Extending diversitree

Rich FitzJohn

25 March 2012, version 0.9-2

1 Introduction

The diversitree package is set up so that it is relatively straightforward to implement new models, and take advantage of a number of package features with little effort, such as the maximum likelihood inference, MCMC, and support for constraining functions, etc.

The package contains a number of support features for writing a model where the calculations can be grouped into three types:

1. Propagating variables from the tip to base of a branch
2. Combining variables at nodes
3. Combining variables at the root

A large number of models can be implemented this way. It may not always be the most efficient, but it can require very little writing.

The code below requires that diversitree is loaded.

```
library(diversitree)
```

There are a number of functions that are used that are not exported from the package namespace by default. Wherever you see a "`diversitree:::`" operator, this extracts these hidden functions.

diversitree makes heavy use of "functional programming" techniques. In particular, functions are often used as arguments to functions, and functions are often returned from functions. This can be confusing at first, but avoids large parameter lists being passed around, and means that higher level functions (such as those that we will use below) do not need to know anything about how the calculations are being carried out.

> **A word of warning**: Everything here is subject to change. If you make a model that works with this version, it may not work with the next version, as these functions may change.

2 Re-implementing the Mk2 model

The Mk2 model already exists in diversitree, but we'll re-implement it here[1].

There are a couple of key calculation features that we need first: a function to compute the probabilities along a branch, and a function to combine probabilities at nodes. After that, we will sort out the book-keeping, assemble the likelihood function and test it out.

[1] The version in diversitree is substantially faster than this version as it is implemented quite differently, but this is a simple model to demonstrate how tip-to-root calculation models can be implemented.

2.1 Branch calculations

Let $D_i(t)$ be the probability that a branch at some time t before the present (at $t = 0$) will yield all the observed data descended from that branch, and let q_{ij} the the rate of transition from state i to j. Along a single branch, the Mk2 model can be expressed as a pair of coupled ordinary differential equations (ODEs):

$$\begin{aligned} \frac{\mathrm{d}D_0(t)}{\mathrm{d}t} &= -q_{01}D_0(t) + q_{01}D_1(t) \\ \frac{\mathrm{d}D_1(t)}{\mathrm{d}t} &= -q_{10}D_1(t) + q_{10}D_0(t) \end{aligned} \tag{1}$$

For clarity, $D_0(t)$ and $D_1(t)$ are our variables, and q_{01} and q_{10} are our parameters.

These can be solved numerically, given initial conditions $D_0(0), D_1(0)$, using deSolve. The deSolve integrators require functions that take three arguments:

- `t`: Time at which the derivatives will be evaluated (ignored here, and in most current diversitree models).
- `y`: Vector of variables.
- `pars`: Vector of parameters.

It must return a list, the first (and possibly only) element of which is a vector of derivatives of each of the variables. Please see the documentation for `lsoda` for more information. We can do this entirely within R code this way (illustrated for clarity, not speed):

```
derivs.mk2new <- function(t, y, pars) {
  D0 <- y[1]
  D1 <- y[2]

  q01 <- pars[1]
  q10 <- pars[2]

  dDdt <- c(-q01 * D0 + q01 * D1,
            -q10 * D1 + q10 * D0)

  list(dDdt)
}
```

We can test this out with the `lsoda` function[2]. We need to specify the initial conditions of the variables, the times at which to return the values of the variables, our derivative function, and the parameters to pass through to this function. All other arguments are optional. Here, the initial condition "`y`" of `c(0,1)` corresponds to a tip in state 1, and the time vector "`tt`" gives equal-spaced times between `0` and `5`. `lsoda` returns a matrix, where each row corresponds to a time in the time vector, and the first column is time: this is dropped with the `[,-1]` below. The remaining columns represent the different variables.

[2]The deSolve package provides an interface to a large number of integrators. The `lsoda` integrator seems to perform well.

```
y <- c(0, 1)
tt <- seq(0, 5, length=101)
pars <- c(.5, 1)
out <- lsoda(y, tt, derivs.mk2new, pars)[,-1]
```

The variables are shown in figure 1. This plot shows the probability of observing a tip in state 1, given that we are in state `0` ($D_0(t)$: black solid) or state 1 ($D_1(t)$: red dashed) over time. Close to the present, the data is much more likely if we are in the same state as the tip, but as time increases, we converge on the stationary distribution for the parameters, indicated by the dotted black line (at $\Pr(\text{state} = 1) = q_{01}/(q_{01} + q_{10})$).

The integration appears to work. However, we still have to do a little work to convert this into what diversitree needs for calculations along branches. It is not the case here directly, but for many problems, the D values shrink over time, and underflow can be a problem (the numbers get too small to accurately work with because of the finite precision available in floating point numbers)[3] This can be avoided by, at the end of the integration, we can sum the data columns to get the factor z. We can then remember $\ln(z)$, and divide the elements of the returned vector by z, so that at least one element stays around order 1 (the largest element in the mk2 model output will never be smaller than 0.5).

For its branches function, diversitree requires a function that takes the arguments

- `y`: Initial conditions, as above
- `len`: Sorted vector of lengths of time for which the integration should performed
- `pars`: Vector of parameters, as above
- `t0`: Initial time.
- `idx`: The branch index (described later)

These arguments do not all need to be used (often `idx` is ignored), but they must be present. Integration starts at `t0`, and runs for length of time `len[1]` to time `t0 + len[1]`, then up to time `t0 + len[2]`, and so on. The required return value differs from what deSolve returns. diversitree expects a list with two elements. The first element is a vector of "compensation factors" from above, and the second is a matrix with `length(y)` rows and `length(len)` columns.

The `make.branches()` function will convert the derivative function above to the required format automatically. This function is not exported by default, so we'll extract it:

```
make.branches.dtlik <- diversitree:::make.branches.dtlik
check.control.ode <- diversitree:::check.control.ode
```

We must give it list with elements:

```
info <- list(name="mk2new", np=2, ny=2, idx.d=1:2, derivs=derivs.mk2new)
```

where `ny` is the number of variables (2 here), `idx.d` is the indices of the "data" variables, which may become small (here, 1 and 2 – both of them), and `derivs` is the derivatives function as above, in the format that `lsoda` expects. We are going to add more things to that list over time.

Generating our function:

[3] This is a problem with Mk2, but only because of combining probabilities at nodes, which we will get to later.

```
matplot(tt, out, type="l", las=1)
legend("topright", c("State 0", "State 1"), col=1:2, lty=1:2)
abline(h=pars[1]/sum(pars), lty=3)
```

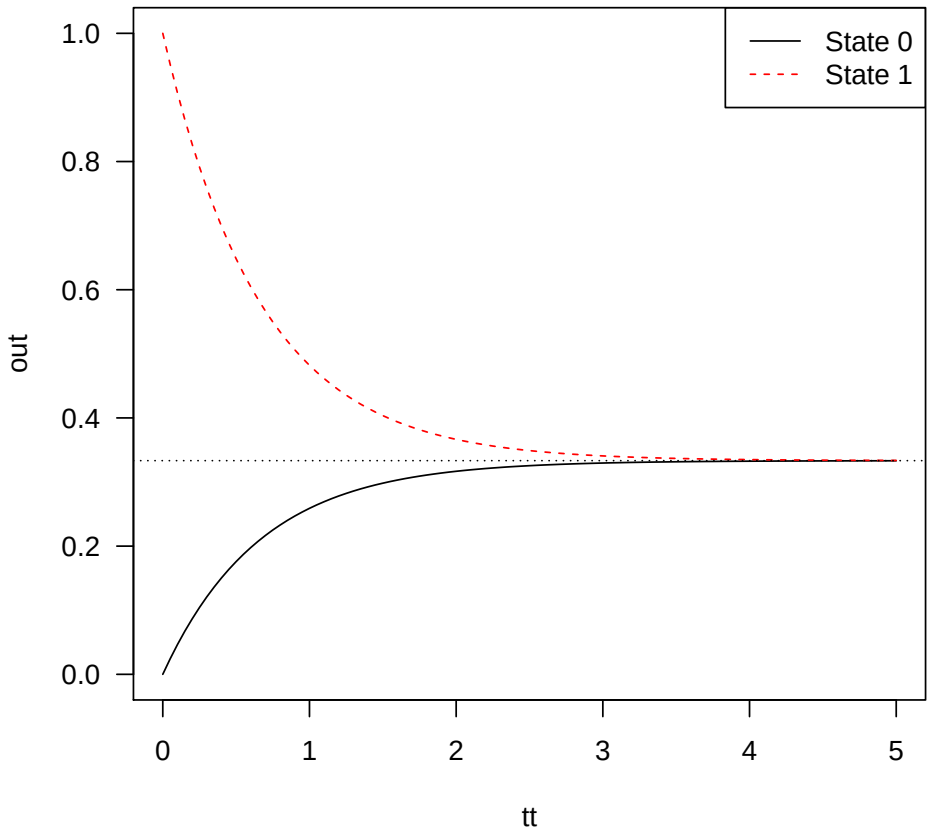

Figure 1: Evolution of the D variables under Mk2 model, starting from a tip in state 1.

```
branches.mk2new <- make.branches.dtlik(info, check.control.ode())
```

The returned function has more arguments:

```
args(branches.mk2new)
```
```
function (y, len, pars, t0, idx)
NULL
```

And returns a list:

```
out.new <- branches.mk2new(y, tt, pars, 0)
str(out.new)
```
```
List of 2
 $ : num [1:101] 0 -0.0244 -0.0475 -0.0695 -0.0904 ...
 $ : num [1:2, 1:101] 0 1 0.0247 0.9753 0.0487 ...
```

2.2 Calculations at nodes

The probability that a node is in state i at a node is just the probability that it will yield its two descendant branches, which is their product. The argument list for this function must be `init`, which is a matrix with with two columns, each column of which are the variable values at the base of the daughter branches from the current node, `pars` (vector of parameters), `t` (time, away from the present), and `is.root` (whether the node is the root node). It must return a vector of variables as output:

```
initial.conditions.mk2new <- function(init, pars, t, idx)
  init[,1] * init[,2]
```

2.3 Initial conditions, and "caching" information about the tree

To plan our traversal along the tree, the `make.cache` function (not exported by default) works out the order in which nodes will be processed, and sorts components of the tree appropriately. First, recall a few features of tree `ape`'s tree format.

- A tree, `phy` has `length(phy$tip.label)` species (say, `n.tip == length(phy$tip.label)`).
- Within the "edge matrix" (`phy$edge`), indices `1:n.tip` refer to taxa, index `n.tip+1` refers to the root, and `(n.tip+2):(2*n.tip - 1)` refer to internal nodes.

Below I will use "entity" to refer to a node or terminal. These correspond to the `n.ent == 2*n.tip - 1` unique indices within the edge matrix.

We also need initial conditions, corresponding to the tip states. These can take two different forms, though only the one relevant for a modest number of discrete characters is covered here. The `dt.tips.grouped` function takes arguments

- `y`: a list of possible initial conditions. `y[[i]]` will contain the initial condition for the `i`'th possible state (`i` here is 1 for state `0`, 2 for state 1 and 3 for state `NA` [unknown state]).
- `y.i`: a vector indicating which of these three possibilities each tip falls into.

- `tips`: a vector of tip indices, generated as part of the cache object.
- `t`: a vector of times.

This will probably change very shortly so that `tips` and `t` are replaced with `cache`. The `check.states` function makes sure that the states are ordered appropriately for the given tree.

There are some checking functions to carry out repetitive sanity checking. `check.tree` checks that the tree contains no polytomies (not allowed by diversitree's traversal algorithm) and ultrametricness (still required by most algorithms). `check.states` checks that the states and taxa can be aligned, and that the states contain only sensible values.

```
make.cache <- diversitree:::make.cache
check.tree <- diversitree:::check.tree
check.states <- diversitree:::check.states
dt.tips.grouped <- diversitree:::dt.tips.grouped
```

```
make.cache.mk2new <- function(tree, states, strict) {
  tree <- check.tree(tree)
  states <- check.states(tree, states, strict=strict, strict.vals=0:1)
  cache <- make.cache(tree)
  cache$info <- list(name="mk2new", np=2, ny=2, idx.d=1:2,
                     derivs=derivs.mk2new, argnames=c("q01", "q10"))

  y <- list(c(1, 0), c(0, 1), c(1, 1))
  y.i <- states + 1
  cache$y <- dt.tips.grouped(y, y.i, cache)

  cache
}
```

2.4 Constructing the likelihood function

Finally, put it all together. The key function here is `all.branches.matrix` (not exported by default). This takes the three components that we have built: `cache`, `initial.conditions` and `branches` and computes values tip to base for each branch in the tree. This returns a list with three elements:

- `init`: variable values at the tip of each branch
- `base`: variable values at the base of each branch
- `lq`: The compensation factor for each branch.

Recall that the compensation factor is the log of the sum of the variables at the base of each branch. These need multiplying back through the likelihood, which we can do by adding the log of the sums back. The value of the variables at the root can is in `ans$init[[cache$root]]`. If we have a flat prior on the root (i.e., assign equal probability to each state), then combining with the compensation factor stored in `ans$lq`, we have `log(sum(ans$init[[cache$root]])/2) + sum(ans$lq)` as the log likelihood.

Putting it all together gives

```
make.all.branches.dtlik <- diversitree:::make.all.branches.dtlik
check.pars.nonnegative <- diversitree:::check.pars.nonnegative
```

```
make.mk2new <- function(tree, states, strict=TRUE) {
  cache <- make.cache.mk2new(tree, states, strict)
  all.branches <- make.all.branches.dtlik(cache, list(),
                                          initial.conditions.mk2new)
  ll <- function(pars) {
    check.pars.nonnegative(pars, 2)
    ans <- all.branches(pars)
    d.root <- ans$vals
    log(sum(d.root * c(.5, .5))) + sum(ans$lq)
  }
  class(ll) <- c("mk2new", "dtlik", "function")
  ll
}
```

2.5 Testing the function out

That's it – we should be good to go. Let's test this on a simulated tree

```
pars <- c(.1, .1, .03, .03, .1, .2)
set.seed(3)
phy <- trees(pars, "bisse", max.taxa=25, max.t=Inf, x0=0)[[1]]
states <- phy$tip.state
```

Here is the likelihood function from diversitree:

```
lik.dt <- set.defaults(make.mk2(phy, states), root=ROOT.FLAT)
lik.dt(c(.1, .2))
```

```
[1] -12.30444
```

and here is our new version:

```
lik.new <- make.mk2new(phy, states)
lik.new(c(.1, .2))
```

```
[1] -12.30444
```

The calculations are very similar, accurate to something on the order of 10^{-7} (may vary by machine).

```
lik.new(c(.1, .2)) - lik.dt(c(.1, .2))
```

```
[1] -3.287332e-08
```

The entire implementation is shown in figure 2.

All of the normal ML and MCMC routines will work on this new function. For example, we can find the ML point (albeit slowly):

```
make.mk2new <- function(tree, states, strict=TRUE) {
  cache <- make.cache.mk2new(tree, states, strict)
  all.branches <- make.all.branches.dtlik(cache, list(),
                                          initial.conditions.mk2new)
  ll <- function(pars) {
    check.pars.nonnegative(pars, 2)
    ans <- all.branches(pars)
    d.root <- ans$vals
    log(sum(d.root * c(.5, .5))) + sum(ans$lq)
  }
  class(ll) <- c("mk2new", "dtlik", "function")
  ll
}
make.cache.mk2new <- function(tree, states, strict) {
  tree <- check.tree(tree)
  states <- check.states(tree, states, strict=strict, strict.vals=0:1)
  cache <- make.cache(tree)
  cache$info <- list(name="mk2new", np=2, ny=2, idx.d=1:2,
                     derivs=derivs.mk2new, argnames=c("q01", "q10"))

  y <- list(c(1, 0), c(0, 1), c(1, 1))
  y.i <- states + 1
  cache$y <- dt.tips.grouped(y, y.i, cache)

  cache
}
initial.conditions.mk2new <- function(init, pars, t, idx)
  init[,1] * init[,2]
```

Figure 2: The entire implementation of the Mk2 model

```
fit.new <- find.mle(lik.new, c(.1, .2))
coef(fit.new)
logLik(fit.new)
```

and this agrees well with the version in diversitree:

```
fit.old <- find.mle(lik.dt, c(.1, .2))
all.equal(coef(fit.old), coef(fit.new))
```

```
[1] "Mean relative difference: 3.213296e-05"
```

```
all.equal(logLik(fit.old), logLik(fit.new))
```

```
[1] TRUE
```

3 Speeding things up

While the calculations above are similar to the diversitree version, they are fairly slow. For 10 evaluations:

```
(t.new <- system.time(replicate(10, lik.new(c(.1, .2))))[[1]])
[1] 0.314
(t.old <- system.time(replicate(10, lik.dt(c(.1, .2))))[[1]])
[1] 0.002
t.new / t.old
[1] 157
```

The total slowdown may depend on the exact hardware, but I get about 170× slower calculations with our new version, compared with the version in diversitree.

4 Reimplementing BiSSE

As an example of a state-dependent diversification model, with more moving parts than the Mk2 model, here, we'll re-implement the BiSSE model. This section assumes knowledge of the functions from above, and of the BiSSE model.

4.1 Branch calculations

As above, let $D_{Ni}(t)$ be the probability that a branch at some time t before the present (at $t = 0$) will yield all the observed data descended from that branch, and now let $E_i(t)$ be the probability that a lineage in state i at time t will go completely extinct by the present, leaving no descendants. BiSSE has six parameters: λ_i is the rate of speciation of a lineage in state i, μ_i is the rate of extinction of a lineage in state i, and as above q_{ij} the the rate of transition from state i to j.

$$\begin{aligned}\frac{\mathrm{d}D_{Ni}}{\mathrm{d}t} &= -(\lambda_i + \mu_i + q_{ij})D_{Ni}(t) + q_{ij}D_{Nj}(t) + 2\lambda_i E_i(t)D_{Ni}(t) \\ \frac{\mathrm{d}E_i}{\mathrm{d}t} &= \mu_i - (\mu_i + q_{ij} + \lambda_i)E_i(t) + q_{ij}E_j(t) + \lambda_i E_i(t)^2\end{aligned} \tag{2}$$

Let's order the variables $\{E_0, E_1, D_{N0}, D_{N1}\}$, and the parameters $\{\lambda_0, \lambda_1, \mu_0, \mu_1, q_{01}, q_{10}\}$. As for Mk2, we build a derivatives function following deSolve's conventions:

```
derivs.bissenew <- function(t, y, pars) {
  E0 <- y[1]
  E1 <- y[2]
  D0 <- y[3]
  D1 <- y[4]

  lambda0 <- pars[1]
  lambda1 <- pars[2]
  mu0 <- pars[3]
  mu1 <- pars[4]
  q01 <- pars[5]
  q10 <- pars[6]

  list(c(-(mu0 + q01 + lambda0) * E0 + lambda0 * E0 * E0 + mu0 + q01 * E1,
         -(mu1 + q10 + lambda1) * E1 + lambda1 * E1 * E1 + mu1 + q10 * E0,
         -(mu0 + q01 + lambda0) * D0 + 2 * lambda0 * E0 * D0 + q01 * D1,
         -(mu1 + q10 + lambda1) * D1 + 2 * lambda1 * E1 * D1 + q10 * D0))
}
```

Test this out with a branch that begins in state 0. This means that E_i is zero for both states, $D_{N0} = 1$ (as we have a tip in state with probability 1) and $D_{N1} = 0$ (as there is no chance to move into state 1 in zero time), so the initial conditions are

```
y <- c(0, 0, 1, 0)
```

Picking parameters that reflect a 2-fold increase in speciation rate for state 1:

```
pars <- c(.1, .2, .03, .03, .01, .01)
```

Integrating over 30 time units, and recording the output regularly:

```
tt <- seq(0, 30, length=101)
out <- lsoda(y, tt, derivs.bissenew, pars)[,-1]
```

The output is plotted in figure 3.

Note that the extinction curves rise to an asymptote over time (this will be similar to μ_i/λ_i, but will not exactly equal this where $q_{ij} > 0$). The $D_{Ni}(t)$ curve for the observed state decreases over time. The other D curve increases slightly, then decreases.

At the node, the E variables are unchanged, and should be identical so we can take either branch. The D variables are multiplied together and with λ:

```
initial.conditions.bissenew <- function(init, pars, t, idx) {
  E       <- init[c(1,2),1] # take the first branch arbitrarily
  D.left  <- init[c(3,4),1]
  D.right <- init[c(3,4),2]
  lambda  <- pars[c(1,2)]
  c(E, D.left * D.right * lambda)
}
```

```
matplot(tt, out, type="l", lty=c(2, 2, 1, 1), col=c("black", "red"))
legend("topright", c("State 0", "State 1"), col=1:2, lty=1:2)
```

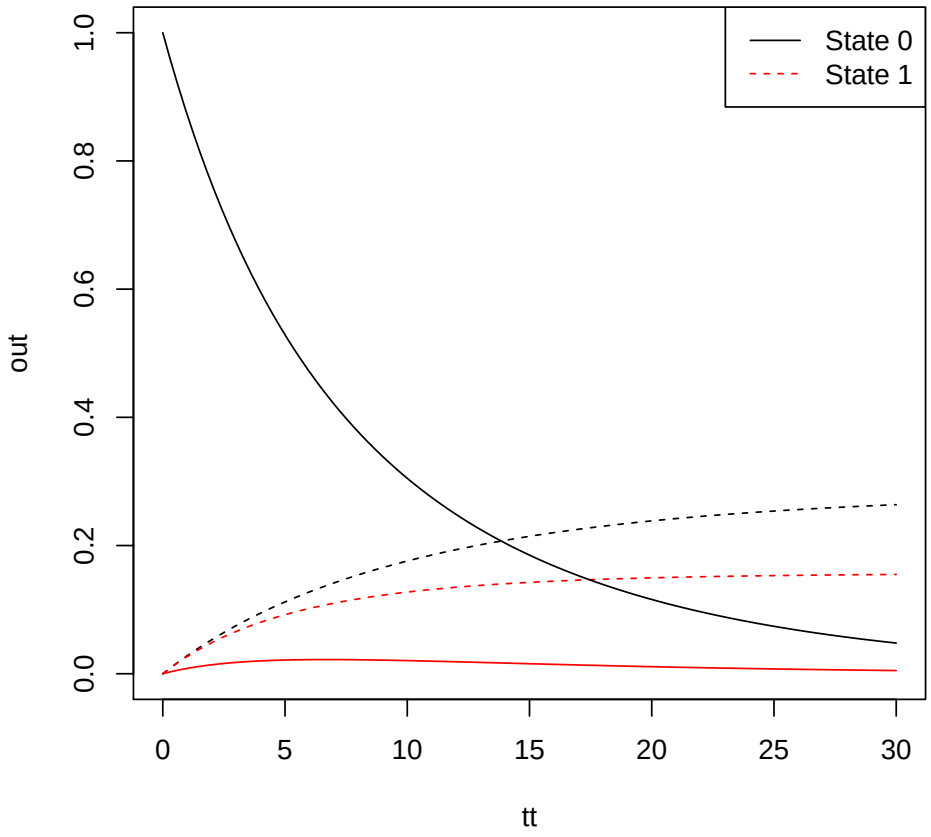

Figure 3: Evolution of the variables under the BiSSE model, starting from a tip in state 0. Black lines indicate state 0, red lines indicate state 1. Dashed lines are $E_i(t)$, and solid lines are $D_{Ni}(t)$.

As above, we'll create an "info" list. Note that ny is now 4, corresponding to our four variables. I've also declared that the function will take 6 parameters (np=6), that variables 3 and 4 are the "data" variables, and will require underflow protection. I also named the model "bissenew" and declared the default parameter names.

```
make.cache.bissenew <- function(tree, states, strict) {
  tree <- check.tree(tree)
  states <- check.states(tree, states, strict=strict, strict.vals=0:1)
  cache <- make.cache(tree)
  cache$states <- states
  cache$info <-
    list(name="bissenew", np=6, ny=4, idx.d=3:4,
         derivs=derivs.bissenew,
         argnames=c("lambda0", "lambda1", "mu0", "mu1", "q01", "q10"))
  cache$y <- initial.tip.state.bissenew(cache)
  cache
}
```

```
initial.tip.state.bissenew <- function(cache) {
  y <- list(c(0, 0, 1, 0), # in state 0: E* = 0, D0 = 1, D1 = 0
            c(0, 0, 0, 1), # in state 1: E* = 0, D0 = 0, D1 = 1
            c(0, 0, 1, 1)) # in state ?: E* = 0, D0 = 1, D1 = 1
  y.i <- cache$states + 1
  dt.tips.grouped(y, y.i, cache)
}
```

```
make.bissenew <- function(tree, states, strict=TRUE, control=list()) {
  cache <- make.cache.bissenew(tree, states, strict)
  all.branches <- make.all.branches.dtlik(cache, control,
                                          initial.conditions.bissenew)
  ll <- function(pars) {
    check.pars.nonnegative(pars, 6)
    ans <- all.branches(pars)
    d.root <- ans$vals[cache$info$idx.d]
    log(sum(d.root * c(.5, .5))) + sum(ans$lq)
  }
  class(ll) <- c("bissenew", "dtlik", "function")
  ll
}
```

Test this out by making a likelihood function and evaluating it at the true parameters:

```
lik.new <- make.bissenew(phy, states)
lik.new(pars)
```

```
[1] -89.57924
```

Compared with diversitree's BiSSE calculation (note that we set some defaults here so that root treatment will be same):

```
lik.old <- make.bisse(phy, states)
lik.old <- set.defaults(lik.old, condition.surv=FALSE, root=ROOT.FLAT)
lik.old(pars)
[1] -89.57924
```

These are really close:

```
lik.old(pars) - lik.new(pars)
[1] 0
```

However, they again differ markedly in time (by a factor of about 15).

```
(t.new <- system.time(replicate(10, lik.new(pars)))[[1]])
[1] 0.513
(t.old <- system.time(replicate(10, lik.old(pars)))[[1]])
[1] 0.031
t.new / t.old
[1] 16.54839
```

4.2 Integrating the ODEs in C

One way of speeding things up is to push the derivative calculations into compiled code via C. Once these are written out in R, this is actually very easy to do. Please see the deSolve manual for details – this example follows very closely. The file `diversitree-ext-bisse.c` contains the derivative calculations from above implemented in C – these look almost identical. See figure 4 for a listing.

Compile the file from the shell (not R) command prompt by typing

```
R CMD SHLIB diversitree-ext-bisse.c
```

and load the library into R with

```
dyn.load("diversitree-ext-bisse.so")
```

We can then repeat the test integration from above:

```
out.C <- lsoda(y, tt, "derivs_bissenew", pars,
               initfunc="initmod_bissenew",
               dll="diversitree-ext-bisse")[,-1]
```

and compare this with the R version from above

```
all.equal(out.C, out)
[1] TRUE
```

```
#include <R.h>

static double parms_bissenew[6];

void initmod_bissenew(void (* odeparms)(int *, double *)) {
  int N = 6;
  odeparms(&N, parms_bissenew);
}

void derivs_bissenew(int *neq, double *t, double *y, double *ydot,
                     double *yout, int *ip) {
  double E0 = y[0], E1 = y[1];
  double D0 = y[2], D1 = y[3];

  double la0 = parms_bissenew[0], la1 = parms_bissenew[1],
    mu0 = parms_bissenew[2], mu1 = parms_bissenew[3],
    q01 = parms_bissenew[4], q10 = parms_bissenew[5];

  ydot[0] = -(mu0 + q01 + la0) * E0 + la0 * E0 * E0 + mu0 + q01 * E1;
  ydot[1] = -(mu1 + q10 + la1) * E1 + la1 * E1 * E1 + mu1 + q10 * E0;
  ydot[2] = -(mu0 + q01 + la0) * D0 + 2 * la0 * E0 * D0 + q01 * D1;
  ydot[3] = -(mu1 + q10 + la1) * D1 + 2 * la1 * E1 * D1 + q10 * D0;
}
```

Figure 4: Contents of `diversitree-ext-bisse.c`

(these may actually be identical on some machines).

Now, all we have to do is modify the `make.cache.bissenew` function, changing the specification of the `info` list. This time, we do not add `derivs` element (as this will always cause the R calculations to be used), and indicate the name of the shared library in which the derivative functions (in C) are contained (note that we do not specify the filename extension here).

```
make.cache.bissenew <- function(tree, states, strict) {
  tree <- check.tree(tree)
  states <- check.states(tree, states, strict=strict, strict.vals=0:1)
  cache <- make.cache(tree)
  cache$states <- states
  cache$info <-
    list(name="bissenew", np=6, ny=4, idx.d=3:4,
         dll="diversitree-ext-bisse",
         argnames=c("lambda0", "lambda1", "mu0", "mu1", "q01", "q10"))
  cache$y <- initial.tip.state.bissenew(cache)
  cache
}
```

The `make.bissenew` function does not need modification.

How does this work? Recall that we set `name="bissenew"`. When using the `deSolve` backend (which is the default), when `info$name` is set to `xxx`, R will look for loaded symbols `initmod_xxx` and `derivs_xxx`. Using this it builds a "branches" function (`make.branches.dtlik` is called by `make.all.branches`

to do this). The `np` and `ny` elements provide important information to this process about how many parameters and variables to expect, and `idx.d` indicates which variables contain conditional likelihoods.

Using this:

```
lik.C <- make.bissenew(phy, states)
lik.C(pars)
[1] -89.57924
lik.C(pars) - lik.new(pars)
[1] 0
```

The new likelihood function is now about as fast as the diversitree version:

```
(t.C <- system.time(replicate(10, lik.C(pars)))[[1]])
[1] 0.03
t.new / t.C
[1] 17.1
t.C / t.old
[1] 0.9677419
```

With a little more work, we can make this faster still. On non-windows platforms, diversitree can use the "cvodes" solver, and carry out all calculations in C. This eliminates quite a bit of overhead from the calculations. It does require that the CVODES header files are on the compiler's search path. The unexported function `cvodes.headers` can determine what the appropriate flag will be (be aware it will change with every new R version). When placed in a file called "Makevars", this will be used, and the following line writes this out:

```
diversitree:::cvodes.headers(to.Makevars=TRUE)
```

The file `diversitree-ext-cvodes.c` contains support for both types of ODE solver, and is shown in figure 5.

As above, compile with

```
R CMD SHLIB diversitree-ext-cvodes.c
```

and load the library into R with

```
dyn.load("diversitree-ext-cvodes.so")
```

We need another version of `make.cache.bissenew`, but only to update the filename of the shared library.

```
#include <R.h>
/* For CVODES */
#include <nvector/nvector_serial.h>
#include <user_data.h>

/* This is the core function that actually evaluates the deriative, as
   in the previous version.  However, to save on duplication, I've
   separated this out a little. */
void do_derivs_bissenew(double *pars, double *y, double *ydot) {
  double E0 = y[0], E1 = y[1], D0 = y[2], D1 = y[3];
  double la0 = pars[0], la1 = pars[1], mu0 = pars[2], mu1=pars[3],
    q01 = pars[4], q10 = pars[5];

  ydot[0] = -(mu0 + q01 + la0) * E0 + la0 * E0 * E0 + mu0 + q01 * E1;
  ydot[1] = -(mu1 + q10 + la1) * E1 + la1 * E1 * E1 + mu1 + q10 * E0;
  ydot[2] = -(mu0 + q01 + la0) * D0 + 2 * la0 * E0 * D0 + q01 * D1;
  ydot[3] = -(mu1 + q10 + la1) * D1 + 2 * la1 * E1 * D1 + q10 * D0;
}

/* Here is the deSolve interface, as before, but using the
   do_derivs_bissenew function defined above */
static double parms_bissenew[6];
void initmod_bissenew(void (* odeparms)(int *, double *)) {
  int N = 6;
  odeparms(&N, parms_bissenew);
}
void derivs_bissenew(int *neq, double *t, double *y, double *ydot,
                     double *yout, int *ip) {
  do_derivs_bissenew(parms_bissenew, y, ydot);
}

/* CVODES */
int derivs_bissenew_cvode(realtype t, N_Vector y, N_Vector ydot,
                          void *user_data) {
  do_derivs_bissenew(((UserData*) user_data)->p,
                     NV_DATA_S(y),
                     NV_DATA_S(ydot));
  return 0;
}

/* This is also required, to compute initial conditions.  It is almost
   identical to the R version */
void initial_conditions_bissenew(int neq, double *vars_l, double *vars_r,
                                 double *pars, double t,
                                 double *vars_out) {
  vars_out[0] = vars_l[0]; /* E0, first branch only */
  vars_out[1] = vars_l[1]; /* E1, first branch only */
  vars_out[2] = vars_l[2]*vars_r[2]*pars[0]; /* D0_l*D0_r*lambda0 */
  vars_out[3] = vars_l[3]*vars_r[3]*pars[1]; /* D1_l*D1_r*lambda1 */
}
```

Figure 5: Contents of `diversitree-ext-cvodes.c`

```
make.cache.bissenew <- function(tree, states, strict) {
  tree <- check.tree(tree)
  states <- check.states(tree, states, strict=strict, strict.vals=0:1)
  cache <- make.cache(tree)
  cache$states <- states
  cache$info <-
    list(name="bissenew", np=6, ny=4, idx.d=3:4,
         dll="diversitree-ext-cvodes",
         argnames=c("lambda0", "lambda1", "mu0", "mu1", "q01", "q10"))
  cache$y <- initial.tip.state.bissenew(cache)
  cache
}
```

Now, when we make the likelihood function, we need to pass in `list(backend="CVODES")` to tell diversitree that we want the alternative backend.

```
lik.CV <- make.bissenew(phy, states, control=list(backend="CVODES"))
lik.CV(pars)
```
```
[1] -89.57924
```
```
lik.CV(pars) - lik.old(pars)
```
```
[1] 1.572636e-06
```

We can still make the deSolve-based version by passing in `backend="deSolve"`, or by omitting the control argument:

```
lik.Cd <- make.bissenew(phy, states, control=list(backend="deSolve"))
lik.Cd(pars)
```
```
[1] -89.57924
```

The CVODES-based likelihood function is now about 5 times faster than the deSolve based version (the speedup varies a lot based on tree size).

```
(t.CV <- system.time(replicate(10, lik.CV(pars)))[[1]])
```
```
[1] 0.007
```
```
t.CV / t.old
```
```
[1] 0.2258065
```

See figure 6 for the complete listing.

```
make.bissenew <- function(tree, states, strict=TRUE, control=list()) {
  cache <- make.cache.bissenew(tree, states, strict)
  all.branches <- make.all.branches.dtlik(cache, control,
                                          initial.conditions.bissenew)
  ll <- function(pars) {
    check.pars.nonnegative(pars, 6)
    ans <- all.branches(pars)
    d.root <- ans$vals[cache$info$idx.d]
    log(sum(d.root * c(.5, .5))) + sum(ans$lq)
  }
  class(ll) <- c("bissenew", "dtlik", "function")
  ll
}
make.cache.bissenew <- function(tree, states, strict) {
  tree <- check.tree(tree)
  states <- check.states(tree, states, strict=strict, strict.vals=0:1)
  cache <- make.cache(tree)
  cache$states <- states
  cache$info <-
    list(name="bissenew", np=6, ny=4, idx.d=3:4,
         dll="diversitree-ext-cvodes",
         argnames=c("lambda0", "lambda1", "mu0", "mu1", "q01", "q10"))
  cache$y <- initial.tip.state.bissenew(cache)
  cache
}
initial.tip.state.bissenew <- function(cache) {
  y <- list(c(0, 0, 1, 0), # in state 0: E* = 0, D0 = 1, D1 = 0
            c(0, 0, 0, 1), # in state 1: E* = 0, D0 = 0, D1 = 1
            c(0, 0, 1, 1)) # in state ?: E* = 0, D0 = 1, D1 = 1
  y.i <- cache$states + 1
  dt.tips.grouped(y, y.i, cache)
}
initial.conditions.bissenew <- function(init, pars, t, idx) {
  E       <- init[c(1,2),1] # take the first branch arbitrarily
  D.left  <- init[c(3,4),1]
  D.right <- init[c(3,4),2]
  lambda  <- pars[c(1,2)]
  c(E, D.left * D.right * lambda)
}
```

Figure 6: The entire implementation of the BiSSE model