Lora Murphy
October, 2005

Maximum Likelihood Estimation of Neighborhood Models using Simulated Annealing

A User's Guide to the `neighparam` Package for R

1. Syntax

This tutorial uses several formatting features:

1. R commands written IN THE CONSOLE will be on lines beginning with ">", and will be in bold
2. Italics within an R command are used to identify terms that are made up by the user (i.e. not a reserved word or formal R term)
3. Portions of R script files are written in `Courier font.`

2. Getting Ready for R

Before you use `neighparam` to parameterize a neighborhood model (or any maximum likelihood model), here's what you need to do:

1. Know what model function you want to use
2. Know which values in your model function you want to parameterize
3. Have your dataset(s) collected (we'll talk more later about how to prepare them)
4. Know which probability distribution function (PDF) you want to use

3. The First Step – The Script File

The first step in preparing a simulated annealing run using `neighparam` in R is to start a script file. A script file is a text file containing a set of R commands. It allows you to run a set of commands whenever you want. You can also use it to save your work and build your run over time. After you use simulated annealing to do your model parameterization, you can archive your scripts and return to them to find out the exact conditions you used for your run.

When you are writing your script, you can try out commands in the R console until they work the way you want before putting them in the script.

You can edit R scripts in any text editor but this tutorial assumes you will edit them in R.

3.A. Creating and Editing the Script File

To start a new script, use **F**ile -> **N**ew script. Save it using **F**ile -> **S**ave. Open a saved script using **F**ile -> **O**pen script…

You can use any valid R commands in your script file. The commands do not execute until you choose to run the file.

Script files have a .R extension.

3.B. Script File Comments

A comment is text in a script file that R ignores. You put it there for the sake of humans reading the script.

It's a good idea to start your script file with a commented section that says when you wrote it and what the script is for. You can also put comments on individual lines or sections to describe what you're trying to do.

The comment character is '**#**'. Whenever it appears in a script, the rest of the line is a comment and is ignored by R.

Comment examples:

```
#################
# Here is a commented section.
# This is good for starting a script with
# descriptive information.
#################

# Read in the dataset – remember to check the path!
trees <- read.table("c:\\trees.txt")

x <- c(1,2,4,5)  # Site values
```

3.C. Running the Script

Once you have a script file, you can run either part or all of it.

To run part of it, select the part you want to run and choose **E**dit -> **R**un line or selection. You can also use this option to run just the line your cursor is on.

Run the whole script by choosing **E**dit -> Ru**n** all.

Running a script just pastes the commands into the R console. So you can mix scripts and console commands in your R session if you want to.

3.D. Examples

The `neighparam` package (indeed, all of R) is full of scripts. You can look at those in the \R\rw2011\library\neighparam\demo\ directory (assuming you have the `neighparam` package installed).

4. Writing the Model Function

4.A. What Simulated Annealing Expects from Your Model Function

Simulated annealing is a global optimization routine. In our case, we will use it to find the maximum likelihood parameter values of a model that generates predicted values from a model function, and compares them to observed data (a column of numbers from your dataset). The model is an R

function that accepts data and parameter values and calculates the set of predicted values. You write the model function yourself, which means you can use simulated annealing to parameterize any function you can write in R (which is almost any function you can think of).

R is a vector-based language, which means that it's designed to work with groups of numbers at once (more on this in a little while). The simulated annealing algorithm will expect your model to return predicted values for all the observations in the dataset in a vector. If your model asks for data from your dataset as an argument, it will receive it as a vector.

The good news is that most of the time R will handle this for you automatically. It's good to keep this in mind, though, and check your model function to make sure it does this correctly.

4.B. Function Writing in R

For further reference on writing functions in R, see the "Writing your own functions" section of the "An Introduction to R" help page. Here's a brief review.

The basic function syntax is:

```
myfun <- function(arg1, arg2) {arg1 + arg2}
```

This function adds two numbers and returns the result. Here's how it would be used in an R session:

```
> myfun(2, 5)
> [1] 7
```

You can name your function anything you want (in the above example, it's called "myfun"). The list of arguments ("arg1, arg2") is what you pass the function. Again, you can call your arguments whatever you want. Brackets enclose the body of the function, which can be any set of valid R statements. The function returns the result of the last statement in brackets (assuming it is something that can be returned).

Script files make functions much easier to write.

4.C. Mixing Data and Parameters: R is a Vector Language…

Remember that simulated annealing expects your model to be able to accept vectors of numbers and return vectors back. This is easy since R is a vector-based language. This means that R is designed to easily do calculations on groups of numbers. When R is asked to perform a calculation that mixes vectors with single values, it will recycle the single values and use them with each value in the vector.

The function we wrote above, `myfun`, above, can already handle vectors.

```
> x <- c(1, 5, 6, 8)
> myfun(x, 3)
> [1]  4  8  9 11
```

From this example, you can see that R took the vector of values in **x** and added 3 to each one.

4.D. Writing a Simple Model Function in R

Let's use the following model as an example:

$$Crown\ radius = a + b * DBH$$

This model calculates a tree's crown radius as a linear function of its DBH. Since DBH is from your dataset, you expect that it will be given to your function as a vector of numbers. *a* and *b* are single values. Your model should return a crown radius for each DBH it is passed.

Here is how you would write this function in your R script file:

```
crownrad_model <- function (a, b, DBH) {
  a + b * DBH
}
```

That's it! This function will automatically calculate a crown radius for any number of DBHs passed to it. (We will talk later about how to tell simulated annealing what to pass to your function.)

4.E. Writing More Complicated Models – Using Multiple Functions

The model function we wrote in the section above is a simple one. Most models are more complicated than that.

One way to handle more complex situations is to call other functions from within the body of your model function. (And where would we be without functions like `sqrt()` and `log()`?)

But sometimes that isn't possible. For instance, perhaps you want to use the R function `dnorm()` for your PDF, and you want the standard deviation to be a linear function of the mean of your data. But you can't make changes to `dnorm()` to make it call your function.

Simulated annealing allows you to "nest" functions; that is, to say that you want an argument for one function to come from the results of another function. If you write the function that calculates the standard deviation, simulated annealing will evaluate it first and pass the results on to `dnorm()`. (Exactly how to do this is explained in the section "Setting Up Your Run".)

You shouldn't nest functions unless you have to. Calling a function inside of your model function is easier, faster, and simpler. The most likely two cases in which you would use nesting are providing arguments to an R function you didn't write, and using the neighborhood functions.

Function nesting is central to using the neighborhood calculation functions, which we will talk about next.

4.F. The Final "Rule" of Function Writing

If it works, do it. You can bring all your R ingenuity to bear when writing functions. If you test it out and it works, you can use it even if it's different from the examples laid out here. Simulated annealing is designed to put as few restrictions on you as possible.

5. Using Neighborhood Calculations

5.A. When to Use a Neighborhood Calculation

Neighborhood calculations evaluate functions based on neighborhoods, which have "agents" within a certain radius of a point. Think of trees contributing seeds to local seed rain, but the agents can be anything you measured in the vicinity of a sample location. Models using neighborhood calculations typically have summation terms, and will look something like this:

$$R = STR * \sum_{k=1}^{N} \left(\frac{DBH_k}{30} \right)^{\beta} e^{-D * m_k{}^{\theta}}$$

This is a variation on a model used for seed dispersal. R is the seed rain at a location i. The calculation sums over all trees within a certain radius of point i. DBH_k is the DBH of the kth tree; m_k is the distance from point I to the kth tree; and D, θ, β, and STR are parameters.

The model statement has two parts: the part on the left of the summation term and the part on the right. The part on the right is the neighborhood calculation.

The `neighparam` package includes functions to do neighborhood calculations. These functions calculate what's on the right of the summation.

Your model function in R, then, is simply:

```
model <- function(STR, summed) { STR * summed }
```

You always nest neighborhood calculations (see previous section for more on nesting), so you tell simulated annealing that the argument "`summed`" is the output of a neighborhood function. In this case, we will use the all-purpose neighborhood function, "`sumneigh`". (For how to actually tell simulated annealing to use the "`sumneigh`" function for the "`summed`" argument, see the section "Setting Up Your Run". For now, we will worry just about how to set up all the parts of the model statement.)

5.B. How sumneigh Works

The function `sumneigh` searches for all eligible "agents" (neighbors) within a given radius of a point. It then evaluates a function for each of these agents, and sums the result over all the neighbors it found. Thus, each target point's result is a single value that represents the whole summation term. The `sumneigh` function then returns a vector of these summation terms, one for each of the target points.

You provide the function that `sumneigh` evaluates for each neighbor (with, again, one level of nesting allowed for this function). You also tell it how to decide which agents are eligible neighbors by giving it a maximum search radius, and optionally, allowed value ranges for columns within the neighbor dataset.

5.C. Writing the Summation Function

You provide `sumneigh` with the function that it uses to sum over neighbors. You write this function the same way you write your model function.

In our example above, you could write the summation function this way:

```
sumfun <- function (theta, beta, D, distance, DBH)  {
  ((DBH / 30)^beta) * (exp(-D * (distance ^ theta)))
}
```

But where does the value for "`distance`" come from? The `neighparam` package includes a function, "`neighdist`", and you can tell `sumneigh` to use it to get the distances.

5.D. Datasets and Eligible Neighbors

The `sumneigh` function can accept either one or two datasets (one for targets, one for neighbors). You can separate targets and neighbors when appropriate or keep them together. When you use two datasets, `sumneigh` assumes that anything having to do with neighbors is to be found in the neighbor dataset.

Sometimes you want to remove agents from consideration as possible neighbors. You can do this in one of two ways: remove them from the dataset, or set a range on allowable values for certain columns in your dataset. For instance, if you had a column called "DBH", you can set minimum and/or maximum allowed DBH values.

You tell `sumneigh` how far away from each target to look for neighbors using its `max_radius` argument. The coordinate system is X, Y based. It can be in any units, with any orientation. The only restriction is that the coordinates must all be positive numbers.

`sumneigh` also matches site codes between targets and neighbors. Only those neighbors with the same site code as the target are evaluated.

5.E. Specialized Neighborhood Functions

Neighborhood calculations in R can be slow, particularly if you have a large dataset. To help, the `neighparam` package includes some specialized neighborhood functions that are optimized for speed.

Each of these functions evaluates a particular equation. If you are able to use one of these specialized functions, they are always the better, faster choice.

6. Choosing and Setting Up the PDF

R has several probability distribution functions you can use. Here is a list (adapted from the "An Introduction to R" help page). You can find out more about any of them from R help.

Distribution	R Name	Some Additional Arguments
Beta	`dbeta`	`shape1, shape2, ncp`
Binomial	`dbinom`	`size, prob`
Cauchy	`dcauchy`	`location, scale`
Chi-squared	`dchisq`	`df, ncp`
Exponential	`dexp`	`rate`
F	`Df`	`df1, df1, ncp`
Gamma	`dgamma`	`shape, scale`
Geometric	`dgeom`	`prob`
Hypergeometric	`dhyper`	`m, n, k`
Log-normal	`dlnorm`	`meanlog, sdlog`
Logistic	`dlogis`	`location, scale`
Negative binomial	`dnbinom`	`size, prob`
Normal	`dnorm`	`mean, sd`
Poisson	`dpois`	`lambda`
Student's t	`Dt`	`df, ncp`
Uniform	`dunif`	`min, max`
Weibull	`dweibull`	`shape, scale`
Wilcoxon	`dwilcox`	`m, n`

If R cannot provide the function you need, you can write your own. It is exactly like writing your model function.

IMPORTANT NOTE
These functions can automatically calculate log values, which gives you log likelihood, but you must ask them to do so. Check for an argument called "`log`", and set it to TRUE. If you've written your own PDF, of course you must take the log yourself in your function.

7. Preparing Data

Spending a few moments thinking about the data you pass to simulated annealing can make things easier. You may need to combine several datasets into one, for example, or convert character values to numeric ones to make it easier to work with in your functions (such as making a site code into a site number).

Your datasets can contain columns of data that aren't used in the simulated annealing process. Perhaps you have a standard dataset format and you don't want to have to edit it every time. This is fine, but extra data may slow down the simulated annealing process. If you can cut your datasets to the bare minimum, do so.

Your datasets must be in the form of data frames. This way, each column of data has a name that you can use to tell simulated annealing how to use it. If you are doing neighborhood calculations,

you may be using separate datasets for your targets and neighbors. It might be easier, and certainly less ambiguous, if your datasets have distinct column names. For instance, your X and Y coordinates could be "target_x" and "target_y" in your target dataset and "neighbor_x" and "neighbor_y" in your neighbor dataset.

Once you've prepared your data, add lines to your script to import it into R. (This is of course easiest if you can place your datasets in a stable directory location.)

8. Setting Up Your Run

8.A. Setting Up the Parameters List

The parameters list is how you tell simulated annealing where to find all the data it needs. The parameters list must contain a value for every argument for every function you're using, and where to find that argument.

You can call the parameters list itself whatever you want, but the list member NAMES must match EXACTLY the arguments of the functions you're using (your model function, your PDF, and any nested functions).

Argument values can be one of three types: single values (either varied in the simulated annealing process or not), columns of data from a dataset, or functions. Assign single values and functions directly. Assign columns of data by using a character string matching the name of the column in the dataset data frame (case sensitive).

For the parameters that you are varying (more on this in the next section), the value that you assign in your parameters list is the initial value; that is, the starting point for the simulated annealing search.

You are most assured of getting the right results if names are unique: if column data names aren't the same as argument names, for example. You are asking for trouble if two functions have arguments with the same name, unless they are supposed to get the same value.

You don't need to provide values for any functions that are called from the interior of a function (for instance, calling `sqrt()` from your model). If the argument has a default that you want to use, you can leave it out of the parameter list.

Simple Example
Recall this model:

$$\textit{Crown radius} = a + b * DBH$$

Let's set up its parameters and the parameters of `dnorm`, the probability distribution function we'd like to use with it. Here's a possible R script file (don't try to run this, due to the fake dataset):

```
####################
# Excerpt from sample R script
####################
```

```
# Write out our model
crownrad_model <- function (a, b, DBH) {
  a + b * DBH
}

# Import a (fake) dataset.  It has two columns, called
# "rad" and "dbh".
trees <- read.table(file = "c:\\fakedata.txt", header=TRUE)

# Create a parameter list, which we'll call "par"
par <- list()

# The model function "crownrad_model" has three arguments,
# a, b, and DBH.  a and b are single values (just picking
# some arbitrarily for this example)
par$a <- 0.5
par$b <- 2

# The DBH argument comes from our dataset called "trees";
# it's the column called "dbh"
par$DBH <- "dbh"

# Our PDF, dnorm, takes four arguments: x, mean, sd, log.
# Put the value for each argument in par

# x is the observed radius value – the "rad" column
# in our dataset
par$x<-"rad"

# mean is our model's predicted value – use the reserved
# simulated annealing word "predicted"
par$mean<-"predicted"

# sd is standard deviation – a single constant
par$sd<-0.815585

# Have it calculate log likelihood
par$log<-TRUE

# End of script example excerpt
```

You can see from the above example that `par` contains a list member for every argument for each function we provide to simulated annealing (in this case, the model statement and the PDF).

More Complicated Example

Recall this model:

$$R = STR * \sum_{k=1}^{N} \left(\frac{DBH_k}{30} \right)^{\beta} e^{-D * m_k \theta}$$

D, θ, β, and *STR* are parameters (that is, single values). *DBH* comes from the dataset. *m* comes from the function "`neighdist`".

For this example, suppose we have two datasets. The first is the target dataset, a list of seed trap locations with the number of seeds for each. The column names are "seed_trap_x", "seed_trap_y", "seed_site_code", and "num_seeds". The other dataset is a list of parent trees. Its column names are "parent_x", "parent_y", "parent_site_code", and "dbh".

Here's a possible R script file (again, don't try to run this):

```
####################
# Excerpt from sample R script
####################

# Write out our model
model <- function(STR, summed) { STR * summed }

# Write the summation function
mysumfun <- function (theta, beta, D, distance, DBH)  {
  ((DBH / 30)^beta) * (exp(-D * (distance ^ theta)))
}

# Import fake datasets
seeds <- read.table(file = "c:\\seeddata.txt", header=TRUE)
parents <- read.table(file = "c:\\parentdata.txt", header=TRUE)

# Create a parameter list, which we'll call "par"
par <- list()

######################
# model arguments
######################

# The model has two arguments, "STR" and "summed".  STR
# is a single value.  summed comes from the neighparam
# package's sumneigh function.
par$STR <- 10.43
par$summed <- sumneigh

######################
# sumneigh arguments
######################

# Now provide all the arguments for sumneigh
# These are target and neighbor coordinates and site codes.
# They come from two different datasets, but we don't have
# to specify which dataset each comes from – sumneigh knows
# where to look
par$targetx <- "seed_trap_x"
par$targety <- "seed_trap_y"
par$neighborx <- "parent_x"
par$neighborx <- "parent_y"
par$targetsite  <- "seed_site_code"
par$neighsite  <- "parent_site_code"

# Max radius is the distance to look for neighbors.  It's
# always a single value
```

```
par$max_radius  <- 10

# Set the dataset data frames
par$target_data <- seeds
par$neighdata <- parents

# Set the summing function
par$sumfun <- mysumfun

# Value ranges – as an example, let's exclude any trees with a
# DBH < 10 or a DBH > 60.  We could have as many ranges as we
# have data columns in our neighbor dataset, and we don't
# have to have both upper and lower bounds
par$max_vals <- list(DBH = 60)
par$min_vals <- list(DBH = 10)

# That takes care of sumneigh's arguments, but we're not
# done.  We've set the function we wrote above, "mysumfun",
# as the function that sumneigh evaluates for neighbors.
# So we have to provide its arguments as well.

#######################
# mysumfun arguments
#######################
# theta, beta, and D are single values
par$theta <- 0.1
par$beta <- 2.43
par$D <- 4.1

# DBH is a column in the neighbor dataset
par$DBH <- "dbh"

# distance comes from another function, neighdist in the
# neighparam package
par$distance <- neighdist

# So by now you know what that means – since we're
# using neighdist as a nested function, we have to
# provide its arguments too.  Luckily, as we see in
# the neighdist help, we already provided them to
# sumneigh so we don't have to do it again.

#######################
# dnorm arguments (our PDF)
#######################
# Use dnorm for our PDF – it takes four arguments:
# x, mean, sd, log.  Put the value for each argument
# in par

# x is the observed radius value – the "num_seeds" column
# in our dataset
par$x<-"num_seeds"

# mean is our model's predicted value – use the reserved
# simulated annealing word "predicted"
par$mean<-"predicted"
```

```
# sd is standard deviation – a single constant
par$sd<-0.815585

# Have it calculate log likelihood
par$log<-TRUE

# End of script example excerpt
```

8.B. Setting Up Annealing Parameters and Finishing the Script

Now you are ready to decide your options for annealing and finish your script.

Varying Vs. Non-Varying Parameters and Search Boundaries

The whole point of simulated annealing is to find the optimal values for each of a set of parameters. You've just finished set up an R list with a (potentially vast) number of values, telling simulated annealing where to find the arguments for a set of functions. Some of those arguments are the ones for which we want simulated annealing to find optimal values ("varying" parameters), and some are constants ("non-varying" parameters).

Varying parameters are always single values, never vectors. You can vary a parameter for any function used – your model statement, your probability distribution function, or any functions they use.

Remember the model we've used for simple examples:

$$Crown\ radius = a + b * DBH$$

We are using simulated annealing to find the best values for *a* and *b*. They are our varying parameters. Our PDF, `dnorm`, takes a standard deviation ("sd"). In our example, we are leaving that value constant. So *sd* is a non-varying parameter.

How do we tell simulated annealing which is which?

Simulated annealing takes the varying parameters and tries to find values for them within a set of boundaries. So, simply, if bounds are provided for a parameter, it's a varying parameter. If they aren't, it's a constant.

The bounds for varying parameters, and their initial range to search, are provided in three lists. The names of the members of each list match the names in the parameter list for those values (which, of course, match the argument names in the functions in which they're called).

Note

Make sure to set your bounds to avoid math errors if possible. For example, if a varying parameter is in the denominator of a fraction, don't set its lower bound at 0; set it at 0.0001. A parameter that appears as a power shouldn't have a very high upper bound.

Simple Example

Let's finish the annealing script for the simple model. You've already seen most of this before.

Don't try to run this, due to the fake dataset. (For a very similar script that you can run, see the file "annealing1.R" in \R\rw2011\library\neighparam\demo\.)

```
####################
# Sample R annealing script
####################

# Write out our model
crownrad_model <- function (a, b, DBH) {
  a + b * DBH
}

# Import a (fake) dataset.  It has two columns, called
# "rad" and "dbh".
trees <- read.table(file = "c:\\fakedata.txt", header=TRUE)

# Create a parameter list, which we'll call "par"
par <- list()

# The model has three arguments, a, b, and DBH.  a and b are
# single values, both of which we're varying.  Set the
# initial values here (just picking some arbitrarily).
par$a <- 0.5
par$b <- 2

# The DBH argument comes from our dataset called trees,
# the column called "dbh"
par$DBH <- "dbh"

# Use dnorm for our PDF – it takes four arguments:
# x, mean, sd, log.  Put the value for each argument
# in par

# x is the observed radius value – the "rad" column
# in our dataset
par$x<-"rad"

# mean is our model's predicted value – use the reserved
# simulated annealing word "predicted"
par$mean<-"predicted"

# sd is standard deviation – a single constant
par$sd<-0.815585

# Have it calculate log likelihood
par$log<-TRUE

# Set up our list of bounds for our varying parameters,
# a and b
par_lo<-list(a = 0, b = 0)
par_hi<-list(a = 50, b = 50)
par_step<-list(a = 5, b = 10)

# Call the neighanneal function – tell it to save its
# results in a list called "results" so we can look at
# them later
```

```
results<-neighanneal(crownrad_model, par, trees, par_lo, par_hi, par_step,
dnorm, "rad")
```

9. Testing

It is important to test the various parts of your script to make sure you have set up your annealing run correctly. If you put a plus sign where you meant to put a minus sign in your model function, or typed the wrong dataset column name, the annealing run will produce output that is completely wrong.

The most thorough test (and also the most tedious) is to pick some values for your varying parameter and calculate the expected likelihood value yourself for those values, using Excel or a similar program. Then use the "`likeli`" function in the `neighparam` package to see if you get the same value that you calculated. Even if you skip this step at the beginning, you may end up coming back and doing it later if you have problems that you can't figure out.

At the very least, make sure that all your functions' math is right and that they correctly accept and return vectors.

10. More Examples

These examples show you techniques for solving certain kinds of model-writing problems. However, there are usually several ways to solve every kind of problem and if you find one that works, use it.

10.A. Using Group-Specific Parameters

Sometimes you have your datasets divided up into groups of some kind, and for a particular parameter, you want to use a different value for each group.

For example, to use our simple model:

$$\textit{Crown radius} = a + b * DBH$$

Perhaps you have three sites, 1, 2, and 3. And you have three values of b, b_1, b_2, and b_3. Your dataset has one more column called "site" with a 1, 2, or 3 for each tree (in addition to the columns "dbh" for DBH and "rad" for crown radius).

If b is non-varying, you can work with it as a vector of values. You might write your model this way:

```
model <- function (a, b, DBH, site) {
  results<-vector()
  for (i in 1:length(DBH)) {
    results[[i]]<-a + b[[site[[i]]]] * DBH[[i]]
  }
  results
}
```

and set up your parameters for your model arguments this way:

```
par$a <- 0.4
par$b <- c(1.12, 1.0, 1.5)
par$DBH <- “dbh”
par$site <- “site”
```

However, if b is varying, this method won't work because simulated annealing expects one-dimensional vectors for parameter bounds. In this case, make b into three separate parameters.

You might write your model this way:

```
model <- function (a, b1, b2, b3, DBH, site) {
  results<-vector()
  for (i in 1:length(DBH)) {
    if (identical(site[[i]], 1))
      results[[i]]<-a + b1 * DBH[[i]]
    else if (identical(site[[i]], 2))
      results[[i]]<-a + b2 * DBH[[i]]
    else
      results[[i]]<-a + b3 * DBH[[i]]
  }
  results
}
```

and set up your parameters for your model arguments this way:

```
par$a <- 0.4
par$b1 <- 1.12
par$b2 <- 1.0
par$b3 <- 1.5
par$DBH <- “dbh”
par$site <- “site”
```