A Primer on Maximum Likelihood Estimation in R Using the bbmle Package

N. T. Hobbs

R provides several tools for estimating model parameters using maximum likelihood. The package we will use was written by a prominent young ecologist, Ben Bolker from (formerly of the University of Florida, now at Guelph), who in my opinion is one of the smartest people in the world.[1] Here, I cover the essentials of using `bbmle()` to get you started using a couple of example models. This code can provide a template for the work you will do.

WHEN TO USE bbmle

If you have relatively a relatively simple non-linear model that predicts a response based on an observation and if you can assume are normally distributed (like the light and trees example), then R's `nls( )` function is easier and simpler to use than `bbmle`. Similarly, `lm( )` is vastly easier to use when you have linear models with normal data. However, we have learned that the first step in thinking about stochasticity is to thoughtfully consider how the data arise. Based on that consideration, we choose an appropriate likelihood function. The `bbmle` package is designed for cases where you can't assume a normal likelihood—so it works well for Poisson, binomial, multinomial, beta, gamma, and other distributions. Moreover, it is ideal for dynamic models where we recursively estimate the value of state variables over time. So, if you can't assume normality (or transform the data to normal) or if you have a dynamic model, then `bbmle` is the tool of choice for maximum likelihood estimation.

FITTING A STATIC MODEL

Program structure

I find it useful to think of likelihood estimation in four parts: 1) a function describing your model, 2) a function that calculates the negative log-likelihood, 3) a call to a function that does the non-linear search for the best parameter values, and 4) analysis of the results.

A function for the model

Your first task is to write a function for your model, a function that returns a vector of predictions. This vector should have the same length as the number of observations that you will use to estimate the model's parameters. I will show you some example code for a static model, the observation model for count data that we looked at in lecture. Consider the following code:

```
#estimating parameters in observation model
```

[1] I highly recommend his book--- Bolker, B. 2008. Ecological Models and Data in R. Princeton University Press, Princeton, N. J USA.

.

```
# get bbmle library

library(bbmle)
#
#Generates random counts for a calibration curve
a = -2.25
b = .60

#We want the independent variable to have global scope--
i.e. to be available to all functions.
x = sort(runif(50,1,20))

#function for model
yhat = function(a,b){
    mu = a + b*x
    mu[mu<0]=0
    return(mu)
    }

# The generating function
y = yhat(a=a,b=b)
plot(x,y, typ='l', col="red", cex.lab = 1.5, cex = 1.5,
xlab="Number Present", ylab="Number Observed",
ylim=c(0,12))
#generate data from fucntion
data=rpois(length(y),y)
points(x,data)
```

In this example, I simulated the data, which is always a good place to start to be sure that your procedure can estimate known parameter values. It also gives you a set of reasonable starting values for maximizing likelihood using numerical methods. For complex models, particularly dynamic models, getting these starting values in the "ballpark" is really important.

Take a look at how I simulated the data. I created a function `yhat( )` representing the deterministic model. To work properly with the next steps (below), this function *must* have arguments for every parameter that you want to estimate. It must return a vector or array or list of predictions that you want to compare with observations. The independent variable (x) must be declared globally. (At least, I haven't figured out any other way to do the steps below with x as an explicit argument.) Note that if the independent variables is a vector, the function `yhat` returns a vector of length = `length(x)`.

So, in the code above, we accomplished two things. We wrote a function for the deterministic model and we used it to simulate some data. We plotted the data against the model.

A function for the negative log-likelihood

The next step is to use your model function and the data to create function that returns the summed negative log-likelihood. Consider the following example:

```
#The function for the negative log likelihood
LL1 = function (a,b,y=data){
    mu = yhat(a=a,b=b)
    #get the total negative log likelihood
    loglike= -sum(dpois(y,mu,log=TRUE))
    return(loglike)
}
```

Look carefully at this code. First, we get a vector of predictions using the function for the deterministic model (`yhat`). Next, we use a a Poisson likelihood function (Why?) to get the total negative log likelihood by summing across the individual likelihoods, just as you did in your tree growth example in Excel. Our function then returns this sum.

It is also possible to forgo writing a function for your model and instead embed the model the equation directing in the likelihood function. For example:

```
LL1 = function(y,k1, sigma){
    Mhat=100*exp(-k1*CDI*t)
    return(-sum(dnorm(Mhat,M_t,sigma,log = TRUE)))
}
```

In this case, the second line of the LL1 function is the model you want to fit. However, this approach is not well suited to data simulation, so I urge you to write a function for your model.

Obtaining the mle's

The next step is a single line of code, rich with things you need to understand:

```
#The call to bbmle
m1 = mle2(minuslogl = LL1, start = list(a = -2, b = .8),
control = list(maxit = 5000))
```

Again, let's look at individual parts.

The function that we use to obtain the maximum likelihood estimates is called `mle2()`. We, assign the results of the `mle2` function to the *model object* `m1`, which we will use later in the analysis. Next, we tell the function to use the negative log-likelihood function that we wrote (`minuslogl = LL1`) to find the likelihood. The statement `start = list(a = -2, b = .8)` tells `mle2` the parameters that you want estimated and a guess at their initial values. Finally there is an optional statement `control=list(maxit=5000)`. This can include a plethora of options telling `mle` about the method to use to search for the parameters. In this case, I have set the maximum

number of iterations of the search to 5000. Most models I work with converge well before 5000 iterations (actually, before 50). The control options are a very deep topic that I will cover in a not terribly deep way. See the help for the `optim()` function which does most of the sweaty labor behind `mle2()`.

```
#The call to bbmle
m1 = mle2(minuslogl = LL1, start = list(a = -2, b = .8),
control = list(maxit = 5000))
```

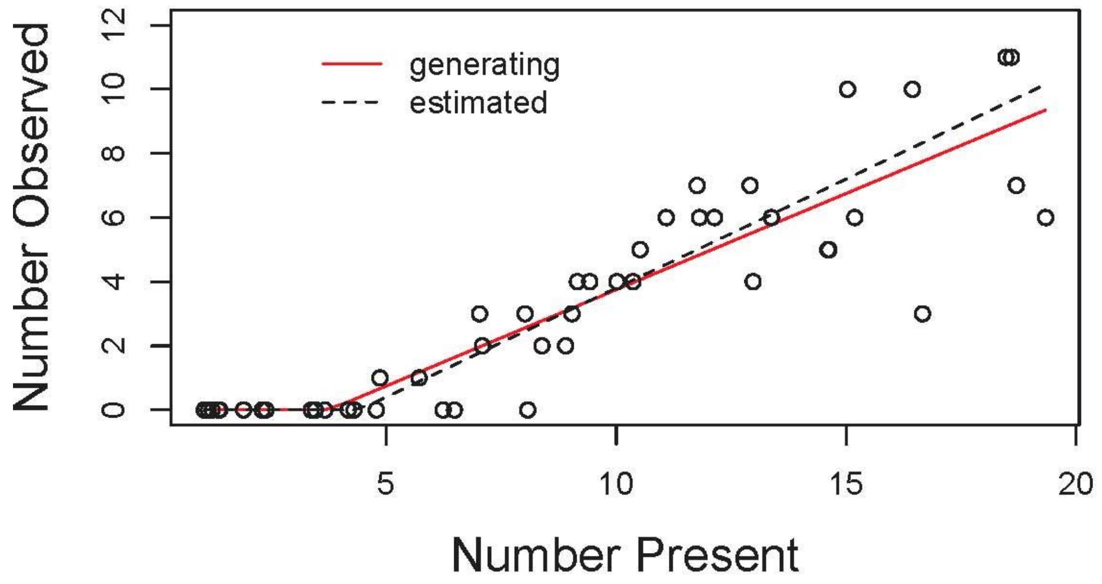
Look at the results:

Below, see code for summarizing the estimates of the coefficients and for plotting the estimates against the observations.

```
#Analysis
summary(m1)

#make a vector for plotting the esimates
y.est=numeric(length(y))
#get the estimated parameters
a.est = coef(m1)[1]
b.est = coef(m1)[2]
y.est=yhat(a=a.est,b=b.est)

#plot the fitted model agains the generating model and the
data
lines(x,y.est, lty='dashed')
legend(3,12,c("generating","estimated"), lty=c("solid",
"dashed"), bty="n")
```



generating
estimated
Number Observed
Number Present
0
2
4
6
8
10
12
5
10
15
20

Putting all of the code together:

```
#estimating parameters in observation model

# get bbmle library

library(bbmle)

#Generates random counts for a calibration curve
a = -2.25
b = .60

#We want the indepdendet variable to have global scope--
i.e. to be available to all functions.
x = sort(runif(50,1,20))

#function for model
yhat = function(a,b){
    mu = a + b*x
    mu[mu<0]=0
    return(mu)
    }

# The generating function
y = yhat(a=a,b=b)
plot(x,y, typ='l', col="red", cex.lab = 1.5, cex = 1.5,
xlab="Number Present", ylab="Number Observed",
ylim=c(0,12))
#generate data from fucntion
data=rpois(length(y),y)
points(x,data)

#The function for the negative log likelihood
LL1 = function (a,b,y=data){
    mu = yhat(a=a,b=b)
    #get the total negative log likelihood
    loglike= -sum(dpois(y,mu,log=TRUE))
    return(loglike)
}

#The call to bbmle
m1 = mle2(minuslogl = LL1, start = list(a = -2, b = .5),
control = list(maxit = 5000))


#Analysis
summary(m1)
```

```
#make a vector for plotting the esimates
y.est=numeric(length(y))
#get the estimated parameters
a.est = coef(m1)[1]
b.est = coef(m1)[2]
y.est=yhat(a=a.est,b=b.est)

#plot the fitted model against the generating model and the
data
lines(x,y.est, lty='dashed')
legend(3,12,c("generating","estimated"), lty=c("solid",
"dashed"), bty="n")
```

FITTING DYNAMIC MODELS

Program structure

Again, we think of likelihood estimation in four parts: 1) a function describing your model, 2) a function that calculates the negative log-likelihood, 3) a call to a function that does the non-linear search for the best parameter values, and 4) analysis of the results.

A function for the model

Your first task is to write a function for your model, a function that returns a vector of predictions. This vector should have the same length as the number of observations that you will use to estimate the model's parameters. Consider the following code:

```
#Logistic likelihood examples using bbmle
#Reworked on 2/02/11

# make sure that you are starting from scratch
rm(list=ls())
#set up 3 rows and columns for plots
par(mfrow=c(3,3))
par(cex.lab=1.2)

#get the Serengeti data
obs=read.csv( "C:\\Documents and Settings\\tom\\My
Documents\\Ecological Modeling Course\\2011 Spring Semester\\Labs\\Lab
6\\Serengeti data.csv")
names(obs) = tolower(names(obs))
obs=obs[,1:3]
head(obs)

# scale to N * 1000
obs$obs_n=obs$obs_n/1000
plot(obs$year,obs$obs_n)

yrs=nrow(obs)

##########################logistic model
```

```
# make a discrete logistic function.  It returns a vector of
# population estimates given initial values and parameters r and K.
logistic <- function(N0,K,r,yrs=30){
            N <- numeric(yrs)
            N[1] <- N0
            for (t in 2:yrs) N[t] = N[t-1] + N[t-1]*r*(1-(N[t-1]/K))
            return(N)
}

#use the function to find some reasonable starting values
N = logistic(N0=200,K=1400,r=.17,yrs=30)
plot(N,typ="l")
points(obs$obs_n)
```

After some preliminaries (getting the data, etc.) I created a function `logistic( )` that returns a vector of 30 estimates of population size as a function of the parameters *N0, r*, and *K*. Here are the key points: The function *must* have arguments for every parameter that you want to estimate. It must return a vector of predictions that you want to compare with observations. (This vector is exactly analogous to the column in your spreadsheet that contained the model predictions.)

Note the default value for yrs. This assures that the length of my predictions matches the length of the observations (with an important caveat, below). Note that the first thing that I did with the function was to plot the predictions against the data to find some parameter values that are reasonable starting points for the mle search What I did was to vary the arguments to `logistic( )` by looking at the data and the model predictions, and adjusting the parameters until I found a set that produced predictions roughly matching the observations `(N0=200,K=1400,r=.17,yrs=30)`. This is a very important step—you should always plot your model's predictions *before* you do any estimation. It may be that you have coded something that can't actually fit the data very well no matter what parameter values you give it. In this case, no amount of fussing with the estimation procedure will yield useful results. *Always, always, always* plot your model's prediction to assure they make sense and to obtain some reasonable initial values for parameters to be estimated. The first step in any estimation procedure is to *know your model*.

One other reason to get your initial parameters in the "ballpark" has to do with local optima. Non-linear search routines can converge to a low point in the likelihood surface (assuming we are minimizing the negative log likelihood as you will do here) that is not the true minimum. This can be particularly problematic if initial conditions are far from the global minimum.

It is also important for you to understand that you could have write a model that is much more complicated and detailed than the one I use here. It could have several parameters, covariates, and require several lines to code to specify. Writing a function for the model keeps it separate from the likelihood function, which I describe next.

A function for the negative log-likelihood

The next step is to use your model function and the data to create function that returns the summed negative log-likelihood. Consider the following example:

```
#a negative log likelihood function for the model and the data
#Assume lognormal distribution for data
#Log transformation of the data allows us to use
#a normal (Gaussian) likelihood function.
LL1 <- function (y, K, r, N0, sigma){
     N = logistic(K=K ,r=r, N0=N0 ,yrs=30)
     N[N<0]=0.001
     N=log(N[c(1,4:14,16:yrs)])  # eliminate missing data from log-
likelihood
     y=log(y[c(1,4:14,16:yrs)])
     return(-sum(dnorm(y,N,sigma,log = TRUE))) # the negative log
likelihoods: the order of N and y don't matter)
}
```

Ok, there is a lot to learn here and we will proceed step by step. First, the arguments to the function must include 1) the data (y) and the parameters to be estimated (K, r, N0, sigma):

```
LL1 <- function (y, K, r, N0, sigma)
```

As you will see later, I will hand the function a vector of the observations (y = obs$obs_n), so the y's in this function are simply the observations of population size.

Next, I create a vector of predictions (N) using the model function that I created above:

```
N = logistic(K=K ,r=r, N0=N0 ,yrs=30)
```

You need to understand the arguments for the parameter values. In this case I am using the same variable names in the likelihood function and the model function—this is why I have `K=K, r=r, N0=N0`.

In the next two lines,

```
N=log(N[c(1,4:14,16:yrs)])
y=log(y[c(1,4:14,16:yrs)])
```

I do two things. First, I log transform the data and the model predictions because I am assuming a log-normal likelihood function. If we give log-transformed data to a normal likelihood, it is the same as using a lognormal likelihood.[2] Next, I eliminate predictions and observations for which the observations are missing. Look at the data file and you will understand what I am doing here. It is tempting to use R features for dealing with missing data like na.rm = TRUE to deal with this problem, but these create problems of their own when you get to the maximization step. It is also tempting to simply eliminate the missing data when it is imported. This doesn't work because your *model* must make 30 sequential predictions, i.e., it must predict years 1961 and 1962 even if they are

[2] You could use a lognormal likelihood like dlnorm() but it requires arguments mean of log N and sd of log N.

missing from the data, in order to predict year 1964. If you don't understand this trick for dealing with missing data, ask me about it in lab.

Almost done. The last bit is the function's `return` statement:

```
return(-sum(dnorm(y,N,sigma,log = TRUE)))# the negative log likelihoods
```

This is perfectly analogous to the column in Excel that you summed to create a target cell for Solver. We are summing the individual log likelihoods. We have a normal likelihood function (remember the data transformation—we are assuming a lognormal distribution) with arguments for a model prediction, a corresponding observation, and the standard deviation. We set the argument log = TRUE so that we get the log likelihoods rather than the likelihoods. We then make the whole thing negative because `bbmle( )` minimizes the negative log-likelihood rather than maximizing the log-likelihood.

Obtaining the mle's

First, make sure that you have the library for mle open.

```
library(bbmle)
```

The next step is a single line of code, rich with things you need to understand:

```
#Now find the MLE's of the parameters
library(bbmle)
m1 = mle2(minuslogl = LL1, start = list(r = .17, K = 500, N0 = 90,
sigma=.01),data = list(y=obs$obs_n), control=list(maxit=5000))
```

Again, let's look at individual parts.

First, assign the results of the function to `m1`. This become a *model object* that we will use later. Next, we tell the function to use the negative log-likelihood function that we wrote (`minuslogl = LL1`) to find the likelihood. Initial values for the parameters are given in `start = list(r = .17, K = 500, N0 = 90, sigma=.01)`. This list tells `mle2` the parameters that you want estimated. We tell mle what data to use in `data = list(y=obs$obs_n)`.

Finally, there is an optional statement `control=list(maxit=5000)`. This can include a plethora of options telling `mle` about the method to use to search for the parameters. In this case, I have set the maximum number of iterations of the search to 5000. Most models I work with converge within well before 5000 iterations (actually, before 50). The control options are a very deep topic that I will cover in a not terribly deep way. See the help for the `optim()` function which does most of the sweaty labor behind `mle2()`.

Analysis

The analysis proceeds at two levels, analyzing the model and its parameters and analyzing the model relative to other models.

Analyze the model and its parameters

There are lots of things you can do with the m1 object you created with your call to mle2.

1) Get summary statistics and profile confidence intervals

```
summary(m1)
confint(m1)
```

2) Look at the fit of your model relative to the data. Notice how I obtain the parameter estimates from the model object.

```
#check fit against data
Nhat=numeric(yrs)
N0=coef(m1)[3]
K=coef(m1)[1]
r=coef(m1)[2]
Nhat=logistic(r=r,K=K,N0=N0)
plot(obs$year, obs$obs_n, xlab="Year", ylab = "Population
size (x 1000)")
lines(obs$year,Nhat)
```

2) The likelihood profile is obtained by varying a single parameter, finding the mle values of all parameters for each value of the varied one, and plotting the likelihood or the deviance.

```
#plot the likelihood profile
#y axis is sqrt(deviance) = sqrt(2*(-log likelihood))
plot(profile(m1))
```

Analyze the model relative to other models

One of the nicest features of `bbmle` is its ability to compare models, which we will do later in the course. But for those of you familiar with information theoretics, consider the following:

```
AICctab(m1,m2, nobs=27, sort=TRUE, delta=TRUE,
weights=TRUE)
```

Presume you have two model objects m1 and m2 created by the logistic model with variable K. AICctab does what you might think—it constructs a table of AICc values, Δ_i's, and and Akaike weights for all of the models you give it. In this case there are only 2, but you can include as many model objects as you like. Notice that to use AICc, the version for small samples, you must provide an argument specifying the sample size

(`nobs =27`). If you want AIC's or BIC's (as opposed to AICc's) then use, duh, `AICtab( )` or `BICtab( )` with the same arguments but omitting the `nobs =`.

All of the IC in bbmle tables calculate the k term [i.e., the k in $-2(\log L(\theta) - 2k)$] as the number of parameters in the model. Strictly speaking it should include the standard deviation as well, because it is calculated from the data. Bolker assures me this doesn't matter, that the results will be the same regardless. However, I am not sure about that and the fastidious side of me wants it to be right. So here is how you do it:

```
attr(m1,"df") = 4
attr(m2,"df") = 5
AICctab(m1,m2.1, nobs=27, sort=TRUE, delta=TRUE, weights=TRUE)
```

Some problems and potential fixes

As I told you earlier, searching for the mle is a bit of an art. You are bound to occasionally encounter, shall I say, *difficulties*, getting reliable estimates. Maximum likelihood is an extremely powerful and flexible approach to parameter estimation, and power and flexibility never come free. They are paid for with suffering. Here are a few problems that are likely. You will discover others. (When you do, please tell me how you fixed them!)

1) *Failure to converge*. Increase the number of iterations using `maxit =`. You can also increase the threshold for convergence using `reltol =`, e.g.,

```
control=list(maxit=5000, reltol=1e-4))
```

2) If you get the message

```
error:

Error in optim(par = start, fn = objectivefunction, method
= method, hessian = !skip.hessian,  :
  initial value in 'vmmin' is not finite
```

don't be alarmed. All it likely means is that you like have missing values in your data. Deal with them as I showed you above. You also may have NaN's, which I describe below or, you may be trying to estimate a parameter that is just too small. See point 10, below.

3) *The most cryptic error message in the world:* If you get the error

```
Error in inherits(minuslogl, "formula") :
  argument "minuslogl" is missing, with no default
```

it means that you have put a one at the end of minuslogl instead of a lowercase L. This is astonishingly easy to do.

4) *System is exactly singular:* If you get the dreaded error

```
Error in solve.default(oout$hessian) :
  Lapack routine dgesv: system is exactly singular
Warning message:
In mle2(minuslogl = LL1, start = list(r = 0.25, K = 3000,
B0 = 10000,  :
  couldn't invert Hessian
```

then you have some work ahead. The Hessian is a matrix that must be inverted to properly estimate stochasticities associated with your parameter estimates, which is one of the reasons that Bayesian methods based on MCMC often work when likelihood methods fail utterly. There are several things you can try to get around this problem, but in my experience it sometimes means you have an ill-formed model or bad initial conditions. I urge you to take a look at pages 387-398 in Ben Bolker's book (Bolker, B. In Press. Ecological Models and Data in R. Princeton University Press, Princeton, N. J USA) to get a flavor of the kinds of efforts you need to go through to fix this problem. As a last resort, you can get parameter estimates and AIC values by setting `skip.hessian = TRUE,` as shown below:

```
m1 = mle2(minuslogl = LL1,
     start = list(r = .25, K = 3000, B0 = 10000, q=.0004),
     data = list(C.obs=C.obs, tend = tend),
     skip.hessian=TRUE, control=list(maxit=50),
     trace=TRUE)
```

You can then try using the estimates of the parameters to rescale the numerical search using the `parscale` statement. This trick is below and it is vital to success with the Ricker problem in your assignment.

5) *mle2() is not as smart as you are*: Occasionally, mle2 will return estimates that are the same as those you gave it, or are only different in one or two parameters. This can occur with *no error message*. You may simply need more iterations or you may have an ill-formed model, that is, one that contains parameters that are not identifiable. Basically this means that you can get the exact same solution, or virtually the same solution, with many combinations of the parameters.

6) *Unreasonable estimates of parameters:* You may get a nice fit to the model, but your parameter estimates exceed biological meaningful bounds. For example, you might get a survival rate that exceeds 1. To restrict the range of parameters to values that are realistic, use code like this:

```
start = list(b0 = 3, b1 = -.0025,
       s1=.75,s2=.90,s3=.75,s4=.90,
       N1=65, N2 = 184, N3 = 65, N4 = 184, m=.5,
```

```
        a1 = .26)

 m3 = mle2(minuslogl = LL1, start = start,
skip.hessian = FALSE, trace=TRUE,
  method="L-BFGS-B",
  upper= c(4, 0, .70, .99, .70, .95, 130, 260, 130, 260, .55, .35),
  lower = c(1, -.01, .3, .85, .3, .85, 20, 20, 20, 20, .45, .25),
  control=list(maxit=50, trace=TRUE, parscale=abs(unlist(start))))
```

Notice that I have specified `method ="L-BFGS-B"`. This allows you to specify upper and lower bounds for each of the parameters in the vectors, `upper and lower`. Note that the elements of these vectors must be in exactly the same order as you parameter list (which makes sense) *and in the argument list to your likelihood function* (which, I know, doesn't make sense. Get over it.)

When your model converges, be sure you check to see if parameters bump against constraints. If they do, you will probably want to change their ranges, at least those that can be changed. There is a bit of an art to using this method---you may need to experiment with different combinations of constraints to get good results.

Note that this method will not work if the optimization produces any NaN's, which is really not hard to do, particularly if your dataset contains any outliers. Dealing with those is the next topic.

7) *Those pesky NaN's.* NaN's arise whenever your code produces mathematically undefined results like, for example, the log of zero. If you get a few of these and an associated warning from R, it is not a big deal unless you are using the `method ="L-BFGS-B"` or unless you are compulsively tidy. If you have model predictions that differ enormously from the data, which can happen when you have an outlier or if you give the optimization poor starting values (remember, always find those using "fit by eye"), then the likelihood can be 0. This makes sense, but the log of 0 is NaN. There are several things you can do.

-Check to be sure that no quantities are going negative (e.g `N[N<0]=0.001`).

- You can also check to be sure that your log likelihood statement returns finite values in code something like

```
sum_log_like = 0
log_like=numeric(length(N))
#note that missing values have already been removed from N and y
      for(i in 1:length(N)){
            log_like[i] = -dnorm(y[i], N[i],sigma,log = TRUE)
      if(is.finite(log_like[i])) sum_log_like = sum_log_like +
      log_like[i]
      }
return(sum_log_like)
```

In this case, you are simply excluding the NaN's from the sum of the likelihood, which is the same as assuming their likelihoods are 0.

9) *Finding the global minimum.* As I described above and in lecture, it is quite possible that `bbmle()` will happily report that it has converged on a minimum that is not the global minimum. In this case, the answer you get depends on the initial conditions. This is not the software's fault—it has done exactly what you told it to do—starting at the initial conditions, search the area around them until a minimum is found with acceptable tolerance. There are two ways this can go wrong. The solution space may be very flat, which means that many different combinations of parameter values give very similar likelihoods. The second problem occurs when the likelihood surface (think of a 3 dimensional plot with 2 parameters as the x and y axes and the likelihood as the z) is very complex with lots of hills and valleys, the search can get stuck on the top of a local hill (if we are maximizing the likelihood) or stuck in a local valley (if we are minimizing the negative likelihood). This is the problem of a local minimum. In either case, you find that you get different results when you use different initial conditions.

The first of these problems (flat solutions) can be solved by adjusting the way the algorithm searches. The optimization methods implemented by mle2 are performed by the R function `optim( )`. It is valuable to take a look at the help on this function. The `optim()` function approximates derivatives using methods resembling those we learned earlier (Eulers and Taylor series). Avoiding the gory details, the accuracy of this approximation depends (as before) on Δx, that is the step size. You can adjust the step size using a control argument to `optim( )`. Consider the following code:

```
start=list(N0=200,B1=.15,K=2000,r=.16, sigma=.01)
##############first call to mle2
m2 = mle2(minuslogl = LL1, start = start, data = list(y=obs$obs_n),
      skip.hessian=FALSE,  control=list(trace=TRUE, maxit=5000))
summary(m2)
#confint(m2)

start=list(N0=200,B1=.15,K=2000,r=.16, sigma=.01)


##############second call to mle2, notice adjusted step size in
parscale( ) using results of first model object, m2.
m22 = mle2(minuslogl = LL1, start = start,      data =
list(y=obs$obs_n),
      skip.hessian=FALSE,  control=list(trace=TRUE, maxit=5000,
      parscale=abs(coef(m2))))
summary(m22)
```

By changing the initial conditions for the first call to mle2, you discover to your enormous distress that the estimates you get are subtly dependent on the initial conditions specified in the list `start`. Moreover, your attempt to get confidence intervals and likelihood profiles causes all sorts of output from R that resembles swearing. However, you are probably very close a true maximum (or minimum). So, in a second call, you scale the step size to the magnitude of the parameters that you discovered in the first call, i.e., the bit of code that reads: `parscale=abs(coef(m2))`. You will very likely find

that the parameter estimates you get from the second call to mle are not sensitive to the initial conditions (unless you give it totally unreasonable values—remember, you discovered you starting point by plotting the data, right?)

The problem of a very complex likelihood surface can easily arise with models of high dimension (i.e., many parameters) and if you have too many parameters relative to the data, well, you *deserve* to suffer. The only sure fix in this case is to embed your call to `mle2( )` in a series of nested `for` loops where you iterate over a set of values for the initial conditions for each parameter, storing the parameter values and the associated negative log likelihood for each iteration in an array. You then search the array for the minimum negative log likelihood. This is a bit tedious and because of that, I won't cover it in the course. But it is a brute force, if time consuming, approach to assuring that your minimum is global.

10) *Never try to estimate really small things*.

Consider the following code:

```
#The first version of the Ricker model with Rainfall
> ##########################Ricker model
> # make a  Ricker function where the carrying capacity is
#modified by rainfall. x[t]is the normalized rainfall for
#the current years growing season (just before census) it
#is 0 when = mean rainfall, negative when below it, and
#positive when above it.
Ricker2 <- function(N0,B2, K,r, x=norm.rain,yrs=30){
 N <- numeric(yrs)
N[1] <- N0
for (t in 2:yrs) N[t] = N[t-1]*exp(r-r/K*N[t-1]+B2*x[t])
return(N)
 }

#The second version:
Ricker2 <- function(N0,B1,B2r, x=norm.rain,yrs=30){
 N <- numeric(yrs)
N[1] <- N0
for (t in 2:yrs) N[t] = N[t-1]*exp(r-B1*N[t-1]+B2*x[t])
return(N)
 }
```

These two models are algebraically identical where $B_1 = \frac{r}{K}$. However, the first version fits nicely, where the second version returns infinite values and never produces an estimate. This is because B_1 is a tiny, tiny number. The numerical methods used here

simply cannot estimate it. There are many tricks for avoiding very small values, changing parameters like this is one of them, rescaling the data is another.