# Modeling the dynamics of diversification through time

Dan Rabosky

August 2, 2011

**Abstract**

Here we will cover some basic aspects of diversification modeling in R . In the first part, we'll cover some basic strategies to build and analyze likelihood models of speciation and extinction through time. We'll also talk about some numerical tools that will be useful for analyzing models like these. This should give you a "custom" likelihood toolkit for analyzing the tempo of speciation through time.

In the second part, we will use multi-state models of diversification and character evolution in diversitree to study the temporal dynamics of speciation in Caribbean anoles.

## 1 Building and analyzing models of speciation and extinction through time

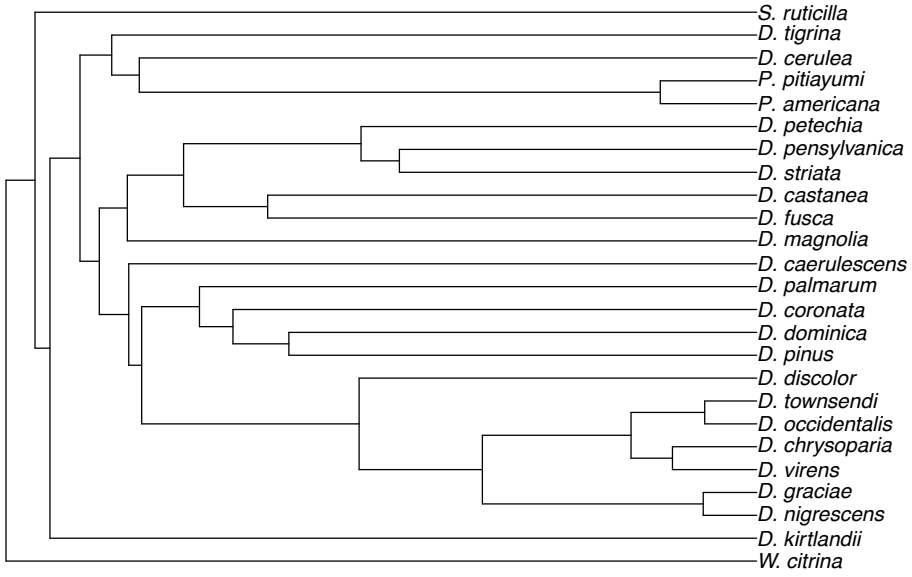
Here we are going to focus on a deceptively simple question: given that we have a time-calibrated molecular phylogeny, what can we say about the tempo and mode of speciation and extinction through time? In this section, we are going to focus on an analysis of wood warblers (formerly in the genus *Dendroica*, but now placed in *Setophaga*).

We are going to start by reading in a time-calibrated phylogeny of wood-warblers. Make sure you have the file `Dendroica_phylogeny.tre` in your working directory.

```
> library(ape)
> v <- read.tree("Dendroica_phylogeny.tre")
```

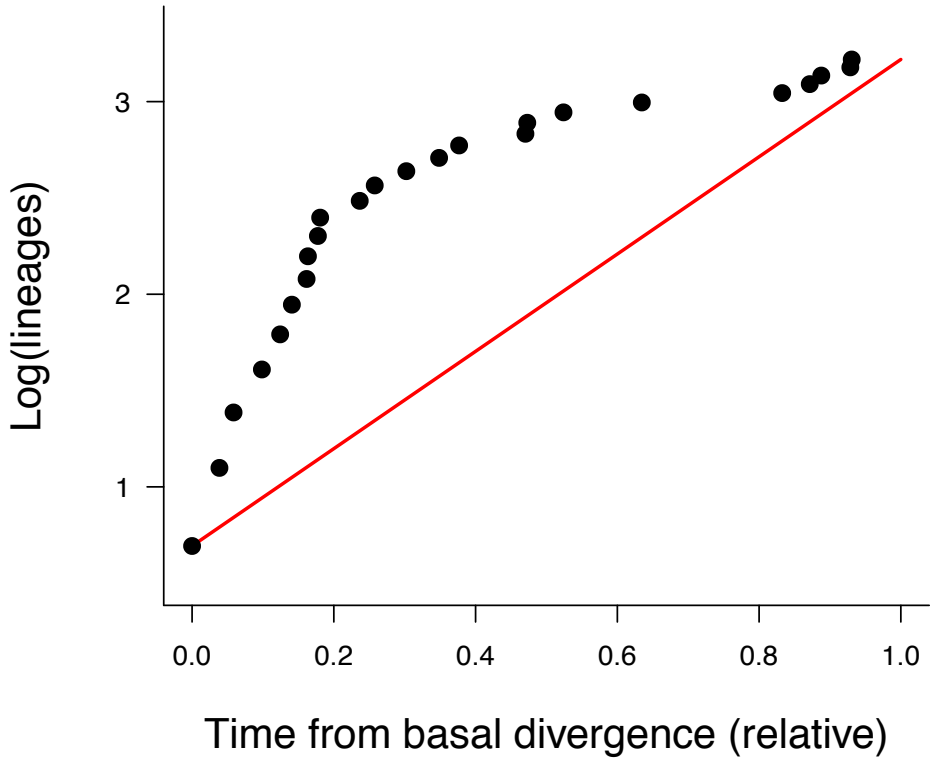
Let's look at the Dendroica phylogeny:

```
> plot.phylo(v, no.margin = TRUE)
```




You can pretty easily see what looks like a cluster of speciation events at the base of the tree. We are going to visualize the tempo of lineage accumulation through time using a lineage-through-time (LTT) plot, plotting the cumulative rise in the (log-transformed) number of lineages as a function of the time since the basal speciation event. Here we'll use a little function from the utility file `Rabosky_SB_functions.R`, which you should have in your working directory.

```
> source("Rabosky_SB_functions.R")
> nicerLttPlot(v)
```




Here, the red line is the 'pure birth' expectation: this is what the lineage-through-time curve should look like under a model with constant speciation through time and no extinction. Actually, the slope of the LTT plot (on a semilog scale) is an estimate of the instantaneous rate of speciation at any point in time. So - what you see here is that speciation rates in warblers - as a whole - appear to have decelerated through time. This is one interpretation of a "concave down" LTT plot.

We can quantify the extent to which nodes are 'clustered' towards the base of the tree using the $\gamma$-statistic (Pybus and Harvey 2000):

```
> gammaStat(v)
```

```
[1] -3.480495
```

Under a simple "pure birth" model, with a single constant rate of speciation through time, the distribution of $\gamma$ is standard normal: only 5% of phylogenies simulated under this model should have a value more negative than - 1.64. A negative value means that there are too many nodes near the base of the tree relative to the pure birth model. (Conversely, a positive gamma would mean too many nodes near the tips relative to the pure-birth model). The $\gamma$ we observe is much more negative than this. In fact, we can compute the one-tailed probability of this value under the time-constant pure-birth null hypothesis:

```
> x <- gammaStat(v)
> pnorm(x)
```

```
[1] 0.0002502438
```

where `pnorm` gives the cumulative area of the standard normal distribution lying to the left of ("less than") the observed Z-score of -3.48. This is obviously an unlikely value under a constant-speciation model.

# 2 Your own likelihood function

Here we will use a little math to build a function that will calculate the likelihood of the wood-warbler phylogeny under different models of speciation and extinction. We'll start with a simple "pure birth" model, where all lineages have the same rate of speciation through time. Then we'll extend this to more complicated models.

Model stuff here....

After all that, we end up with a likelihood function for the observed data. Given that we have a speciation rate $\lambda$, and a vector of "branching times", or $\mathbf{X}$ . Thus, $\mathbf{X_1}$ is the initial branching-time, or the root-to-tip distance of an ultrametric tree. The likelihood of $\lambda$, conditional on the data is

$$L(\lambda|data) = \lambda^{n-2} e^{-2\lambda X_1} \prod_{i=2}^{n-1} e^{-\lambda X_i} \tag{1}$$

or, taking the log of both sides:

$$Log(L) = (n-2)Log(\lambda) - 2\lambda X_1 - \sum_{i=2}^{n-1} \lambda X_i \tag{2}$$

So let's use this model to write a simple likelihood function. We'll write a function that takes two arguments: the `branching times` from the phylogeny, and a speciation rate, lambda.

```
> pureBirthLikelihood <- function(lambda, x) {
+     x <- rev(sort(as.numeric(x)))
+     Ntaxa <- length(x) + 1
+     lh <- (Ntaxa - 2) * log(lambda)
+     lh <- lh - 2 * lambda * x[1]
+     lh <- lh - sum(lambda * x[2:length(x)])
+     return(lh)
+ }
```

Let's try this function:

```
> bt <- rev(sort(as.numeric(branching.times(v))))
> pureBirthLikelihood(lambda = 0.1, x = bt)

[1] -60.84989

> pureBirthLikelihood(lambda = 0.2, x = bt)
```

```
[1] -52.79793
```

```
> pureBirthLikelihood(lambda = 0.3, x = bt)
```

```
[1] -51.36267
```

Great! We now have a function that can compute the likelihood of the warbler phylogeny under a simple pure-birth model of speciation. How do we go about finding the maximum likelihood estimate of the speciation rate under this model? We also want to know the likelihood of the data at the maximum, which we'll use for model-based comparisons a little bit later. A common-sense solution here is what I'll call the `bounded grid` search. We'll just try a bunch of values and find the maximum.

```
> intervals <- 30
> lambdavec <- seq(0.001, 5, length.out = intervals)
> lhvec <- numeric(intervals)
> for (i in 1:intervals) {
+     lhvec[i] <- pureBirthLikelihood(lambdavec[i], x = bt)
+ }
> lambdavec[lhvec == max(lhvec)]
```

```
[1] 0.3457586
```

```
> max(lhvec)
```

```
[1] -51.70818
```

So, let's repeat this using a denser bit of sampling. Repeat the same analysis above, but let's set the interval of sampling to 1000. The maximum likelihood estimate:

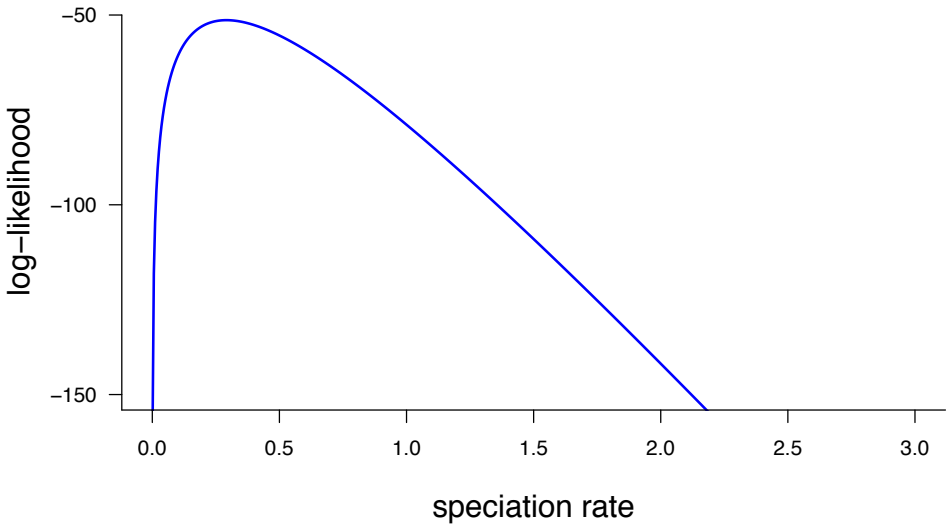
```
[1] 0.2912322
```

and the log-likelihood at the maximum:

```
[1] -51.35307
```

and let's plot the likelihood surface. I'll do this the 'long' way for illustration, not using the function `plot`. Your graphics in R will be much, much more powerful if you learn to build figures without using the "canned" functions, especially `plot` , `barplot` , and related functions.

```
> plot.new()
> par(mar = c(6, 6, 1, 1))
> plot.window(xlim = c(0, 3), y = c(-150, -48))
> lines(x = lambdavec, y = lhvec, lwd = 2, col = "blue")
> axis(1, at = seq(-0.5, 5, by = 0.5))
> axis(2, at = seq(-200, -50, by = 50), las = 1)
> mtext(side = 1, text = "speciation rate", line = 3.5, cex = 1.5)
> mtext(side = 2, text = "log-likelihood", line = 3.5, cex = 1.5)
```




We can clearly see a peak at the maximum (0.29 or so). Make sure you can replicate these results. This graphical approach is useful for visualization, but does not take advantage of some really powerful numerical tools that you can use. Moreover, for more complicated likelihood functions - especially with more than 1 or 2 parameters - the graphical or grid approach will simply be intractable or impossible.

# 3 One dimensional optimization

The above analyses use a pretty crude way of tackling the problem of optimization. Fortunately, R has lots of handy built-in functions that make this pretty straightforward. (actually, we could have just computed the analytical solution to the problem for the likelihood model above). But now we will focus on one of the most powerful numerical techniques you can use in R: optimization. R has a number of functions that will take a function and find the values of the parameters that maximize the value of the function. How convenient! For simple 1-parameter problems, we can use the bounded search algorithm available through the function `optimize`. Here, we just specify the minimum and maximum bounds on the parameter search space and use the likelihood function itself as an argument to `optimize`:

```
> optimize(pureBirthLikelihood, lower = 0, upper = 5, x = bt, maximum = T)

$maximum
[1] 0.2914885

$objective
[1] -51.35306
```

Very useful! And it gives us both the likelihood at the max and the ML estimate of the speciation rate! Note that there are arguments that control the bounds on the parameter search space, as well as whether we want to maximize (versus minimize) the

value of the function.

Also, for the heck of it, we will estimate this using laser as well. The "pure birth" function in laser is just `pureBirth` , not to be confused with the `pureBirthLikelihood` function we just wrote.

```
> library(laser)
> pureBirth(bt)
```

```
$LH
[1] 3.431671

$aic
[1] -4.863343

$r1
[1] 0.2914923
```

Now, the ML estimate of speciation should be identical to ours, but the likelihoods are off. This is because the calculations in laser include a constant term that changes the absolute magnitude of the likelihoods. Specifically, the likelihoods in laser will be higher by a factor of $log(1:(n-1))$. We can check this:

```
> Ntaxa <- length(bt) + 1
> pureBirth(bt)$LH - sum(log(1:(Ntaxa - 1)))
```

```
[1] -51.35306
```

This should agree with the likelihood we got from `optimize` . Whether this term is included or not makes no difference, **as long as you only compare models that consistently include or exclude it** .

# 4 Time-dependent models with speciation

Here we'll build a simple likelihood function to evaluate the likelihood of any time-varying model of speciation through time. We are now going to make a very simple (but potentially confusing) terminological shift. Rather than work with *branching times* , we will work with *speciation times* measured such that the basal speciation event occurred at time t = 0. And thus, the present will correspond to time T. This will make it easy to understand the output of our time-dependent speciation models. In the general case, the log-likelihood of the speciation data is:

$$Log(L) = Log(\sum \lambda(t)) - 2\int \lambda(t)dt - \sum \int \lambda(t)dt \tag{3}$$

where each integral $\int \lambda(t)dt$ is evaluated from the time of the speciation event to the present, or

$$\int_{t_i}^{T} \lambda(t)dt \tag{4}$$

where $t_i$ is the i'th speciation time.

OK. Now to implement this. Here, we will use a linear model of speciation through time, such that

$$\lambda(t) = \lambda_0 + tk \tag{5}$$

such that $\lambda_0$ is the speciation rate at the start of the radiation, and k is the "slope" of the rate change through time. This is a very simple and analytically tractable model. One nice thing about *simple* models like this is that we can actually compute the value of the integral analytically. But here, we are not going to do this. We are going *numerically integrate* the speciation function.

To do this, we are going to do something a little non-intuitive. We are going to create special functions for speciation and extinction through time. In both cases, we want a function that accepts 2 arguments:

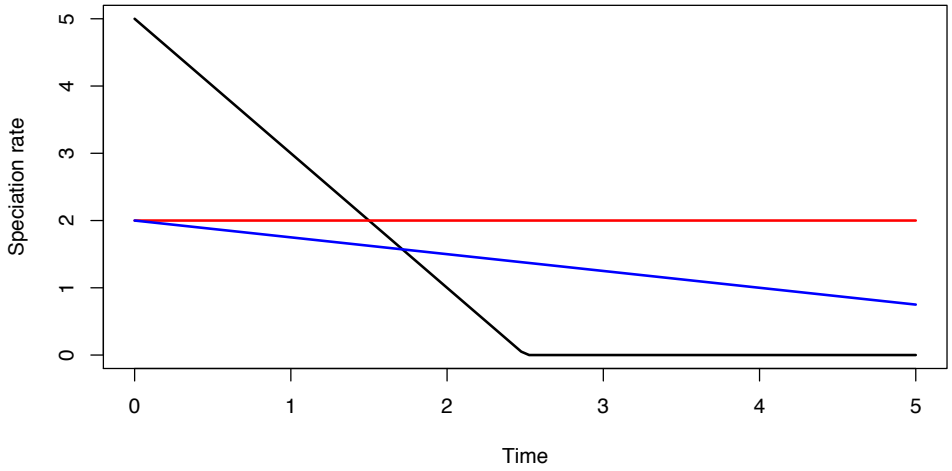
- time.
- a vector of length 2 with parameters that describe how speciation and extinction change through time.

Here is such a function:

```
> lambdaFx <- function(time, pars) {
+     srate <- pars[1] + time * pars[2]
+     srate[srate < 0] <- 0
+     return(srate)
+ }
```

The argument `pars[1]` is the intercept, or the rate at the start of the radiation; and `pars[2]` is the slope of the rate change. If the rate drops below 0, we set it to 0, thus preventing negative speciation rates. To get a feeling for this function, let's plot a rate trajectory through time:

```
> timevec <- seq(0, 5, length.out = 100)
> rates1 <- lambdaFx(timevec, c(5, -2))
> rates2 <- lambdaFx(timevec, c(2, 0))
> rates3 <- lambdaFx(timevec, c(2, -0.25))
> plot(rates1 ~ timevec, type = "l", lwd = 2, col = "black", xlab = "Time",
+     ylab = "Speciation rate")
> lines(timevec, rates2, lwd = 2, col = "red")
> lines(timevec, rates3, lwd = 2, col = "blue")
```




Now we'll use a new function from the `Rabosky_SB_Functions.R` file that will actually create a likelihood function for us for arbitrary models of time-dependent speciation and extinction. As an argument, this function will take 2 items:

- The branching times, or `bt`
- The speciation function we just created (or any other 2 parameter speciation model)

Now we build a likelihood function for a linear time-dependent model of speciation-through-time:

```
> source("Rabosky_SB_Functions.R")
> lfx.linear <- buildLikelihoodFunction.purebirth(bt, lambdaFx)
```

And you can have a look 'under the hood' at the function. Now, the function already contains our data. It was "built" to compute the likelihood of the warbler data under a 2 parameter linear model of speciation through time. We can play around with this:

```
> params <- c(0.5, -0.1)
> lfx.linear(params)

[1] -48.74319

> lfx.linear(runif(2))

[1] -252.163
```

And we can check to make sure that we get identical results to the pure birth model. To calculate the likelihood using `lfx.linear` under a constant-rate model, just remember that we can feed in a speciation rate as `par[1]` and set `par[2] <- 0`

```
> res.purebirth <- optimize(pureBirthLikelihood, lower = 0, upper = 5,
+     x = bt, maximum = T)
> res.purebirth
```

```
$maximum
[1] 0.2914885

$objective
[1] -51.35306
```

```
> srate <- res.purebirth$maximum
> pars <- c(srate, 0)
> lfx.linear(pars)
```

```
[1] -51.35306
```

# 5 Optimization in two or more dimensions

We will now use another optimization function to maximize the likelihood of our function, `lfx.linear` . This function is `optim` , R's multiparameter workhorse optimization function (although there are others). To use `optim` , we will make a "wrapper" function for lfx.linear. There a number of advantages to this:

- You can have much greater control of the optimization through parameter transformation
- you can customize output and watch the progress of the optimization

Here's such a "wrapper" function:

```
> optimFx <- function(pars, myFunction, verbose = FALSE) {
+     loglik <- myFunction(pars)
+     if (verbose) {
+         cat(loglik, "\n")
+     }
+     return(loglik)
+ }
```

Now let's try this function with random starting parameters:

```
> set.seed(1)
> pars.initial <- runif(2)
> cl <- list(maxit = 1000, fnscale = -1)
> res <- optim(pars.initial, optimFx, myFunction = lfx.linear,
+     method = "Nelder", control = cl)
> res
```

```
$par
[1]  0.6515210 -0.1179818

$value
[1] -46.47292
```

```
$counts
function gradient
      95       NA

$convergence
[1] 0

$message
NULL
```

There are a lot of things to learn about optim. Here are some of them:

- Key arguments to option: initial parameters, your function name, the algorithm to use, and control parameters
- By default, `optim` will find the *minimum* value of a function. This is NOT what we want. To address this, we set one of the control parameters to shift it to a *maximization* problem. We do this with `control` argument that gets passed to `optim`. We set $fnscale = -1$ to turn this into a maximization problem.
- The algorithm. Just use "Nelder" for the moment. It is quite robust and reasonably fast. THere are strengths and weaknesses of each algorithm. The other useful one is "L-BFGS-B", which lets you impose upper and lower bounds on parameter search space.
- Now for output: `res$value` is the log-likelihood at the maximum
- `res$par` contains the ML parameter estimates
- `res$counts` and `res$convergence` contain critical information about whether the algorithm actually converged on a meaningful solution. If the convergence value is not == 0, there is a problem!!

Lots of stuff to learn about optimization, but it is super-useful! Now let's try it on your own:

- Try running optim with optimFx and this set of initial parameters. What happens?

```
> pars.initial <- c(0.1680415, 0.8075164)
> optim(pars.initial, optimFx, myFunction = lfx.linear, method = "Nelder",
+     control = cl)
```

OK - those parameters should have generated an error message. This brings us to a very common problem in optimization studies: we often want to repeat our optimization many times, because it is always possible that our optimization is finding a local (rather than global) optimum. This is a common problem in phylogenetic analyses, and the situation is no different here. So, we should try running such optimizations many times and with different initial starting values.

But, as you've just seen, some initial parameter values can lead to an error in the

likelihood calculation. R has a very useful function for dealing with errors during optimization (and during everything else): `try`. The function `try` will 'try' an argument, and if there is an error, it will just return an error message without actually causing your calculations to choke. This isn't too important for what we are doing now, but will be very important a little later on. Now we'll repeat our optimization, but putting the call to `optim` in a `try` function.

```
> pars.initial <- c(0.1680415, 0.8075164)
> cl <- list(maxit = 1000, fnscale = -1)
> res <- try(optim(pars.initial, optimFx, myFunction = lfx.linear,
+     method = "Nelder", control = cl))
```

So - you used the same initial parameter values, but what actually happened? You didn't get an error message. Let's see what value is stored in res:

```
> res
```

```
[1] "Error in integrate(fx, t1, t2) : the integral is probably divergent\n"
attr(,"class")
[1] "try-error"
```

In this case, optim actually choked. But when the `try` function received the error message from optim, it stored the error message itself within `res` . When you include a function call within `try` , it returns an object of class "try-error". This lets you know that the calculation failed, but it won't choke your analysis.

This may seem a bit abstract, but to make good inferences for these types of optimization problems, we should be doing our analyses lots of times with different random starting parameters. Let's try this. We are going to do 20 independent optimizations of the linear time-dependent model for the warblers, using different starting parameters. We'll store the results in each case. For "initial parameters", we'll choose random numbers from a uniform (0, 10) distribution. And we'll use 'try' with `optim`

```
> N_optimizations <- 20
> best_params <- matrix(0, nrow = N_optimizations, ncol = 2)
> likevec <- numeric(N_optimizations)
> for (i in 1:N_optimizations) {
+     pars.init <- runif(2, 0, 10)
+     temp <- try(optim(pars.init, optimFx, myFunction = lfx.linear,
+         method = "Nelder", control = cl))
+     best_params[i, ] <- temp$par
+     likevec[i] <- temp$value
+ }
```

What happened? Now we'll try the same thing again, but we'll explicitly put our call to `optim` in a `try` wrapper. And we'll use a `while` loop so that if our initial parameters fail, we'll just keep trying new ones until the optimization DOES NOT FAIL!

```
> N_optimizations <- 10
> best_params <- matrix(0, nrow = N_optimizations, ncol = 2)
> likevec <- numeric(N_optimizations)
> for (i in 1:N_optimizations) {
+     pars.init <- runif(2, 0, 10)
+     temp <- try(optim(pars.init, optimFx, myFunction = lfx.linear,
+         method = "Nelder", control = cl))
+     while (class(temp) == "try-error") {
+         pars.init <- runif(2, 0, 10)
+         temp <- try(optim(pars.init, optimFx, myFunction = lfx.linear,
+             method = "Nelder", control = cl))
+     }
+     best_params[i, ] <- temp$par
+     likevec[i] <- temp$value
+ }
```

My personal preference is to make a single general-purpose wrapper for optim that will choose random starting parameters and evaluate `optim` within a try wrapper. For example:

```
> doOptimization <- function(likefunc, nparams, pmin, pmax) {
+     cl <- list(maxit = 1000, fnscale = -1)
+     pars.init <- runif(nparams, pmin, pmax)
+     res <- try(optim(pars.init, fn = likefunc, method = "Nelder",
+         control = cl))
+     while (class(temp) == "try-error") {
+         pars.init <- runif(nparams, pmin, pmax)
+         res <- try(optim(pars.init, fn = likefunc, method = "Nelder",
+             control = cl))
+     }
+     return(temp)
+ }
```

There are all sorts of other things to learn about optimization, but this is probably enough to get you started.

# 6 Challenge problems: optimization section

- Try running `doOptimization` on the warbler linear likelihood function. Note that you have to feed the function a number of arguments: `nparams` is the number of parameters in your likelihood function; `pmin` is the minimum value of the parameters, and `pmax` is the max param value.

- Plot the maximum likelihood estimate of the (linear) speciation rate-through-time curve. Hint: you have have the ML parameter estimates and a function that transforms those parameters into rates....

- Now for a challenge: build a speciation function to describe an exponential change in the speciation rate through time, and fit the model to the wood-warbler data (just as we have done for the linear model). Here, I want you to assume the speciation rate through time can be modeled as

$$\lambda(t) = \lambda_0 e^{-kt} \tag{6}$$

  So $k$ is a parameter that controls the magnitude of rate-decline through time, and $\lambda_0$ is the initial speciation rate at time $t = 0$. This shouldn't involve much more than building a new speciation-through-time function and using it with the optimization tools we have already developed.

- Using different sets of initial paramers, fit the exponential-decline function to the warbler data - until you are satisfied that you are converging on a stable result. Try this perhaps 10-20 times, keeping track of both the likelihoods of each optimization as well as the estimated parameter values. Do you find any evidence for multiple optima?

- Now compute the AIC scores for each of the 3 models you've used. Remember that the AIC is just

$$AIC = (-2LogLik) + 2k \tag{7}$$

  where k is the number of parameters in the model.

# 7 Additional tips: `optim`

Here are a few additional things that will be useful when you use `optim`. You will often encounter situations where you need your parameter values to be constrained. For example, if you are trying to find the maximum likelihood estimate of a speciation rate, you will be trying to optimize the function on the interval [0, Inf], because the speciation rate is only defined for numbers $\geq 0$. The critical point: **trying to optimize a function over an interval where a parameter is undefined can lead to bizarre behavior and errors.**

There are at least two ways of dealing with this.

- Explicitly hard-coding bounds for your optimization routine, possibly by using a 'bounded search' algorithm. The built-in algorithm for `optim` is `method = 'L-BFGS-B'`, instead of `"method = 'Nelder-Mead'`
- The above works well for many problems. But optimization often works better when the search space is unbounded. What you really want to do is transform your problem from a $(lower, upper)$ bounded optimization, to one where the optimization is performed on $(-\infty, \infty)$.

If you are optimizing a rate that is only defined on $(0, \infty)$, you can easily transform this into a $(-\infty, \infty)$ problem by log-transforming your rates before optimization. So, you optimize the logarithm of the rate, and just reverse-transform your parameters. To do this, I almost always write a wrapper function that performs all of the necessary transformations. So, in the linear-change model for speciation, suppose we want to constrain the initial speciation rate $\lambda_0$ to be greater than or equal to zero. But we don't want any constraints on the slope parameter. We'll write a wrapper function for `optim` that does the necessary transformation:

```
> optimFx <- function(pars, myFunction, verbose = FALSE) {
+     newpars <- c(exp(pars[1]), pars[2])
+     loglik <- myFunction(newpars)
+     return(loglik)
+ }
> cl <- list(maxit = 1000, fnscale = -1)
> pars.init <- runif(2, -5, 5)
> optim(pars.init, optimFx, myFunction = lfx.linear, method = "Nelder",
+     control = cl)
```

It may not be obvious that we are working with transformed parameters, but we are: note that we raised $e$ to the `pars[1]` power, which is "reverse-logging" it. So, `optim` optimize a number on the interval $(-\infty, \infty)$, but we "undo" this by exponentiating `pars[1]` within the `optimFx` function. **No matter what value of $\lambda_0$ `optim` tries, this transformation will always map the value back to the interval $(0, \infty)$ before feeding it to the likelihood function** . Here are some useful transformations:

- Map a variable defined on $(0, \infty)$ to $(-\infty, \infty)$ by using a log-transform. To do this, simply make sure that you do $e^x$ before feeding $x$ into the likelihood function. Also remember that whatever output you get from `optim` , it will be on the log-scale, and you'll have to manually transform it as appropriate.

- Map a variable from $(A, B)$ to $(-\infty, \infty)$. To do this, note that any number $X$ between $A$ and $B$ can be mapped to the interval $(0, 1)$ by the following transformation:

$$(X - A)/(B - A) \tag{8}$$

  And any $(0, 1)$ optimization problem can be mapped to a $(-\infty, \infty)$ range using a logit transformation. If $z$ is our rescaled $(0, 1)$ variable, we can map it to $(-\infty, \infty)$ as $v = log(z) - log(1 - z)$. And we can transform it back to $(0, 1)$ using the inverse-logit function, or $z = 1/(e^{-v} + 1)$

  In R, these functions are:

```
> logit <- function(p) return(log(p) - log(1 - p))
> invlogit <- function(x) return(1/(exp(-x) + 1))
```

  To use this transformation within `optimFx`, you would first `invlogit` whatever variable `optim` is working with. This maps a $(-\infty, \infty)$ variable back to $(0, 1)$. Then you map this back onto $(A, B)$ using $X = z(B - A) + A$. If $A = 0$ and $B = 1$, you don't need to do anything other than the inverse-logit transformation.

# 8 Problems with bounded searches

Constrained optimization can lead to all sorts of cryptic problems if you aren't careful. For example, if you impose bounds on a particular parameter but the parameter is defined *outside* of the specified range, you might find that your ML estimate of the parameter will be stuck on a boundary. I recommend never using generic bounds on parameters unless you have a good reason to do so.

Here's an example using trait data from fitContinuous in geiger . I'm going to simulate data under a Brownian motion process, then use fitContinuous to estimate the parameters of the BM process.

```
> library(geiger)
> scale <- 100
> vcvmat <- vcv.phylo(v) * scale
> simdata <- mvrnorm(1, mu = rep(0, nrow(vcvmat)), Sigma = vcvmat)
> xx <- fitContinuous(phy = v, data = simdata, model = "BM")

Fitting  BM model:

> xx

$Trait1
$Trait1$lnl
[1] -129.792

$Trait1$beta
[1] 20

$Trait1$aic
[1] 263.5841

$Trait1$aicc
[1] 264.1555

$Trait1$k
[1] 2
```

This model fitting exercise leaves us with the impression that a Brownian motion process with a rate parameter of $\beta = 20$ best explains the observed data. But is this really OK? Actually, it isn't. Here we have a case where the default bounds for the $\beta$ parameter in geiger are not adequate for our problem. geiger defaults to (0, 20) bounds on $\beta$, so we can increase them as follows:

```
> xx <- fitContinuous(phy = v, data = simdata, model = "BM", bounds = list(beta = c(0,
+     Inf)))

Fitting  BM model:

> xx

$Trait1
$Trait1$lnl
[1] -105.885

$Trait1$beta
[1] 87.84898
```

```
$Trait1$aic
[1] 215.77

$Trait1$aicc
[1] 216.3414

$Trait1$k
[1] 2
```

Notice what happened here: the default bounds were too narrow, so fitContinuous found a maximum on the boundary of $\beta = 20$. But in fact, the true ML estimate of $\beta$ is much larger. You can see that the likelihood of the data is much higher when we increase the bounds of $\beta$ to $(0, \infty)$.

This is not a problem with geiger or `fitContinuous` : it is an issue that comes up anytime you are dealing with optimization (e.g, most of the time), and it requires careful attention to details.

**Summary:** Optimization underlies almost everything we do in phylogenetic comparative methods, and this is a very superficial overview of the topic. If you are fitting a model to data, you are almost certainly doing some form of optimization. Even Bayesian MCMC methods (and ABC) can be viewed as a form of optimization.

# 9 Incorporating extinction into the model

Earlier, we explored the function `buildLikelihoodFunction.purebirth` which takes a 2-parameter speciation model as an argument and builds a likelihood function. Now we'll use a function that builds likelihood functions that include both speciation and extinction. The likelihood function will assume that there are 2 parameters that control the speciation rate, and 2 parameters that control the extinction rate (it will accept a vector of 4 parameters as input).

Let's try an implementation of this where $\lambda$ and $\mu$ vary linearly through time. Here we make speciation and extinction functions:

```
> lam <- function(time, pars) {
+     rate <- pars[1] + time * pars[2]
+     rate[rate < 0] <- 0
+     return(rate)
+ }
> mu <- function(time, pars) {
+     rate <- pars[1] + time * pars[2]
+     rate[rate < 0] <- 0
+     return(rate)
+ }
```

Now we'll put these together to build the likelihood function:

```
> bdLik <- buildLikelihoodFunction.extinction(bt, lam, mu)
```

`bdLik` is now a likelihood function that accepts a four-parameter argument, where `pars[1:2]` are speciation parameters, and `pars[3:4]` are extinction parameters.

```
> pars <- c(1, 0, 0.5, 0)
> bdLik(pars)

[1] -64.48452

> pars <- c(0.7, 0, 0.2, 0)
> bdLik(pars)

[1] -59.43909
```

And for comparison, we can set $\lambda$ and $\mu$ equal to their values under a pure-birth model, just to see if we get the same log-likelihood:

```
> pars <- c(0.295, 0, 0, 0)
> bdLik(pars)

[1] -51.35471
```

Now you can use the tools you've learned to (i) find the ML estimates of speciation and extinction parameters, and (ii) plot the estimated rates through time.