RBrownie User's Guide (Version 0.0.4)

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

August 29, 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). More specifically, the original Brownie function was a censored rate test under Brownian motion, which allows the comparison of the rates of evolution (under brownian motion) 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).

As Brownie has grown, more and related analyses have been added. Now it is also possible to perform "non-censored" ratetests and discrete character reconstructions. The former allows you to compare the rates (Brownian motion model) or means (Ornstein-Uhlenbeck model) of evolution in different character states that are mapped (or painted) onto the phylogeny. For example, say you had a tree with binary characters mapped onto it branches indicating whether or not ancestors along that branch were thought to have or not have a certain trait and you also had morphological data for all the taxa present in the tree (the tips). Using a model-testing approach, Brownie allows to you statistically assess whether or not that binary trait is significantly correlated with the rate of evolution of the morphological characteristic measured (or set of morphological characteristics). Brownie also 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 How to run

Over it's lifetime Brownie has evolved from a MATLAB module into a command-line executable available to anyone with a suitable compiler. In it's current iteration, Brownie 2.0, is available in executable form for Macintosh operating systems and as source code which can be installed onto most other operating systems using gcc and the GSL. (Of course, we encourage you to check out RBrownie as it automates the install process and supports most Brownie functions!)

1.2 Input / Output

The Brownie command-line program is very similar in look, feel, and structure to command-line PAUP*. It reads NEXUS-formatted files including, like PAUP*, a special, proprietary nexus block (BROWNIE) which holds a list of the commands to be executed by the program itself. We mention this because `RBrownie` reads and writes these specialized nexus files and any files which `RBrownie` outputs are directly executable by the Brownie command-line program.

1.3 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. It simply builds brownie as a static library and automates the piping and execution of strings that would normally be entered at the Brownie's command line. As we will see later, the `brownie` S4 class, one of RBrownie main classes, has a slot for "commands" which holds these strings. In addition to providing this more streamlined interface to the Brownie core functions, RBrownie adds plotting and visualization functions which make working with output data much easier.

In order to process the specially formatted nexus files required by Brownie and manipulate their contents from within R, RBrownie has supplemented R's already rich phylogenetic toolkit with a number of novel data structures and methods. It literally extends a number of classes and methods from `phylobase`, on which, it could be said, RBrownie is roughly based. To see what other R

packages RBrownie uses, load up the package itself which you'll need to do at some point anyway.

```
> require(RBrownie)
```

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 trees and sequence data, while phylobase also reads CHARACTERS (or DATA) blocks. For more information on the options presented by these too packages, check out their documentation files:

```
> ?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 it's 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`.

```
> class(parrot)
> class(parrot[[1]])
```

The list class (a very common class in R) is used when multiple trees are found in a single nexus file. RBrownie represents them as a list of objects which all are of class `brownie` - these are discussed in the next section. The `readNexus` function in phylobase has the same behavior.

2.2 Classes in RBrownie

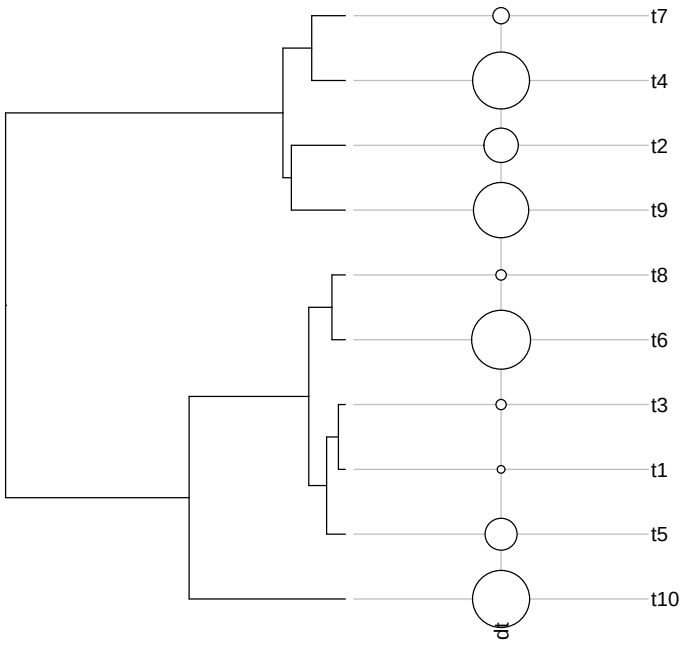
At it's core RBrownie is built around two major classes, `phylo4d_ext` and `brownie`, which both extend `phylo4d` from phylobase. In fact, `phylo4d_ext` extends `phylo4d` and `brownie` in turn extends `phylo4d_ext`. In non-computerese this mean that `phylo4d_ext` adds a few new data containers (or "slots"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

```
> require(RBrownie)
> phylo4d_slots = names(getSlots("phylo4d"))
> phylo4d_ext_slots = names(getSlots("phylo4d_ext"))
> brownie_slots = names(getSlots("brownie"))
> phylo4d_slots
> setdiff(phylo4d_ext_slots, phylo4d_slots)
> setdiff(brownie_slots, phylo4d_ext_slots)
```

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 phylogenetic tree and add junk data to the tips of tree which in practice might represent morphological data for each of the taxa.

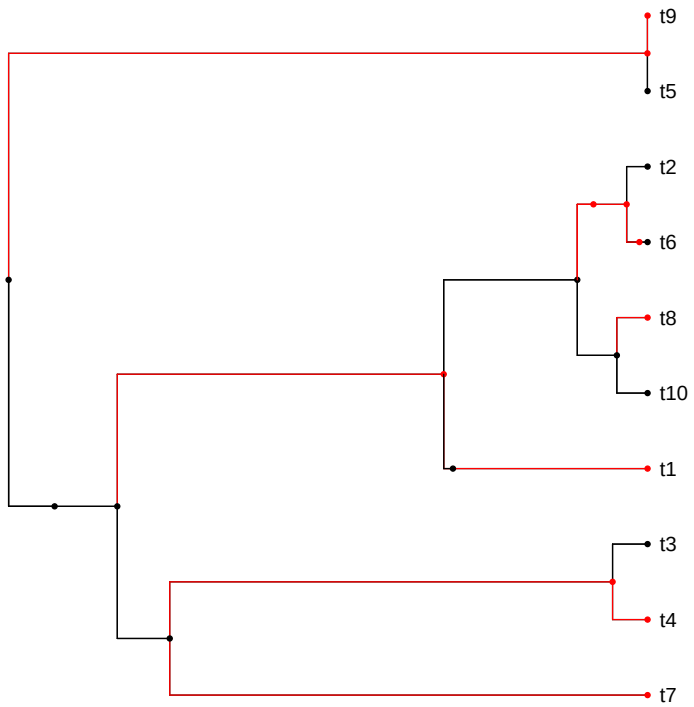
```
> require(phylobase)
> # generate a random coalescent tree with 10 tips
> ape_tree = rcoal(10)
> # convert tree from 'phylo' (ape) to 'phylo4' (phylobase) format
> phy_tree = as(ape_tree,"phylo4")
> # generate junk data for the tips
> sample_tipdata = runif(10)
> # combine the phylogenetic tree and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

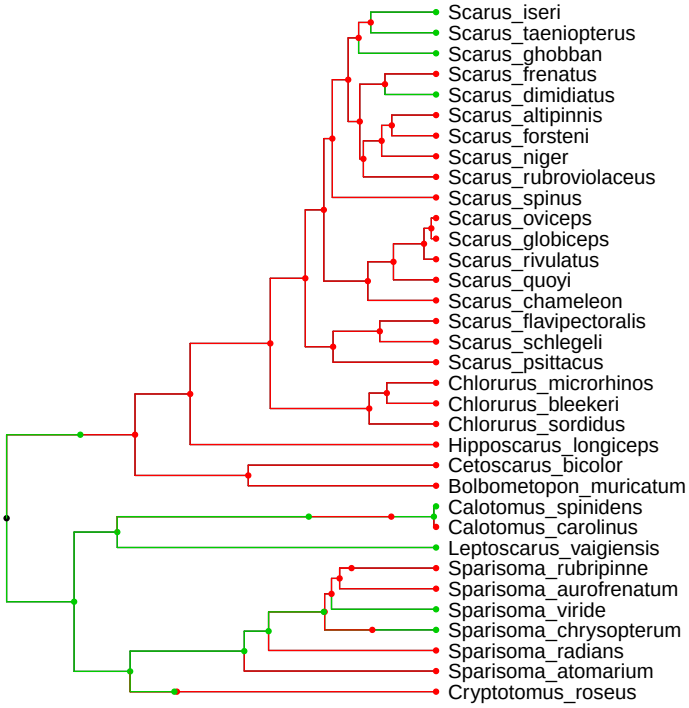
```
> require(RBrownie)
> ape_tree = rcoal(10)
> phy_tree = as(ape_tree,"phylo4")
> #
> ### create tip data
> #
> # create binary data indicators for all nodes (including internal)
> sample_binarydata = data.frame(sample(c(0,1),19,replace=TRUE))
> names(sample_binarydata) <- "hasTrait"
> #
> #
> ### subnodes
> sample_subnodedata = data.frame(sample(c(0,1),4,replace=TRUE))
> #
> # should be the same name as the parent data.frame
> names(sample_subnodedata) <- "hasTrait"
> #
```

```
> # add subnodes to 4 random branches
> sample_edges = sample(seq(nrow(edges(phy_tree))),4)
> #
> # could positions for the subnodes along their branches
> # (as a fraction of the overall branch length)
> sample_positions = runif(4)
> # note that "extra" arguments (all.data) are passed
> # on to the phylo4d constructor
> 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. This is usually used to indicate state changes along a branch, but because of the randomly generated data above you might see an entirely red branch with a red subnode or something similarly odd. A better example might 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 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)

Anyhow, to view subnode information these accessor functions are available.

```
> snid(phyext_tree)
> sndata(phyext_tree)
> snposition(phyext_tree)
> snbranch(phyext_tree)
> showSubNodes(phyext_tree)
```

And to add a new subnode manually, try this

```
> # place the node along the first branch
> ancestor.node = edges(phyext_tree)[1,1]
> descendant.node = edges(phyext_tree)[1,2]
> node.position = 0.5 # half way along the branch
> node.data = data.frame(1) # give it state = 1
> names(node.data) <- "hasTrait"
> phyext_tree = addSubNode(phyext_tree,
+                                  ancestor.node,
+                                  descendant.node,
+                                  node.position,
+                                  node.data)
```

You might notice that the subnode position is represented currently as a two-column matrix. In a future version, the subnodes will float between two values representing our confidence on where the state change (and thus the subnode) might occur. `weight` is another aspect of the `phylo4d_ext` class that will be supported better in future versions.

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` - we are working to add support for others. `RBrownie` has functions for reading and / writing in this format with functions similar to `ape`'s `read.nexus` and `write.nexus` functions. (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). We can confirm that this is the case:

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

`brownie` simply adds a few new slots to `phylo4d_ext`. The two classes were not combined into one larger class as we thought that the extended phylo4d class would hopefully have some use outside of RBrownie and furture development should reflect that. Moving on though, the two new slots are `commands`, to hold brownie text commands, and `datatypes`, which adds annotations to the `data` slot.

First, the `commands` slot. It's purpose 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.

For all but the most advanced users however, these commands will be filled in automatically by some higher level functions (seen later on) 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 number of accessor functions set up for you to do this

```
> require(RBrownie)
> data(geospiza_ext)
> commands(geospiza_ext)

[1] "discrete model= nonrev freq=unif reconstruct=yes file=nonrev_unif.txt replace=y;"
[2] "discrete model=nonrev freq=equilib reconstruct=yes file=nonrev_equil.txt replace=y;"

> # Removal
> geo_none1 = clearCommands(geospiza_ext) # remove all commands
> geo_none2 = removeCommands(geospiza_ext,"discrete") # remove all 'discrete' commands
> geo_3 = removeCommands(geospiza_ext,1) # remove the first command
> # Addition
> geo_add = geospiza_ext
> # add a command to the end of the vector
> commands(geo_add,add=TRUE) <- "choose t=1"
> # add a command at a certain index
> commands(geo_add,add=TRUE,index=1) <- "choose c=1"
> # replace ALL commands with this one
> commands(geo_add,replace=TRUE) <- "choose d=1"
> #
> geo_add = geospiza_ext
> # replace command at a certain index with this one
> commands(geo_add,add=TRUE,index=1,replace=TRUE) <- "choose d=1"
```

Second, the `datatypes` slot. `datatypes` 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, it's 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 it's 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()
> taxaData()
> genericData() # undefined data (shouldn't be used)
> brownie.datatypes() # show all
```

Adding new data to a `brownie` object is almost identical to how it is done in phylobase - the `addData` function is actually specified for class `brownie` - with the exception that you can specify which datatype you want to label it as. For example:

```
> 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)
> parrot_new = addData(parrot, tip.data=junkcont, known.types=contData())
> parrot_new = addData(parrot_new, all.data=junkdisc, known.types=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.

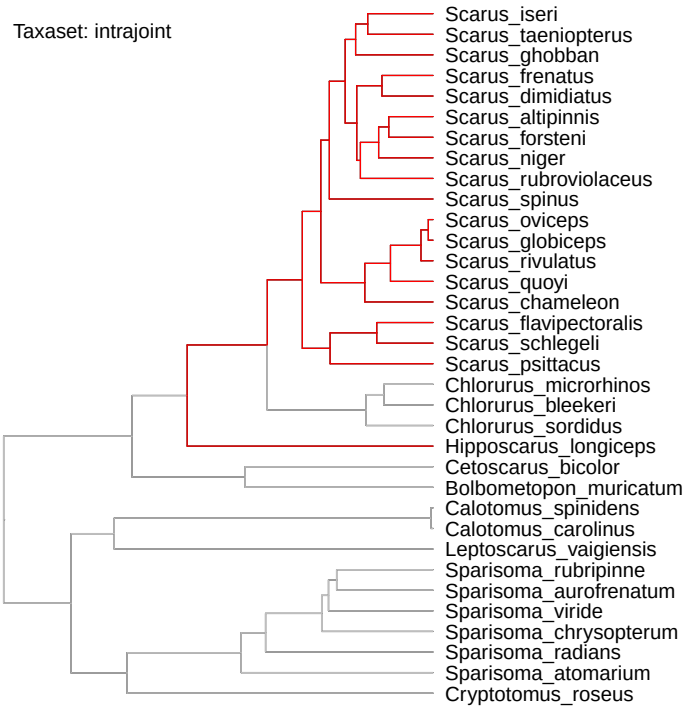
Adding taxa set data is a bit different. There is an accessor function in `RBrownie` called `taxasets` which gives the user access to all the data columns representing sets of taxa. It can also be used to create new taxasets:

```
> hasTaxasets(parrot) # see if there are any taxasets
> taxasets(parrot) # return taxasets as data.frame
> #
> # use binary data to indicate membership in a taxa set
> junktaxaset = sample(c(0,1),nTips(parrot[[1]]),replace=TRUE)
> taxasets(parrot,taxnames="Random_set") <- junktaxaset
> #
> # OR
> #
> # 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
```

And you can plot a taxaset as well:

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

Renaming internal nodes (might take a while if there are a lot of taxa):
```



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)
> par.new = clearCommands(parrot) # empty the commands slot
> par.new = addCensored(par.new,reps=1000,
+                                          taxset=taxaset.names(par.new)[2],charloop=TRUE)
> commands(par.new)

[1] "ratetest taxset=intrajoint reps=1000 treeloop=no charloop=yes quiet=no;"
```

As you can see, the 'ratetest' command has been added to all the `brownie` objects in the parrot list. Now that we have added this command, we can run the analysis manually:

```
> # run the brownie objects as is and get the text
> # output from the brownie core
> rawtxt = run.asis(par.new)$textout
> rawtxt = scan.textout(rawtxt)  # process text output
> test.results = read.ratetest.output(txt=rawtxt)
> 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 it's summary. But getting back to RBrownie, there is a more straight-forward way to run this test than the manual method used above that adds commands and processes the text output automatically. Here it is:

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

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

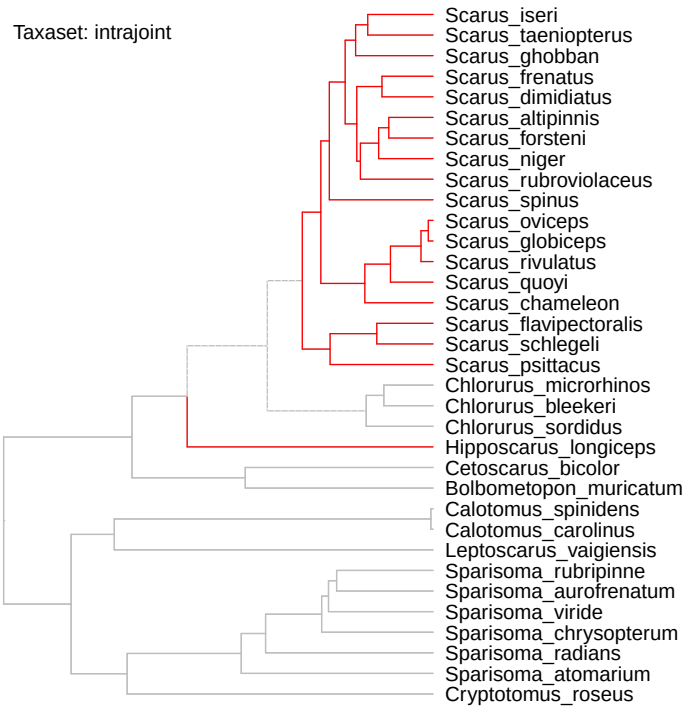
will remove (see the brownie manual for more information on this). The branches to be removed are shown as dashed, the branches of the taxa in the taxaset 'intrajoint' are shown in red.

```
> 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)

Renaming internal nodes (might take a while if there are a lot of taxa):
```



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):

```
> require(RBrownie)
> data(parrot)
> par.new = clearCommands(parrot) # get rid of current commands
> par.new <- addModel(par.new,"BM1")
> par.new <- addNonCensored(par.new,treeloop=T,
+         charloop=T,taxset="all",filereplace=T)
> par.new <- addModel(par.new,"BMS")
> par.new <- addNonCensored(par.new,treeloop=T,
+         charloop=T,taxset="all",fileappend=T,filereplace=F)
> commands(par.new)

[1] "model type=BM1;"
[2] "cont taxset=all treeloop=yes charloop=yes;"
[3] "model type=BMS;"
[4] "cont taxset=all treeloop=yes charloop=yes;"
```

As you can see, in the non-censored case it is necessary to add more than one command - two for the two different models and two to instruct the Brownie core execute a Non-censored ratetest using which ever continuous evolution model is set. Executing the whole bunch is similar to the censored test:

```
> # run the brownie objects as is and get the text
> # output from the brownie core
> rawtxt = run.asis(par.new)$textout
> rawtxt = scan.textout(rawtxt)  # process text output
> test.results = read.continuous.output(txt=rawtxt)
> summaryCont(test.results)
```

`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).

Again, there is a more direct way to execute a non-censored rate test:

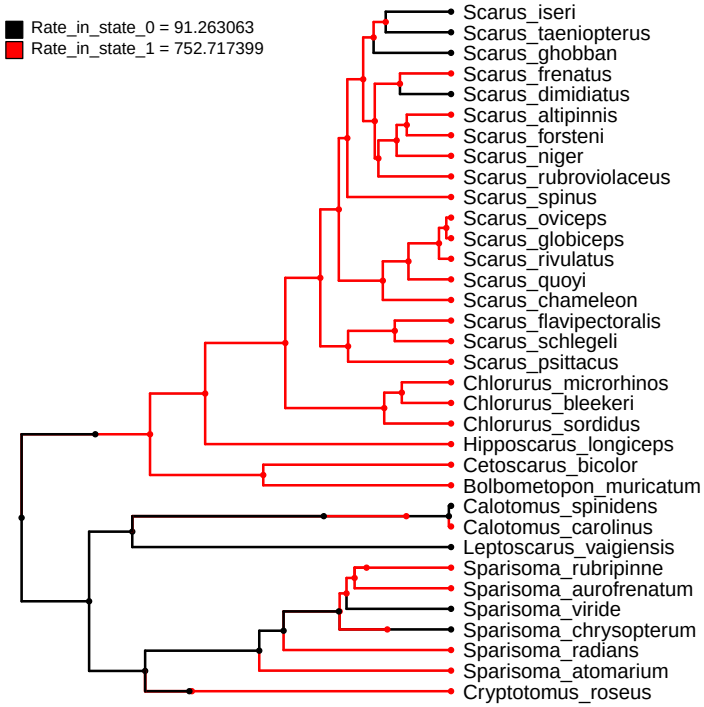
```
> require(RBrownie)
> data(parrot)
> # brownie.models.continuous(T) will print out available models and descriptions
> test.results = runNonCensored(parrot,models=brownie.models.continuous()[1:2],
+         treeloop=T,charloop=T)
> summaryCont(test.results)
```

In the example above `runNonCensored` removes all commands currently in the `commands` slot and replaces them with the same commands we added earlier manually and then executes the resulting `brownie` list (all within the function). In returns the same data.frame.

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)
> geoext=clearCommands(geospiza_ext)
> #
> # brownie.models.discrete(T), to see avail. models
> # brownie.freqs(T), to see avail. freq. models
> geoext=addDiscrete(geoext,
+         model="nonrev",
+         freq="unif",
+         reconstruct=T,
+         filereplace=T)
> geoext=addDiscrete(geoext,
+         model="nonrev",
```

```
+         freq="equilib",
+         reconstruct=T,
+         fileappend=T)
> commands(geoext)

[1] "discrete model=nonrev freq=unif treeloop=no charloop=no allchar=no variable=no reconstr
[2] "discrete model=nonrev freq=equilib treeloop=no charloop=no allchar=no variable=no recon
```

At this point we have essentially re-created the geospiza_ext object. Now we will manually run the analysis:

```
> rawout = run.asis(geoext)
```

If we look at the `rawout` we will see that both list elements `textout` and `treesout` are non-null. The `treesout` element includes the trees with reconstructed discrete values mapped on to either branches - in this case two model,freqs combinations were analyzed so two tree reconstructions are returned. But first, process the text output using some familiar commands.

```
> rawtext = scan.textout(rawout$textout)  # process text output
> test.results = read.discrete.output(txt=rawtext)
```

And the trees can be retrieved like so:

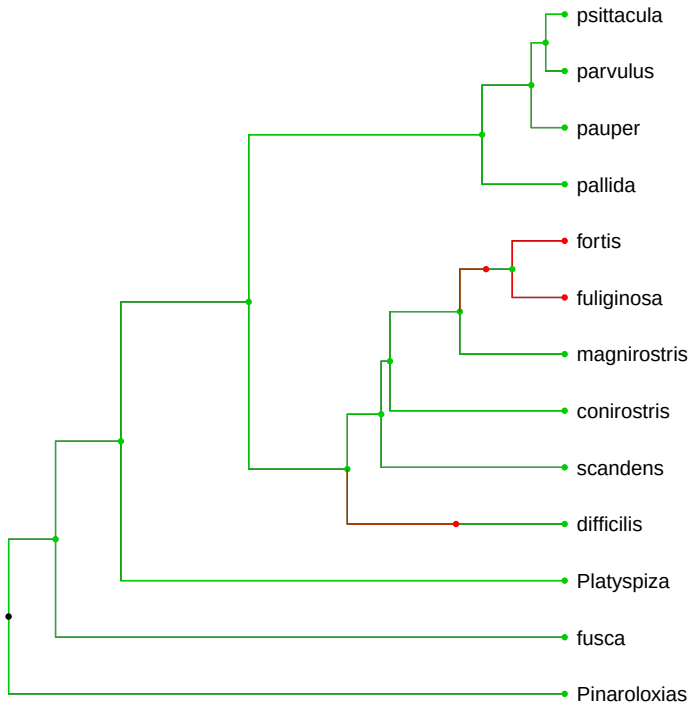
```
> # condition brownie output, by removing the ticks it adds
> treesout = gsub("'","",rawout$treesout)
> #
> # retrieve phylo4d objects with subnodes as singletons
> tree1 = read.simmap(text=treesout[1])
> tree2 = read.simmap(text=treesout[2])
> #
> # Convert singletons to subnodes
> # (and phylo4d objects to phylo4d_ext objects)
> tree1.ext = phyext(tree1)
> tree2.ext = phyext(tree2)
> #
> # Visualize the results:
> plot(tree1.ext)
> plot(tree2.ext)
```

Once more, there is a less round-about way to run this analysis:

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

```
Analysis returned some warnings:
[1] "WARNING: if some characters have fewer than 2 character states, the output in the outpu
[2] "WARNING: if some characters have fewer than 2 character states, the output in the outpu

> #
> # View the reconstruction results for the first tree:
> plot(junkrun$trees[[1]])
```



3.4 Custom Tests

As shown above all of the `runTEST` commands can also be put together manually using `addTEST` and `addOPTION` commands. There is an add command for most Brownie options and commands - to view the available options, see:

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

4 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.

We are continually updating `RBrownie` and would love your feedback on how it can be improved. If there is a feature you would like to see supported in either Brownie or RBrownie, please post a feature request to the brownie-users google group.

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.