterministic pattern:
detMod <- mod1

if (allowDispersal) {
# If the 'species' is allowed to disperse into neighbouring
# squares (generating false positives) as part of the noise
# generate procedure, identify the neighbours and set presence
# in the first order neighbours to 1:

    pOnly <- mod1 == 1
    nns <- unique(unlist(neighbourLists[[1]][pOnly]))
    mod1[nns] <- 1
}

# Count the new presences:
nmod       <- sum(mod1)

# Count the number of neighbours each square has in each
# distance class:
nNeighList <- lapply(neighbourLists, function(lst) {
                mapply(length, lst)})

while (nmod != n1) {
# While the distribution still has more than the required
# prevalence:

    # Calculate the summary statistics (including conditional
    # probabilities) for the spatial pattern exactly as in the
    # null distribution generation function, randDists:
    realHists       <- calcHists( pa)
    simProbs        <- calcCondProbs( mod1)
    condProbDiffs   <- condProbs - simProbs
```

```
condProbErr     <- sqrt( mean(( condProbDiffs) ^ 2))
# Identify distance class that is most different between
# the simulation and the real distribution:
mostWrongDegree <- which( abs( condProbDiffs) ==
                    max( abs( condProbDiffs)))[1]
# Identify the squares with simulated presence:
pOnly           <- mod1 == 1
# Select the appropriate neighbour list and identify the
# squares which, if turned to absence, would most likely
# to improve the conditional probability match between the
# simulation and real distribution (exactly as in
# randDists and documented there):
nnlist         <- neighbourLists[[mostWrongDegree]][pOnly]
presSums        <- mapply (function (vec) sum(mod1[vec]),
                   neighbourLists[[mostWrongDegree]][pOnly])
simHist         <- hist( presSums / sapply( nnlist,
                    length), breaks = seq(0, 1, length=
                    9), plot = FALSE)
histDiffs       <- realHists[[mostWrongDegree]][[2]] –
                    simHist[[2]]
mostWrongBin    <- which(abs(histDiffs) == max (abs(
                    histDiffs)))[1]
signOfWrong     <- sign (histDiffs[mostWrongBin])
simNeighSums    <- calcOneSum(mod1,
                    neighbourLists[[mostWrongDegree]])
bins            <- seq(0, 1, length = 9)

if (signOfWrong == -1) {
# If the most wrong distance class is a consequence of too
# many presences in this class, identify candidates,
# select one and set to 0:

    candidates  <- simNeighSums[mod1 == 1] /
                    nNeighList[[mostWrongDegree]][mod1 ==
                    1]
    candidates  <- (1:(min(sum(pa), 50)))[mod1 ==
                    1][order(abs( candidates - mean(
                    bins[mostWrongBin], bins[mostWrongBin +
                    1])), decreasing = FALSE)]
    selectedloc <- sample(candidates[1:(min(sum(pa),
                    50))], 1, prob = seq(.1, .01, length =
                    min( sum(pa), 50)))
    mod1[selectedloc] <- 0

} else {
# Otherwise select the distance class with most errors
# caused by overestimates and select one to remove as
# before:
    mostOppositeBin <- which(histDiffs == min (
                        histDiffs))[1]
    candidates <- simNeighSums[mod1 == 1] /
```

```
                    nNeighList[[mostWrongDegree]][mod1 == 1]
            candidates <- (1:nlocs)[mod1 == 1][order( abs(
                    candidates - mean(bins[mostOppositeBin],
                    bins[mostOppositeBin+1])), decreasing =
                    FALSE)]
            selectedloc <- sample( candidates[1:(min(sum(pa),
                     50))], 1, prob = seq(.1, .01, length =
                     min( sum( pa), 50)))
            mod1[selectedloc] <- 0
        }

        # Print the error in prevalence:
        print( sum( mod1) - n1)
        nmod <- sum( mod1)
    }
    # Calculate the proportion of the final pattern that was from
    # the deterministic pattern (the signal to noise ratio):
    SNR <- sum( detMod == 1 & mod1 == 1) / n1

    # Return the simulated distribution and the signal to noise
    # ratio:
    return(list(PA = mod1, SNR = SNR))
}
```