RBrownie User's Guide (Version 0.0.4)

J. Conrad Stack with credit due to others(1)

September 13, 2010

1 Introducing Brownie

`RBrownie` is an R package constructed around the C++ command-line program Brownie (available here) which was designed by Brian O'Meara in order to fascilitate the estimation and comparison of rates and means of character evolution over different parts of a phylogeny (O'Meara, et al., 2006). It it built on three primary analyses: First are censored rate tests under Brownian motion which allow for the comparison of the rates of evolution of morphological traits across two monophyletic sister-clades or between a monophyletic and paraphyletic clade. This analysis allows you to answer questions about differences in the evolutionary rate of two distinct clades (e.g. large subspecies vs. small subspecies). Second are non-censored ratetests, so called because they do not remove (censor) any branches of the phylogeny being tested. This class of analyses allows you to compare the rates (Brownian motion model) or means (Ornstein-Uhlenbeck model) of evolution under different character states which are mapped onto the phylogeny. For example, if you had a tree with binary characters mapped onto it branches indicating possession a certain trait and you also had morphological data for all the taxa present in the tree (the tips), then, using a model-testing approach, this analysis allows to you statistically assess whether or not that binary trait is associated with the significantly different rate of evolution of the morphological characteristic measured. Lastly, Brownie performs discrete ancestral state evolution (maximum likelihood approach), which produces branch-annotated trees which can used in these non-censored rate tests.

`RBrownie`'s aim is to exposes all of Brownies capabilities to R users so that they can be used in conjuction with R's existing phylogenetic toolkit. It provides users with easy ways to run common Brownie analyses and the flexibility to write more complex analyses if desired. Below I give a short introduction to Brownie and `RBrownie`, discusses the main datatypes in `RBrownie` and how to use them, and then a number of examples of the sort of evolutionary analyses that can be completed in `RBrownie`, including visualizing and understanding the results.

1.1 More information

Brownie has a large number of functions in addition to the rate tests described above that we cannot describe in full here, but are worth looking into at least because they will help you better understand `RBrownie`'s capabilities and will allow you to exploit all the useful functions `RBrownie`) has to offer. The manual for Brownie can be found here.

2 Introducing RBrownie

It was mentioned before that `RBrownie` is an R wrapper for the Brownie program described briefly above. Using the `Rcpp` package, `RBrownie` links R with the exact same Brownie code that is used in the Brownie command line program. In addition to providing a streamlined interface to Brownie's core functionality, `RBrownie` adds plotting and visualization functions which make working with the output data much more easy and flexible.

Like PAUP*,Mesquite, and other phylogenetic programs, Brownie uses specially formatted nexuses files to load phylogenetic data into the program. `RBrownie` adds functions and classes to help the user read, write, and manipulate these tailored nexus files, supplementing the more generic NEXUS read/write functions found in the `ape` and `phylobase` packages (see `ape:::read.nexus.data`, `ape:::write.nexus.data`, `phylobase:::writeNexus`).

To get started using the package (once it has been installed), type the following:

```
> require(RBrownie)
```

Note that the other packges on which `RBrownie` depends are listed. Now that we have the package loaded, we can look into how to load and/or manipulate our phylogenetic data in `RBrownie`.

2.1 Reading nexus files

Nexus files can be read into R using a number of packages. `ape` and `phylobase` both include the functionality to read in standard trees and sequence data, while phylobase also reads standard CHARACTERS (or DATA) blocks. For more information on the options presented by these too packages, check out their documentation:

```
> ?read.nexus                 # ape
> ?read.nexus.data          # ape
> ?readNexus                  # phylobase
```

However, Brownie uses nexus blocks beyond those supported by `ape` and `phylobase` and `RBrownie` extends the methods above to accomadate. For example, Brownie uses an ASSUMPTIONS block to define subsets of the taxa which can be used in its various rate tests. It also uses a BROWNIE block to

store a list of brownie commands to be run automatically if the file is "executed" within the Brownie environment (this is similar to how PAUP* executes files). So, RBrownie has its own functions for reading and writing nexus files with these special blocks, readBrownie and writeBrownie.

To illustrate how they work we'll use the parrot dataset which is included with RBrownie (NOTE: you need to have write access to the directory you are in to run this example).

```
> data(parrot)
> writeBrownie(parrot, file = "parrotdata.nex")
> newparrot = readBrownie("parrotdata.nex")
```

The two objects parrot and newparrot should be indentical. It would also be instructive to open the parrotdata.nex file to see how RBrownie writes ASSUMPTIONS blocks, but this isn't necessary. You may have also noticed that parrot and newparrot are both of class list. The list class (a very common class in R) is used when multiple trees are found in a single nexus file. For example, if you opened parrotdata.nex, you would find 10 trees in the TREES block. Each element in the parrot (and newparrot) list contains one of these trees and is of class brownie:

```
> class(parrot)           # list class
> length(parrot)                # == 10
> class(newparrot)           # list class
> length(newparrot)         # == 10
> #
> class(parrot[[1]])  # brownie class representing the first tree in the list
```

brownie is the main class that RBrownie uses to represent phylogenetic data. It and other RBrownie classes are discussed in the next section,

2.2 Classes in RBrownie

At its core RBrownie is built around two major classes, phylo4d_ext and brownie, which both extend phylo4d from phylobase. In fact, phylo4d_ext extends phylo4d and brownie in turn extends phylo4d_ext. In non-computer-ese this mean that phylo4d_ext adds a few new data containers (or "slots" as they are known in the S4 world) to phylo4d and brownie adds a few new data containers to phylo4d_ext. To illustrate this run the R code below. It shows which new slots have been added to phylo4d.

```
> require(RBrownie,quietly=TRUE,warn.conflicts=FALSE)
> phylo4d_slots = names(getSlots("phylo4d"))
> phylo4d_ext_slots = names(getSlots("phylo4d_ext"))
> brownie_slots = names(getSlots("brownie"))
> phylo4d_slots
```

```
[1] "data"        "metadata"    "edge"        "edge.length"
[5] "label"       "edge.label"  "order"       "annote"
```

```
> # set of differences between phylo4d and phyl4d_ext
> setdiff(phylo4d_ext_slots,phylo4d_slots)
```

```
[1] "subnode.id"     "subnode.data"   "subnode.branch"
[4] "subnode.pos"    "weight"
```

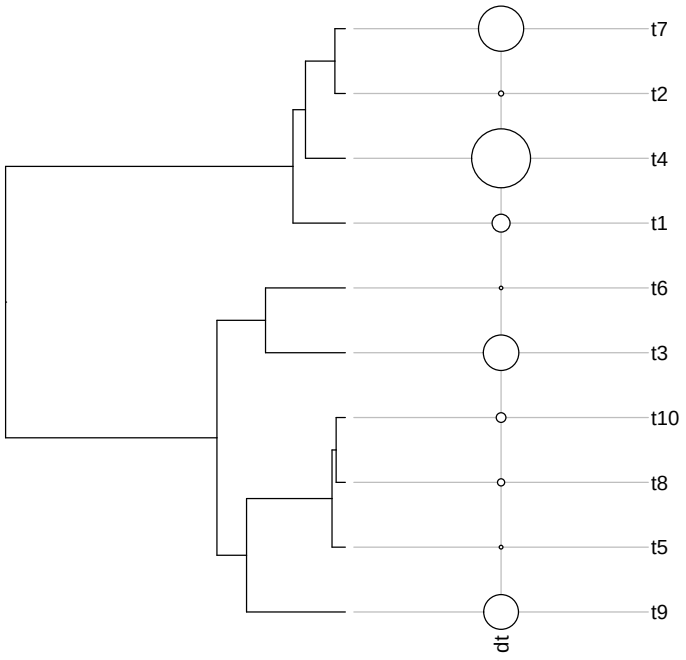
```
> # set of differences between phyl4d_ext and brownie
> setdiff(brownie_slots,phylo4d_ext_slots)
```

```
[1] "commands"  "datatypes"
```

2.2.1 phylo4d_ext class

This class (as implied by the name) is an extension of the `phylo4d` class from phylobase. This base class holds a phylogenetic tree and a `data.frame` containing data for each node of the tree. The code below shows how to construct a random coalescent tree and add junk data to the tips of tree which in practice might represent morphological data for each of the taxa.

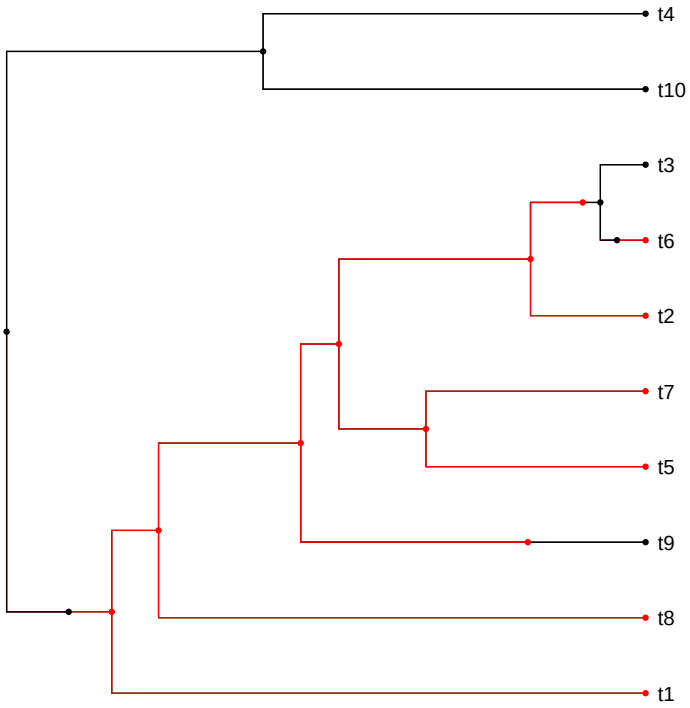
```
> require(phylobase)
> # generate a random coalescent tree with 10 tips
> ape_tree = rcoal(10)
> #
> # convert tree from 'phylo' (ape) to 'phylo4' (phylobase) format
> phy_tree = as(ape_tree,"phylo4")
> #
> # generate junk data for the tips
> sample_tipdata = runif(10)
> #
> # combine the phylogenetic tree and the sample data
> phyd_tree = phylo4d(phy_tree,tip.data=sample_tipdata)
> plot(phyd_tree)
```



`phylo4d_ext` adds to this class "subnodes" and tree weights. Subnodes are internal nodes mapped directly onto an existing branches and are conceptually similar to singleton nodes (but offer a bit more flexibility from a programming standpoint). The `phylo4d_ext` class was created mainly to handle SIMMAP-formatted nexus trees, but instances can be created from scratch as well. For example, the code below constructs a random coalescent tree like we did in the last section, but with binary data representing say the presence or absence of a life history trait and also with subnodes along different branches of the tree each representing either loss(0) or gain(1) of the trait along the branch.

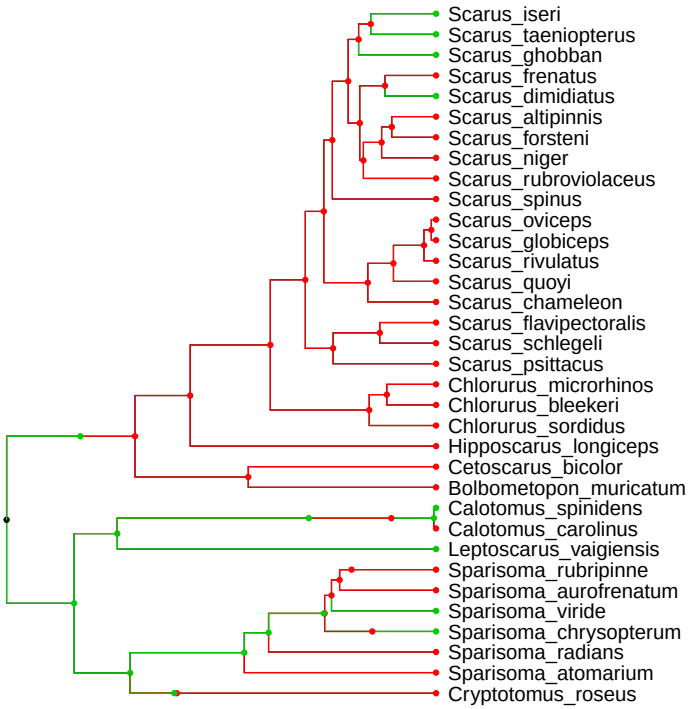
```
> require(RBrownie)
> ape_tree = rcoal(10,br=runif(9,5,15))
> phy_tree = as(ape_tree,"phylo4")
> #
> # Discrete trait data:
> # (Loosely simulate the evolution of a binary trait down a tree,
> #  choosing a random value for the tip and then changing
> #  the trait based on the the change matrix.)
> #
> phy_tree = reorder(phy_tree,order="preorder")
> sample_binarydata = data.frame(rep(0,19)) # junk data
> names(sample_binarydata) <- "hasTrait"
> sample_binarydata[rootNode(phy_tree),1] = sample(c(0,1),1)  # choose root value
```

```
> changemat = matrix(c(0.8,0.2,0.2,0.8),byrow=T,nrow=2) # prob. of changing on branch
> sample_edges = integer(0)  # track edges where change occurs
> for(i in seq(nrow(edges(phy_tree))))
+ {
+         anc = edges(phy_tree)[i,1]
+         desc = edges(phy_tree)[i,2]
+         if(anc == 0)
+                 next
+
+         newvalue = sample(c(0,1),1,prob=changemat[(sample_binarydata[anc,1]+1),])
+         sample_binarydata[desc,1] = newvalue
+         if( newvalue != sample_binarydata[anc,1] )
+                 sample_edges = append(sample_edges,i)
+ }
> #
> #
> # Subnodes
> # (put subnodes on the branches where the trait changed showing
> #  where on the branch the change happened.  Note that the
> #  subnode data.frame must have the same structure as the
> #  parent data.frame)
> #
> tmpdata = as.integer(!sample_binarydata[edges(phy_tree)[sample_edges,2],1])
> sample_subnodedata = data.frame(tmpdata)
> names(sample_subnodedata) <- "hasTrait"
> #
> # Position the subnodes along their respective branches
> # using the fraction of the overall branch length.  In
> # this case, I'll put them near the middle of the branch.
> sample_positions = runif(n=length(tmpdata),min=0.25,max=0.75)
> #
> # Create the phylo4d_ext object, which first creates
> # a phylo4d object and then adds the subnodes to it.
> #
> # NOTE: any "extra" arguments (all.data) are passed
> # on to the phylo4d constructor via the ellipsis argument (...).
> phyext_tree = phyext(phy_tree,
+                         snode.data=sample_subnodedata,
+                         snode.branch=edges(phy_tree)[sample_edges,],
+                         snode.pos=sample_positions,
+                         all.data=sample_binarydata)
> # nodes, subnodes, and branches without the "trait" (state 0) are colored black
> # nodes, subnodes, and branches with the "trait" (state 1) are colored red
> plot(phyext_tree,states=c(0,1),states.col=c("black","red"))
```



In the plot above, the subnodes are clearly seen as dots on an existing branch where the state (colors) change. We roughly simulated the evolution of a binary trait down the tree, added subnodes to show that those changes occurred at a certain point along the branch (instead of only at nodes), and then plotted the tree colored to represent which state it was in. Another example of the use of subnodes can be seen in the parrot data set:

```
> require(RBrownie)
> data(parrot)
> plot(parrot[[1]])
```



Now the binary state data are represented by the default colors in `RBrownie` and a state change is clearly visible on the branch from the taxon Sparisoma_chrysopterum. In this case, state 0 and state 1 represent a life history trait of parrot fish, namely whether they scrape (0) or excavate (1) coral. You can clearly see that, in this tree at least, that life history trait changes throughout the evolutionary history of the parrot fish at both internal nodes and along a branch at a subnode. Brownie uses this state information, as we will see later, to address questions such as "Does coral excavatingscraping limit morphological evolution in parrotfishes" (Bodega Phylogenetics Wiki , 2010)

To view subnode information these, accessor functions are available:

```
> snid(phyext_tree)  # access subnode ids (not currently used)
> sndata(phyext_tree)  # access subnode data.frame (should reflect the parents d.f)
> snposition(phyext_tree)  # access subnode positions
> snbranch(phyext_tree)  # access branches subnodes are on
> showSubNodes(phyext_tree)  # print a textual representation of the subnodes
```

In the example above where we simulated the evolution of a binary trait down a tree, we created an instance of `phylo4d_ext` using its constructor `phyext`. If you would like to add a new subnode manually to this tree you would use `addSubNode`. For example:

```
> #
> ancestor.node = edges(phyext_tree)[1,1]
```

```
> descendant.node = edges(phyext_tree)[1,2]
> node.position = 0.5
> node.data = data.frame(1)
> names(node.data) <- "hasTrait"
> #
> phyext_tree = addSubNode(phyext_tree,
+                                  ancestor.node,
+                                  descendant.node,
+                                  node.position,
+                                  node.data)
```

`weight` is an aspect of the `phylo4d_ext` class that will be supported better in future versions and will not be discussed here.

Finally, subnodes will most be read in from SIMMAP-formatted nexus files and not added manually. The SIMMAP format was created represent state mappings onto branches (subnodes) in nexus tree files and the standard has gone through a number of iterations. Currently, only SIMMAP version 1.0 and 1.1 are supported by `RBrownie` but we are working to add support for the latest version (1.5) which is used by Mesquite. `RBrownie` can be used to read and write trees with subnodes in SIMMAP (vers 1.0, 1.1) format: (NOTE: you need to have write access to the directory you are in to run this example).

```
> data(parrot)
> write.nexus.simmap(parrot, file = "parrottmp.nex")
> newparrot = read.nexus.simmap("parrottmp.nex")
```

2.2.2 brownie class

Many examples in the last section where the phylo4d_ext class was explained used the parrot dataset included in RBrownie. But, the parrot object is a list of `brownie` objects, right? Yes, the `brownie` class extends the `phylo4d_ext` class further adding two new slots to fascillitate doing Brownie analyses. This can be seen using the following code:

```
> data(parrot)
> class(parrot[[1]])
> inherits(parrot[[1]], "phylo4d_ext")
```

The purpose of the `commands` slot is to hold the text strings that will eventually be executed in Brownie itself. That is, once the phylogenetic tree and any node data is loaded into the Brownie core, these commands are run. They describe actions that the brownie core should take: analyses might need to be run on certain trees and not others, choose which models to use when model tests are done, etc. A full list and description of all the commands can be found in the Brownie manual.

In common use however, these commands will be filled in automatically by some higher level functions (e.g. `addCensored`) which were designed to make

it easy to set up and execute common analysis. If you do find yourself needing to manipulate these commands, there are a series of functions to manipulate commands directly (`clearCommands`, `removeCommands`, etc) that are probably irrelevant to most end users and are not discussed here.

The `datatypes` slot is a character vector which contains a description of each column in the `data` slot. `RBrownie` uses this information in order to write different parts of the `data` data.frame to different blocks in a nexus file. For example, discrete and continuous data are treated differently in Brownie and each require their own nexus block (CHARACTERS and CHARACTERS2 are used). Also, taxa sets are stored in the `data` slot as binary data (1 is taxon is in the set, 0 if not) and they are written to an ASSUMPTIONS nexus block. To illustrate this, lets look at the data in parrot data set. (NOTE: The `data` slot comes from phylobase, its accessor functions also come from there. `tdata` is an example)

```
> require(RBrownie)
> data(parrot)
> # How many columns are in the data.frame?
> ncol(tdata(parrot[[1]]))

[1] 5

> # What are their names?
> names(tdata(parrot[[1]]))

[1] "simmap_state"      "pc1"
[3] "pc2"               "TAXSET_all"
[5] "TAXSET_intrajoint"

> # What are their types?
> datatypes(parrot[[1]])

[1] "discrete" "cont"     "cont"     "taxset"   "taxset"
```

We can see here that this brownie object contains five columns in its data.frame and they are of 3 different types. The first `discrete`, indicates discrete data; the second and third `cont` indicate continuous data, and the forth and fifth `taxset` indicate a special binary data column showing which taxa are in a certain set of taxa.

There are handy ways to access these datatype strings when using them:

```
> discData()
> contData()
> taxData()
> genericData() # undefined data (shouldn't be used)
> brownie.datatypes() # show all
```

Adding new data to a `brownie` object is done like so:

```
> #
> require(RBrownie)
> data(parrot)
> #
> # junk morphological data for tips
> junkcont = runif((nNodes(parrot[[1]])+1),-10,10)
> #
> # junk discrete data for tips / nodes
> junkdisc = sample(letters[1:4], length(junkcont) + nNodes(parrot[[1]]),replace=TRUE)
> #
> # (Note: if dataTypes argument is left out, the RBrownie will
> #  attempt to guess the right datatype.  It is always better to specify.)
> #
> parrot_new = addData(parrot, tip.data=junkcont, dataTypes=contData())
> parrot_new = addData(parrot_new, all.data=junkdisc, dataTypes=discData())
> #
> datatypes(parrot_new)

[1] "discrete" "cont"     "cont"     "taxset"   "taxset"
[6] "cont"     "discrete"
```

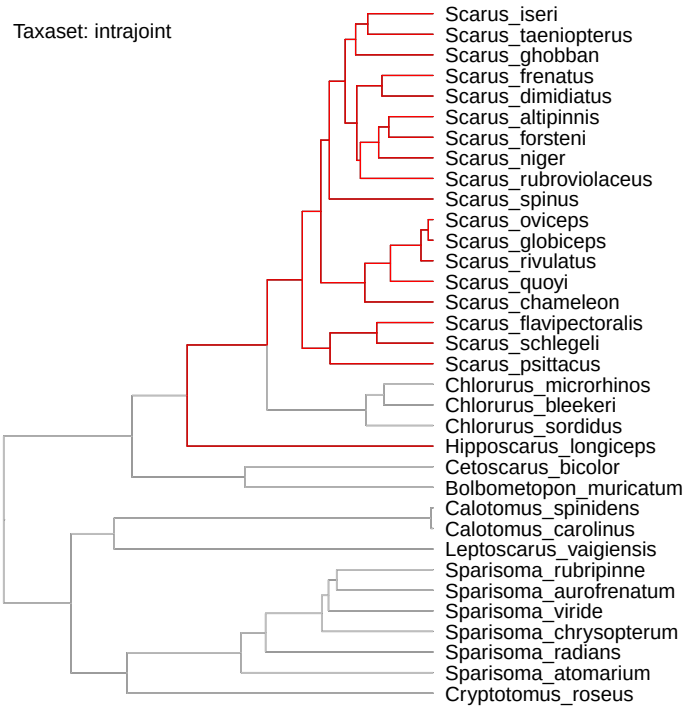
Now there are two new datatypes in the parrot data which we specified. Also note how `addData()` and `datatypes()` can be called on list objects as well as brownie objects - the list variant of these functions actually adds the data to each object in that list. `RBrownie` has a number of such convenient functions.

Adding taxa set data is a bit different. There is an accessor function in `RBrownie` called `taxasets()` which gives the user access to all the data columns representing sets of taxa. It can also be used to create new taxasets:

```
> hasTaxasets(parrot) # see if there are any taxasets
> taxasets(parrot) # return taxasets as data.frame
> #
> #
> # Use taxa names to indicate membership in a taxa set:
> all_sparisoma = grep("Sparisoma",tipLabels(parrot[[1]]),value=TRUE)
> taxasets(parrot,taxnames="sparisomas") <- all_sparisoma
```

Notice how we can both view all the sets of taxa and assign a new taxaset using the same command, `taxasets()`. This is a quirk of S4 methodology - just remember that if you call the function alone it will show you all the sets of taxa and if you "assign" a new value the function a new taxa set will be created (using `<-`). You can visualize a set of taxa as well:

```
> require(RBrownie)
> data(parrot)
> plot.taxaset(parrot[[3]],2) # plot the second taxaset
```



And that is the `brownie` class which can be thought of as a specialization of generic phylogentic trees + data. It would be instructive to take some of the examples above, where data is added to data contained within `RBrownie`, write it out to a file using `writeBrownie`, and look at how the different changes manifest in the nexus file itself.

3 Running Analyses

Now that you are familiar with the data structures in `RBrownie`, we can move on to using their methods effectively. `RBrownie` includes a number of high-level functions to perform the most common evolutionary analyses (discussed in the first three subsections), but also exposes a number of lower-level functions making it easier for the user to tweak or customize the commands the will eventually be run on the Brownie core.

`RBrownie` provides two general ways to run any of these analyses. First, they can be run directly by calling one of the pre-packaged functions of the form `runTEST`. These functions clear away anything currently in the commands block and run the `TEST` specified. Alternatively, commands can be added to a `brownie` objects one by one using functions of the form `addTEST` or `addOPTION` and then all the commands in the brownie object can be executed using `run.asis`. We'll see examples of all of these below.

3.1 Censored Rate Test

Briefly, the purpose of this test is to calculate and compare rate of continuous character evolution in different parts of a tree. A detailed description of this test is beyond the scope of this guide, but can be found in the Brownie manual.

Brownie uses the 'ratetest' command to perform this test allowing the user to specify a number of optional parameters and so function `RBrownie` provides functions which add this command along with various options to the `commands` slot of a `brownie` object. It requires the use of a `phylo4d_ext` object as input (see the manual) - Let's look at an example:

```
> require(RBrownie)
> data(parrot)
> test.results = runCensored(parrot, taxset = "intrajoint",
+     reps = 1000, charloop = T)
> summaryRatetest(test.results)
```

The last command, `summaryRatetest`, provides a summary of the ratetest results - it's a very handy command, providing a rough interpretation of the data in `test.results`. Again, the reader is referred to the brownie manual for a more detailed description of the data.frame and its summary.

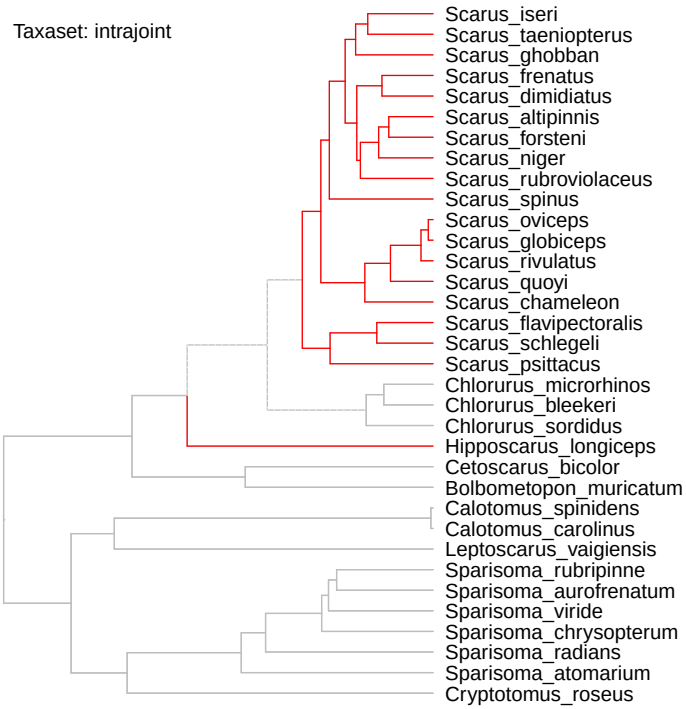
3.1.1 Plots

In addition to the `summaryRatetest` function, there is also a specialized plotting function for visualizing which branches of a tree the ratetest command will remove (see the brownie manual for more information on this). The branches to be removed are shown as dashed, the branches of the taxa in the taxaset 'intrajoint' are shown in red.

```
> require(RBrownie)
> data(parrot)
> taxaset.names(parrot) # we want to look at 'intrajoint', so use taxind=2

[1] "all"        "intrajoint"

> plot.censored(parrot[[2]],taxind=2)
```



3.2 Non-Censored Rate Test

The non-censored rate test is similar to the original censored version, except that where exactly on the branch the state changes matters to the likelihood estimate. Agian, describing the test in full is outside of the scope of this guide, but a full description is available in the Brownie manual and (O'Meara, et al., 2006).

Brownie uses the 'cont' command to perform this test - let's look at an example where we will run and compare two different models of morphological evolution (in this case, the two models are based on brownian motion, to view all the available models call the `brownie.models.continuous(T)` command):

```
> require(RBrownie)
> data(parrot)
> # brownie.models.continuous(T) will print out available models and descriptions
> test.results = runNonCensored(parrot,models=brownie.models.continuous()[1:2],
+         treeloop=T,charloop=T)
> summaryCont(test.results)
```

In the example above `runNonCensored` removes all commands currently in the `commands` slot and replaces them with custom commands and then executes the resulting `brownie` list.

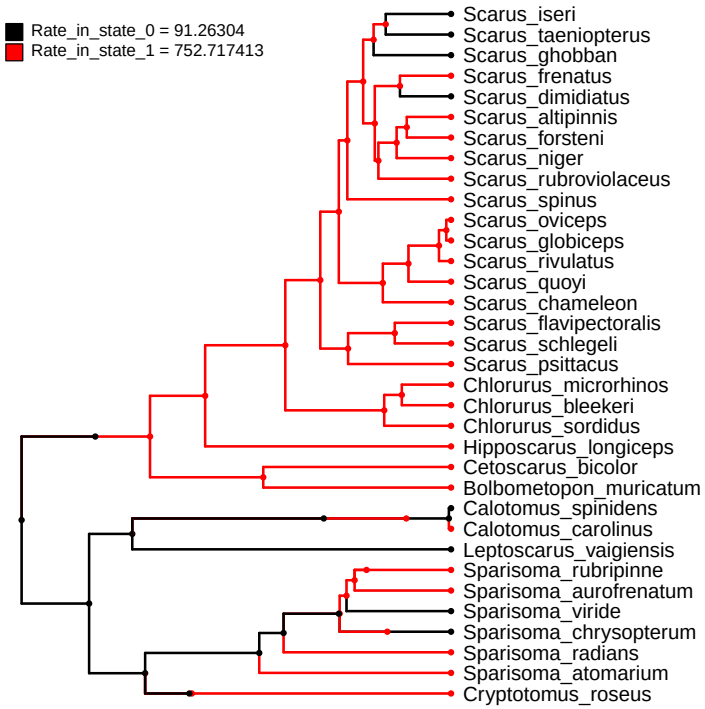
`test.results` is again a data.frame with the log-likelihood and and AIC values calculated for each tree, character, and model used and `summaryCont` prints out the model comparisons for each character, averaging over all the trees used. For more information about the test and how to interpret it, check out the Brownie manual and (Bodega Phylogenetics Wiki , 2010).

3.2.1 Plots

Some of the models tested return values that can be plotted on top of phylogenetic trees for easier visualization. One such model is "BMS", **B**rownian **m**otion with rate varying by state of the **s**tochastically. It returns two different evolutionary rates, one for when the internal state is 0 and one is 1.

```
> # run non-censored rate test
> require(RBrownie)
> data(parrot)
> test.results = runNonCensored(parrot,models=brownie.models.continuous()[1:2],
+         treeloop=T,charloop=T)
> #
> # Plot the evo rates from the BMS model
> # use tree #1 and character #1
> modname = "BMS"
> tree = 1
> char = 1
> x = parrot[[tree]]
> states = c(0,1)
> state.cols = c(1,2)
> line.widths = c(2,2)
> state.ratename = c("Rate_in_state_0","Rate_in_state_1")
> rowind = which(test.results$Tree==tree &
+         test.results$Char==char &
+         test.results$Model == modname)
> junkrow = test.results[rowind,]
> colinds = which(colnames(junkrow) %in% state.ratename)
> legends = paste(names(junkrow[colinds]),junkrow[colinds],sep=" = ")
> plot(x,states=states,states.col=state.cols,line.widths=line.widths,plot.points=T)
> #
> # Add jury-rigged legend
> fontsize=10
> boxSize = unit(fontsize, "points")
> for(ii in seq(length(states)))
+ {
+         yoff = unit( ((ii-1)*fontsize)+ ifelse(ii==1,0,2) ,"points")
+         grid.text(legends[ii],
+         x=unit(2, "mm"),
+         y=unit(1, "npc") - unit(2, "mm") - yoff,
```

```
+          just=c("left", "top"),
+          gp=gpar(fontsize = fontsize))
+          grid.rect(x=unit(2, "mm") - boxSize,
+          y=unit(1, "npc") - unit(2, "mm") - boxSize - yoff,
+          width = boxSize, height = boxSize,
+          just = "bottom", gp = gpar(fill = state.cols[ii]))
+ }
```



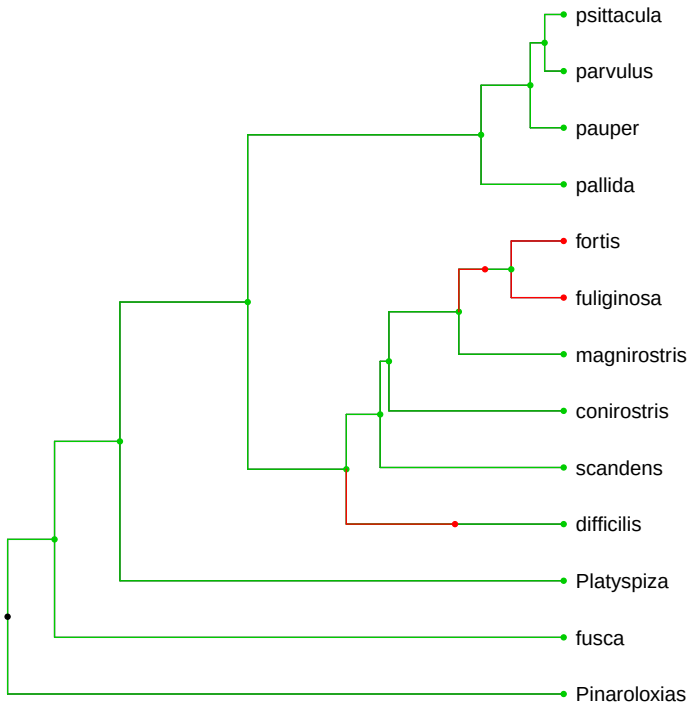
3.3 Discrete Ancestral State Reconstruction

Brownie can do discrete character reconstructions about which more information can be found here. Also check out the `addDiscrete` documentation.

Brownie uses the 'discrete' command to run these tests. Unlike the Censored and Non-Censored rate tests, this command simply requires a phylogenetic tree (with no branch annotations) and discrete data at the tree tips. Let's see an example using the geospiza_ext dataset built into `RBrownie`:

```
> require(RBrownie)
> data(geospiza_ext)
> junkrun=runDiscrete(geospiza_ext,
+          models=c("nonrev","nonrev"),
+          freqs=c("unif","equilib"),
```

```
+         reconstruct=T)
> #
> # (View the reconstruction results for the first tree;
> #  under the model 'nonrev' and using freq 'unif'):
> plot(junkrun$trees[[1]])
```



Unlike the previous two analyses, `runDiscrete` returns a list containing the trees with ancestral state reconstructions and a data.frame giving details about the support each model/freq has. More details about the discrete reconstruction function (and options that can be passed to the run function above) can be found by typing `?addDiscrete`.

3.4 Custom Tests

The examples above show how to execute the three most common commands `RBrownie` has to offer, using function of the format `runTEST`. In all cases, it is possible to manually put together an analysis using `addTEST` and `addOPTION` commands. There are add commands for most Brownie options and commands - to view these, see:

```
> help.search("add", package = "RBrownie")
```

Once an the `commands` slot of a brownie object has been filled with all the commands you want executed, the object can be executed directly using

`run.asis`. A further vignette aimed at power users will describe in detail how to piece together custom analyses, so we will leave this thread here. Check back fo

4 Getting Help

We'd like to invite anyone who has questions about how to use `Brownie` or `RBrownie` to post them to the Brownie-users google group. Also, we are continually updating `RBrownie` and would love your feedback on how it can be improved. Feel free to also suggest new features, bring bugs to our notice, or give general feedback - it would be appreciated!

5 Final Thoughts

More work is currently in progress to have `RBrownie` support more branch annotation formats (those currently supported by Mesquite and BEAST) and work being done to support returning confidence intervals for rate tests and reconstructions. Check back soon to see if there are any updates or more detailed tutorials.

References

Price, S., Wainwright, P. (2010). Testing for different rates of continuous trait evolution using likelihood. (http://bodegaphylo.wikispot.org/Morphological_Diversification_and_Rates_of_Evolution)

O'Meara, B. C., Ane, C., Sanderson, M. J. and Wainwright, P. C. (2006). Testing for different rates of continuous trait evolution using likelihood. In *Evolution*, 60 ,922–933.