Hy-Phy Users Manual

Spencer Muse [1]
North Carolina State University

Sergei Kosakovsky Pond [2]
University of Arizona

May 31, 2000

---

[1] Program in Statistical Genetics, Department of Statistics, Campus Box 8203, NC State University, Raleigh NC 27695-8203

[2] Program in Applied Mathematics, Department of Mathematics, University of Arizona, Tucson AZ 85721

Abstract

Hy-Phy is a high-level programming language for performing molecular evolutionary analyses. Although ***Hy-Phy*** has a growing number of tree reconstruction features, its primary design is not to reconstruct phylogenetic trees. Instead, the object of ***Hy-Phy*** is to provide a flexible platform for studying rates and patterns of sequence change.

Chapter 1

Using the Shell Interface

1.1 Getting Help

The documentation for ***Hy-Phy*** is a work in progress. However, there are several sources of information in addition to this basic document. An html-based help system is available, and should be available as part of the basic distribution. Depending on your platform, it may be available through the Help menu. There is also online help available at the ***Hy-Phy*** home page. Both sources include an exhaustive listing of ***Hy-Phy*** functions, commands, constants, and syntax.

1.2 Starting *Hy-Phy*

Macintosh and Windows

To start ***Hy-Phy*** simply double-click the program icon located in the distribution folder.

UNIX

If you installed ***Hy-Phy*** in your home directory, execute the command `HYPHYKernel` from the prompt. If ***Hy-Phy*** is installed in a shared directory in your system, the system administrator should have set up a `HYPHY` alias which can be used to start up the program. If you wish to run ***Hy-Phy*** from a directory other than its installation directory, execute the command `'path to` *Hy-Phy*/HYPHYKernel BASEPATH=path to ***Hy-Phy***'.

Running *Standard Analyses*

Macintosh and Windows

Standard analyses can be accessed via the `Analysis` menu. A dialog box which contains the list of currently available analyses will appear and you can select an item from the list. There will also be a brief description of each item in the list for you to peruse. For complete details of what each standard analysis does, refer to the relevant topics in ***Hy-Phy*** documentation.

Note that if the standard analyses dialog box comes up empty, the most likely cause is that ***Hy-Phy*** can't find its `TemplateBatchFiles` folder, which should be in the same folder as ***Hy-Phy*** itself.

UNIX

When ***Hy-Phy*** starts up, its default behavior is to scan for and display the list of currently available standard analyses. You can select a standard analysis to run by typing in its abbreviation (specified in the list).

Hy-Phy will not display the list of standard analysis in 2 cases:

1. A batch file to process was passed to ***Hy-Phy*** via a command line argument
2. ***Hy-Phy*** could not locate the `TemplateBatchFiles` directory.

   If `TemplateBatchFiles` is not in the current working directory, you may need to explicitly tell ***Hy-Phy*** where the standard analyses are located. To do so, use the

   `BASEPATH=absolute path to` *Hy-Phy* installation directory

   (no spaces and all capital letters in BASEPATH) command line argument.

   For example, if you running ***Hy-Phy*** from the directory */home/joeuser/*, and the ***Hy-Phy*** installation is located in */usr/local/HYPHY/*, then you would execute (assuming that */usr/local/HYPHY/* is in the search paths),

   HYPHYKernel BASEPATH=/usr/local/HYPHY/

   In most cases, there should be a system wide alias which will automatically include the BASEPATH argument.

   All standard analyses include prompts from data files and similar input. A typical prompt will look like this:

   Select a nucleotide file(/home/joeuser/):

   The path in parentheses is the current base for ***Hy-Phy*** relative path names. For instance, if you supply *data/50seq.nuc* as the input to the prompt, ***Hy-Phy*** will look for the file */home/joeuser/data/50seq.nuc*. You can specify an absolute path name as input to any file related prompts, and use ../ in relative paths to refer to parent directory(ies).

Running Prewritten Batch Files

Macintosh and Windows

Batch Files in *Examples*. To run a batch file from *Examples*, select *Run Batch File* from the *F*ile menu, and choose the file you wish to process via a standard file selection dialog box. The *Examples* folder should be located in the same folder as ***Hy-Phy*** itself.

Other Batch Files. To run any batch file, select *Run Batch File* from the *F*ile menu, and choose the file you wish to process via a standard file selection dialog box. By default, ***Hy-Phy*** Batch files have extensions .bf. Please note that some batch file may write their output directly to a file, rather than to the screen.

UNIX

Batch Files in *Examples* and other Batch Files. There are two ways to tell ***Hy-Phy*** to run a specific batch file:

1. Pass the name/path to the file as a command line argument.

   For example, if current working directory is */home/joeuser/*, and you wish to run the file */home/joeuser/BatchFiles/joes.bf* the command to execute would be:

   HYPHYKernel BatchFiles/joes.bf

   or

   HYPHYKernel /home/joeuser/BatchFiles/joes.bf.

2. Start ***Hy-Phy*** and when prompted to select a standard analysis, press *Enter* to skip the selection. A prompt

   Process Bath File:

   should appear, and you can direct ***Hy-Phy*** to execute a particular batch file by giving a relative or absolute path to the file, as in the example above.

Chapter 2

Basics

Hy-Phy is very feature rich, as its primary purpose was to serve as the authors' research tool. Therefore, there are many features that most users will never implement. In fact, many, if not most users, will rely almost totally on prewritten ***Hy-Phy*** "batch files" (lines of code that ***Hy-Phy*** interprets, much like a SAS program). To introduce the major components of the ***Hy-Phy*** language, we will first step through some very basic batch files so that the core batch file elements are described.

2.1 Simple Batch Files

Most batch files will include a few key elements:

1. Read a data file
2. Select the species and characters to be analyzed (Filter the data)
3. Tabulate or define frequencies of characters
4. Describe the form of a substitution matrix
5. Combine the character frequencies and substitution matrix into an evolutionary model for your characters
6. Describe a phylogenetic tree
7. Define a likelihood function based on the tree, data, and model.
8. Maximize the likelihood function
9. Print the results to the screen and/or an output file.

Let us begin with a very simple example. In the following batch file we will fit a set of DNA sequence data from 4 species to an unrooted tree using the F81 (Felsenstein 1981) model of sequence evolution. (The batch file is named "basics.bf" and should be in the *Tutorial* directory of the ***Hy-Phy*** distribution.)

```
DataSet myData = ReadDataFile ("data/demo.seq");
DataSetFilter myFilter = CreateFilter (myData,1);
HarvestFrequencies (obsFreqs, myFilter, 1, 1, 1);
F81RateMatrix =
               {{* ,mu,mu,mu}
                {mu,* ,mu,mu}
                {mu,mu,* ,mu}
                {mu,mu,mu,* }};
Model F81 = (F81RateMatrix, obsFreqs);
Tree myTree = ((a,b),c,d);
LikelihoodFunction theLikFun = (myFilter, myTree, obsFreqs);
Optimize (paramValues, theLikFun);
fprintf  (stdout, theLikFun);
```

First of all, notice that the nine lines of code in ”basics.bf” (the comment lines have been removed in the above display) correspond to the nine steps enumerated previously. Let us consider each of them.

Read a data file.

The line of code

```
DataSet myData = ReadDataFile ("../data/demo.seq");
```

reads data from the file *demo.seq* (located in the *data* directory) and stores the data in a structure called *myData*. The line of code uses two ***Hy-Phy*** commands, *DataSet* and *ReadDataFile*. The first part of the program statement, `DataSet myData`, simply tells ***Hy-Phy*** to create a structure named *myData* that will hold a data set. The second part of the statement,

```
ReadDataFile("../data/demo.seq")
```

actually reads and processes the data in the file *demo.seq*.

Select the species and characters to be analyzed (Filter the data)

The line of code

```
DataSetFilter myFilter = CreateFilter (myData,1);
```

performs two important tasks. First, it selects the precise data elements from *myData* that we want to use for our analysis. In this case, we use all of the data by default by ignoring some optional arguments to the *CreateFilter* command. See Chapter 5. to learn more about the features of *CreateFilter*. The second important task is to let ***Hy-Phy*** know how we want to

”interpret” the data. The argument ”1” indicates that each individual nucleotide is to be treated as a single character. Had we instead used ”3”, each consecutive triplet of nucleotides would be considered as a ”character”. Obviously, this type of filtering might be of use for analyzing codon data. If one wanted to analyze dinucleotides, the second argument to *CreateFilter* would be ”2”. Note that multiple data set filters can be created from a single data set. We will see several examples of creating data set filters in later examples.

Tabulate or define frequencies of characters

For likelihood calculations it is necessary to define the equilibrium frequencies of characters. In most cases we will estimate these from the data, often by simply using the observed empirical frequencies. The statement

```
HarvestFrequencies (obsFreqs, myFilter, 1, 1, 1);
```

tabulates the nucleotide frequencies in *myFilter* and stores them in a vector named *obsFreqs.*

Had we wanted to use different frequency values for some reason, *HarvestFrequencies* statement would simply have been replaced with something like

```
myFavoriteFreqs = {{.10}{.40}{.20}{.30}};
```

This statement would create a vector named *myFavoriteFreqs* with elements 0.10, 0.40, 0.20, and 0.30. *myFavoriteFreqs* could be used just like the variable *obsFreqs.*

Describe the form of a substitution matrix

One of the unique strengths of ***Hy-Phy*** is its ability to implement any special case of a general time reversible model, regardless of the dimensions of the character set. We will devote an entire chapter to model specification (see ???). To accomplish this feat, we decompose the evolutionary model into two components: the character frequencies, and the substitution parameters. We rely on the fact that any special case of the general reversible model can be written in a form where entries in the substitution matrix are products of substitution parameters and character frequencies (see ???). In F81.bf we see one of the very simplest model specifications. The line

```
F81RateMatrix =
                {{* ,mu,mu,mu}
                 {mu,* ,mu,mu}
                 {mu,mu,* ,mu}
                 {mu,mu,mu,* }};
```

in conjuction with a frequency vesctor such as *obsFreqs*, is sufficient to define the F81 model. Note that the syntax of the matrix definition consists simply of the rows of a matrix. For the F81 model, the instantaneous rate matrix is traditionally denoted

$$
\begin{array}{c|cccc}
 & A & C & G & T \\
\hline
A & -\mu(1-\pi_A) & \mu\pi_C & \mu\pi_G & \mu\pi_T \\
C & \mu\pi_A & -\mu(1-\pi_C) & \mu\pi_G & \mu\pi_T \\
G & \mu\pi_A & \mu\pi_C & -\mu(1-\pi_G) & \mu\pi_T \\
T & \mu\pi_A & \mu\pi_C & \mu\pi_G & -\mu(1-\pi_T)
\end{array}
$$

Observe the similarity between this matrix and the ***Hy-Phy*** syntax. The ***Hy-Phy*** operator * is defined as ”the negative of the sum of all non-diagonal entries on the row”. (Rate matrices have the property of row elements summing to zero.)

Define an evolutionary model

We now combine the results of the two previous steps to obtain a model of sequence evolution:

```
Model F81 = (F81RateMatrix, obsFreqs);
```

The *Model* statement declares that the rate matrix *F81RateMatrix* and the vector *obsFreqs* will determine the evolutionary model, which we name *F81.*

Define a phylogenetic tree

Hy-Phy uses standard (Newick) tree definitions. Thus, the statement

```
Tree myTree = (((a,b),c),d);
```

defines a tree named *myTree* with four OTUs, or taxa, named a, b, c, and d. In this case, the names correspond to the names in the ***Hy-Phy*** batchfile. While this is not necessary, it is strongly recommended to avoid confusion. ***Hy-Phy*** will accept either rooted or unrooted trees; however, for most purposes rooted trees are automatically unrooted by ***Hy-Phy*** because likelihood values for unrooted trees are the same as those for rooted trees.

The ***Hy-Phy*** *Tree* data structure is much more complex than simply describing a tree topology. The *Tree* variable includes both topology information, as well as evolutionary model information. The default behavior of a *Tree* statement is to attach the most recently defined *Model* to all branches in the tree. Thus, it is *critical* that the *Model* statement appears before the *Tree* statement. We will discuss more advanced uses of the *Tree* statement later.

Define the likelihood function

The likelihood function for phylogenetic trees depends on the dataset, tree topology, and the substitution model (and its parameters). To define a likelihood function, we use a statement like

```
LikelihoodFunction theLikFun = (myFilter,myTree);
```

We name the likelihood function *theLikFun*, and it uses the data in *myFilter* along with the tree topology and substitution model stored in in *myTree*. Recall that the *Tree* structure *myTree* inherited the *Model F81* by default.

Maximize the likelihood function

Asking ***Hy-Phy*** to maximize the likelihood function is simple:

```
Optimize(paramValues,theLikFun);
```

finds maximum likelihood estimates of all independent parameters and stores the results in the matrix *paramValues*, which ***Hy-Phy*** creates automatically.

Print the results to the screen and/or a file

The simplest way to print the results of a likelihood maximization step is simply to print the likelihood function:

```
fprintf(stdout,theLikFun);
```

This C-like command prints the structure *theLikFun* to the default output device *stdout* (stdout is typically the screen). The results of this statement are the following:

```
Log Likelihood = -589.252801936419;
Tree myTree=((a:0.0320567,b:0.00598813)Node1:0.045548,c:0.0257871,d:0.0769886);
```

When asked to print a likelihood function, ***Hy-Phy*** first reports the value of the likelihood function. It follows with a listing of all estimated parameters. Any parameters associated with particular branches in a tree are reported in a tree-like description, as shown in the output above. Each of the branches in the unrooted phylogeny has an associated ”branch length”; branch lengths are defined as “the expected number of nucleotide substitutions per site”. Those values appear after the colon following the label for each branch. For example, the estimated branch length leading to the node ”b” is 0.00598813. Note that the internal node in the tree has been automatically named “Node1” by ***Hy-Phy*** and its associated branch length is 0.045548.

We will see many examples of this type output in the later examples. ***Hy-Phy*** also allows for more detailed user control of printed output, using a C-like fprintf syntax. The later examples will illustrate this functionality.

2.2 Exercises

1. ***Hy-Phy*** automatically recognizes a variety of file formats, including NEXUS and PHYLIP variants. To see more clearly what this statement does, create and execute a simple batchfile with only the following code:

```
DataSet myData = ReadDataFile ("../data/demo.seq");
fprintf(stdout,myData);
```

2. Create and execute a batchfile that uses the model of Jukes and Cantor (1969). The JC69 model differs from the F81 model only by imposing the assumption that all nucleotide frequencies are 0.25.

3. Users will often want to write batch files that are not aware of the name or location of the data file. One can ask ***Hy-Phy*** to prompt for a filename using the following syntax:

   ```
   SetDialogPrompt ("Please specify a nuleotide data or aminoacid file:");
   DataSet myData = ReadDataFile (PROMPT_FOR_FILE);
   DataSetFilter filteredData = CreateFilter (myData,1);
   ```

   Modify *basics.bf* to prompt the user for the data file.

Chapter 3

A Tour of Batch Files

3.1 Defining Substitution Models

Simple Nucleotide Models: ***modeldefs.bf***

One of the primary objectives of ***Hy-Phy*** is to free users from relying on the substitution models chosen by authors of software. While a relatively small set of model choices may be sufficient for performing phylogenetic analyses, having only a few potential models is often limiting for studies of substitution rates and patterns. To define a model in ***Hy-Phy***, one needs only to describe the elements in a substitution rate matrix. If the characters being studied have n states, the rate matrix is $n \times n$. For example, nucleotide models are 4×4; models of amino acid change are 20×20; codon-based models might be 64×64. ***Hy-Phy*** can work properly with any member of the class of general time reversible models. Instantaneous rate matrices in this class of models satisfy the condition $\pi_i R_{ij} = \pi_j R_{ji}$, where π_i is the equilibrium frequency of character i (for nucleotide data,) and R_{ij} is the ij^{th} entry in the instantaneous rate matrix. ***Hy-Phy*** comes with many predefined rate matrices for commonly used substitution models. You can find examples in the batchfiles under the *Examples* and *TemplateBatchFiles* directories of the ***Hy-Phy*** distribution.

To illustrate the basics of model definition, examine the batch file *modeldefs.bf* from the *Tutorial* directory:

```
SetDialogPrompt("Select a nucleotide data file:");
DataSet myData  = ReadDataFile(PROMPT_FOR_FILE);
DataSetFilter myFilter = CreateFilter(myData,1);
HarvestFrequencies(obsFreqs,myFilter,1,1,1);
F81RateMatrix  = {{*,m,m,m}{m,*,m,m}{m,m,*,m}{m,m,m,*}};
Model F81 = (F81RateMatrix, obsFreqs);
Tree myTree = ((a,b),c,d);
fprintf(stdout,"\n\n F81 Analysis \n\n");
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);
fprintf(stdout,"\n\n HKY85 Analysis  \n\n");
HKY85RateMatrix = {{*,b,a,b}{b,*,b,a}{a,b,*,b}{b,a,b,*}};
Model HKY85 = (HKY85RateMatrix, obsFreqs);
Tree myTree = ((a,b),c,d);
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);
```

This batch file illustrates two new concepts. First, and most importantly, the lines

```
HKY85RateMatrix = {{*,b,a,b}{b,*,b,a}{a,b,*,b}{b,a,b,*}};
Model HKY85 = (HKY85RateMatrix, obsFreqs);
```

illustrate the definition of a new substitution matrix. In this case, we have defined the model of Hasegawa, Kishino, and Yano (1985) and named the model HKY85. If you are familiar with the HKY85 model, you will probably recognize the form of the matrix: transitions occur with rate a and transversions occur with rate b, with each of those values multiplied by the appropriate nucleotide frequency. The second important point to note is that we must associate the model with a tree before we can do anything useful. In this case, we simply redefined the old tree to use the HKY85 model instead of the F81 model (Recall that a tree consists of both the topology and the substitution matrices attached to its branches). When the statement `Tree myTree = ((a,b),c,d);` is issued, the variable *myTree* is assigned the topology ((a,b),c,d) and the branches are assigned the HKY85 substitution model, which was the most recently defined *Model.* If we wanted to preserve the original variable *myTree*, we could simply have defined a new *Tree* structure using a command like `Tree myNextTree = ((a,b),c,d);`

Finally, for completeness, we created a new *Tree* and assigned it the F81 model and reproduced the original F81 analysis. These steps simply illustrate how predefined *Models* can be assigned to *Trees* using the *UseModel* command.

```
UseModel(F81);
Tree myTree = ((a,b),c,d);
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);
```

One of the most general models of nucleotide substitution is the general time reversible model (REV). The instantaneous rate matrix for the REV model is

$$R_{\text{REV}} = \begin{array}{c} \\ \texttt{A} \\ \texttt{C} \\ \texttt{G} \\ \texttt{T} \end{array} \begin{array}{c} \begin{array}{cccc} \texttt{A} & \texttt{C} & \texttt{G} & \texttt{T} \end{array} \\ \begin{pmatrix} -\sum & \theta_0\pi_C & \theta_1\pi_G & \theta_2\pi_T \\ \theta_0\pi_A & -\sum & \theta_3\pi_G & \theta_4\pi_T \\ \theta_1\pi_A & \theta_3\pi_C & -\sum & \theta_5\pi_T \\ \theta_2\pi_A & \theta_4\pi_C & \theta_5\pi_G & -\sum \end{pmatrix} \end{array}$$

It is simple to implement this model in ***Hy-Phy***

```
REVRateMatrix  = {{*,a,b,c}{a,*,d,e}{b,d,*,f}{c,e,f,*}};
Model REV = (REVRateMatrix, obsFreq);
```

does the job.

To illustrate these notions in a more usefule context, consider the batchfile *models.bf*:

```
SetDialogPrompt("Select a nucleotide data file:");
DataSet myData  = ReadDataFile(PROMPT_FOR_FILE);
DataSetFilter myFilter = CreateFilter(myData,1);
HarvestFrequencies(obsFreqs,myFilter,1,1,1);

equalFreqs = {{0.25}{0.25}{0.25}{0.25}};

myTopology = "((a,b),c,og)";

F81RateMatrix  = {{*,m,m,m}{m,*,m,m}{m,m,*,m}{m,m,m,*}};
Model F81 = (F81RateMatrix, obsFreqs);

HKY85RateMatrix = {{*,b,a,b}{b,*,b,a}{a,b,*,b}{b,a,b,*}};
Model HKY85 = (HKY85RateMatrix, obsFreqs);

REVRateMatrix  = {{*,a,b,c}{a,*,d,e}{b,d,*,f}{c,e,f,*}};
Model REV = (REVRateMatrix, obsFreqs);

Model JC69 = (F81RateMatrix, equalFreqs);

Model K2P = (HKY85RateMatrix, equalFreqs);

UseModel(JC69);
Tree myTree = myTopology;
fprintf(stdout,"\n\n JC69 Analysis \n\n");
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);

UseModel(F81);
Tree myTree = myTopology;
fprintf(stdout,"\n\n F81 Analysis \n\n");
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);

UseModel(K2P);
Tree myTree = myTopology;
fprintf(stdout,"\n\n K2P Analysis \n\n");
LikelihoodFunction theLikFun = (myFilter, myTree);
```

```
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);

UseModel(HKY85);
Tree myTree = myTopology;
fprintf(stdout,"\n\n HKY85 Analysis \n\n");
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);

UseModel(REV);
Tree myTree = myTopology;
fprintf(stdout,"\n\n REV Analysis \n\n");
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);
```

If you understand the components of *models.bf*, then you are ready to define substitution models, attach them to tree topologies, and find maximum likelihood estimates. *models.bf* also demonstrates a few useful ***Hy-Phy*** features. First, notice the definition of the simple vector (0.25,0.25,0.25,0.25). In a similar manner, we define the string constant *myTopology.* By changing the topology in the definition of *myTopology*, the entire analysis can be repeated using the new topology. This single step is faster than updating the topology for every *Tree* statement. Finally, note the reuse of the three substitution matrices and the two frequency vectors. The original matrix definitions are used as templates by the *Model* statements.

Global vesus local parameters: *localglobal.bf*

Interpretation. Because the primary goal of ***Hy-Phy*** is to provide flexible modeling of the nucleotide substitution process, ***Hy-Phy*** includes a more general parameterization scheme than most phylogeny estiamtion programs. Perhaps the most important difference for the user to recognize is the difference between *local* and *global* parameters. A local parameter is one that is specific for a single branch on a tree. In contrast, a global parameter is shared by all branches. To illustrate, consider the output generated by the batch file *localglobal.bf* when run using *(*demo.seq):

```
 Original HKY85 Analysis

Log Likelihood = -579.03975528721;
Tree myTree=((a{a=0.106343,b=0.024239},b{a=0.000000,b=0.007742})
          Node1{a=0.118759,b=0.041943},c{a=0.085374,b=0.022151},
             og{a=0.246895,b=0.050413});

 Local HKY85 Analysis

Log Likelihood = -579.039980005965;
Tree myTree=((a{R=4.350942,b=0.024447},b{R=0.000000,b=0.007724})
          Node1{R=2.835357,b=0.041878},c{R=3.828977,b=0.022304},
            og{R=4.876743,b=0.050627});

 Global HKY85 Analysis

Log Likelihood = -579.621469358837;
Shared Parameters:
V=3.772178

Tree myTree=((a{b=0.024878},b{b=0.004718})
          Node1{b=0.035926},c{b=0.020933},
            og{b=0.060754});
```

In *localglobal.bf* we have moved beyond the default settings of ***Hy-Phy***, and the details of the batch file will be discussed below. For now, concentrate on the results. *localglobal.bf* performs three analyses of the data in *demo.seq*, all using the HKY85 model of sequence evolution. The first, labeled "Original HKY85 Analysis", is the same analysis that was performed in the previous example (*models.bf*). Note that the output format is different. Rather than report the branch lengths for each branch in the tree, the individual parameter estimates are shown. Note that each branch has an associated value of a and b. These parameters control the relative frequencies of transitions and transversions in the HKY85 model. This output reveals that the original analysis was an example of a *local* analysis. In the context of this batch file, there was a *local* value of the transition-transversion ratio for each branch in the tree.

The second analysis in *localglobal.bf* repeats the original analysis, but uses a different formulation of the HKY85 model (see below). Note that the likelihood values differ only slightly between the "Original HKY85 Analysis" and the "Local HKY Analysis". The differences are the results of approximation error in the numerical optimization process. The parameter lists for the two analyses are also different. Except for approximation error, the values of b in the two analyses are the same. On inspection, one notices that the values of a in the first analysis are equal to the products of b and R in the second analysis. The second analysis is simply a reparametrization of the first one, making explicit use of the transition-transversion ratio, R.

The third analysis performed in *localglobal.bf* is an example of a *global* analysis. In contrast to the previous two analyses (which you should now understand were really replicates of a single analysis), the "Global HKY85 Analysis" invokes a *global* transition-transversion ratio, V. In other words, all branches share the same value of V. This fact is evident in the output for the "Global HKY85 Analysis". Note that the tree list contains only values of b for each branch. The estimated global value of V is shown under the heading of `Shared Parameters`.

The local and global analyses use different numbers of parameters. The local analysis uses a transition and transversion rate for each of the 5 branches, along with 3 base frequencies, for a total of 13 parameters. The global analysis includes a transversion rate for each branch, 3 base frequencies, and a single transition-transversion rate, for a total of 9 parameters. The global analysis is a special case of the local analysis; therefore, the log-likelihood value for the global analysis (-579.62) is lower than that of the local anaysis (-579.03). The fact that the addition of 4 parameters results in such a small difference in model fit suggests that the data harbor little support for the hypothesis that the transition-transversion rate varies among these lineages.

Implementation. The code for *localglobal.bf* is the following:

```
OPTIMIZATION_PRECISION=0.0001;
OPTIMIZATION_PRECISION_METHOD=1;

LIKELIHOOD_FUNCTION_OUTPUT = 4;

SetDialogPrompt("Select a nucleotide data file:");
DataSet myData  = ReadDataFile(PROMPT_FOR_FILE);
DataSetFilter myFilter = CreateFilter(myData,1);
HarvestFrequencies(obsFreqs,myFilter,1,1,1);

fprintf(stdout,"\n\n Original HKY85 Analysis  \n\n");
HKY85RateMatrix = {{*,b,a,b}{b,*,b,a}{a,b,*,b}{b,a,b,*}};
Model HKY85 = (HKY85RateMatrix, obsFreqs);
Tree myTree = ((a,b),c,og);
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);

fprintf(stdout,"\n\n Local HKY85 Analysis  \n\n");
LocalHKY85Matrix = {{*,b,b*R,b}{b,*,b,b*R}{b*R,b,*,b}{b,b*R,b,*}};
Model LocalHKY85 = (LocalHKY85Matrix, obsFreqs);
Tree myTree = ((a,b),c,og);
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);

fprintf(stdout,"\n\n Global HKY85 Analysis  \n\n");
global V=2.0;
GlobalHKY85Matrix = {{*,b,b*V,b}{b,*,b,b*V}{b*V,b,*,b}{b,b*V,b,*}};
Model GlobalHKY85 = (GlobalHKY85Matrix, obsFreqs);
Tree myTree = ((a,b),c,og);
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);
```

The first three lines of the file illustrate the use of three of the many constants that ***Hy-Phy*** uses to tailor its behavior. The first two affect the precision (and speed) of the optimization process; the third changes the format of the printed likelihood function. (For details, see the *Batch Language Command Reference*).

The code for the first analysis is identical to that from *models.bf*. You will note that the second analysis uses the same code, with the value of `a` replaced by `R*b`. The substitution matrix and model have been assigned different names for illustrative purposes. Again, these two parameterizations are equivalent.

The global analysis introduces a new statment: global V=2.0; This statement declares V to be a global variable. By default, the description of a model (and variables within that model) is used as a template that is copied for every branch on the tree. An important fact is that we can not simply redefine R as a global variable. The scope of a variable is determined at the time of its creation and can not be altered. Thus, it was necessary to create the new variable, V.

3.2 More complex models

Hy-Phy has support for an infinite number of substitution models. Any time reversible model using any finite sequence alphabet can be used. Models for codon and amino acid sequences are available through the Standard Analyses menu selection. For now, we refer users who are interested in writing code for such alphabets to the files in the *Examples* subdirectory.

3.3 Imposing constraints on variables

Simple Constraints: *relrate.bf*

The primary reason for developing ***Hy-Phy*** was to provide a system for performing likelihood analyses on molecular evolutionary data sets. In particular, we wanted to be able to describe and perform likelihood ratio tests (LRTs) easily. In order to perform an LRT it is first necessary to describe a constraint, or series of constraints, among parameters in the probability model. To illustrate the syntax of parameter constraints in ***Hy-Phy***, examine the code in *relrate.bf*:

```
SetDialogPrompt("Select a nucleotide data file:");
DataSet myData = ReadDataFile (PROMPT_FOR_FILE);
DataSetFilter myFilter = CreateFilter (myData,1);
HarvestFrequencies (obsFreqs, myFilter, 1, 1, 1);
F81RateMatrix =
              {{* ,mu,mu,mu}
               {mu,* ,mu,mu}
               {mu,mu,* ,mu}
               {mu,mu,mu,* }};
Model F81 = (F81RateMatrix, obsFreqs);
Tree myTree = (a,b,og);

fprintf(stdout,"\n Unconstrained analysis:\n\n");
LikelihoodFunction theLikFun = (myFilter, myTree, obsFreqs);
Optimize (paramValues, theLikFun);
fprintf  (stdout, theLikFun);

fprintf(stdout,"\n\n\n Constrained analysis:\n\n");
myTree.a.mu := myTree.b.mu;
Optimize (paramValues, theLikFun);
fprintf  (stdout, theLikFun);
```

3.4 Molecular Clocks

Perhaps the most common hypothesis tested using molecular data is that a sequence has evolved according to a molecular clock. It now seems quite clear that a global molecular clock exists for few, if any, gene sequences. In contrast, the existence of local molecular clocks among more closely related species is more probable. ***Hy-Phy*** allows for both types of constraints, including the possibility of testing for multiple local clocks for different user-defined clades in the same tree.

Global clocks: *molclock.bf*

The batch file *molclock.bf* is a simple example of testing for a global molecular clock. The code should be familiar, except for the new `MolecularClock` statement, which declares that the values of the parameter *mu* should follow a molecular clock on the entire tree *myTree*. An important difference in this batch file is that the `Topology` statement defines a rooted tree. Had an unrooted tree been used, it would be treated as a rooted tree with a multifurcation at the root.

```
SetDialogPrompt("Select a nucleotide data file:");
DataSet myData  = ReadDataFile(PROMPT_FOR_FILE);
DataSetFilter myFilter = CreateFilter(myData,1);
HarvestFrequencies(obsFreqs,myFilter,1,1,1);

equalFreqs = {{0.25}{0.25}{0.25}{0.25}};

myTopology = "(((a,b),c),og)"; /* Note: need a rooted tree! */

F81RateMatrix  = {{*,m,m,m}{m,*,m,m}{m,m,*,m}{m,m,m,*}};
Model F81 = (F81RateMatrix, obsFreqs);

fprintf(stdout,"\n\n Unconstrained Analysis: \n");
Tree myTree = myTopology;
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);
unconstrainedLnLik = results[1][0];
numparUn = results[1][1];

fprintf(stdout,"\n\n Molecular Clock Analysis: \n");
MolecularClock(myTree,m);
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
fprintf(stdout,theLikFun);
constrainedLnLik = results[1][0];
numparCon = results[1][1];

lnlikDelta = 2 (unconstrainedLnLik-constrainedLnLik);
```

```
pValue = 1-CChi2 (lnlikDelta, numparUn - numparCon);

fprintf (stdout, "\n\n P-value for Global Molecular Clock Test:", pValue, "\n");
```

The output of this analysis reveals a likelihood value of -589.2528 without a clock, and a value of -593.0986 when a clock is imposed. You also see that code has been added to compute the likelihood ratio test of the clock hypothesis, including the calculation of the P-value (0.021) based on the chi-squared approximation for the likelihood ratio test statistic. These calculations make use of the ***Hy-Phy*** command *CChi2* (see the *Batch Language Command Reference*).

Local clocks: *localclocks.bf*

Particularly when studying data sets consisting of many species spanning a wide level of taxonomic diversity, it may be of interest to assign local molecular clocks to some clades. For instance, in a study of mammalian molecular evolution one might specify that each genus evolves in a clocklike manner, but that different genera evolve at different rates. To allow such analyses, the *MolecularClock* command can be applied to any node on a tree. Unlike the global clock of the previous case, it is not necessary for the *MolecularClock* command to be applied to a rooted tree; the placement of the *MolecularClock* command "roots" the tree, at least locally. To illustrate this feature, we use *localclocks.bf*. Instead of the data file *demo.seq* we use the larger file *largedemo.seq*. The relevant new sections of the code are the tree topology definition:

```
myTopology = "(((a,b)n1,(c,(d,e))n2),f)";
```

and the declaration of two local molecular clocks:

```
fprintf(stdout,"\n\n Local Molecular Clock Analysis: \n");
ClearConstraints(myTree);
MolecularClock(myTree.n1,m);
MolecularClock(myTree.n2,m);
LikelihoodFunction theLikFun = (myFilter, myTree);
Optimize(results,theLikFun);
```

.

The topology string used in *localclocks.bf* takes advantage of ***Hy-Phy***'s extended syntax. Notice how we have named two of the internal nodes *n1* and *n2*. Those names override ***Hy-Phy***'s default (and rather cryptic) node naming convention and allow for us to call functions–in this case, *MolecularClock*– on the clades they tag. The syntax of the *MolecularClock* statements is rather C-like. `MolecularClock(myTree.n1,m);` imposes a local clock on the clade below node *n1* in tree *myTree*. The parameter with clocklike behavior is *m*, the only option for the F81 model being used. By examining the output you notice that the two subtrees do, indeed, have clocklike branch lengths, yet, the tree as a whole is not clocklike. The likelihood ratio test suggests that the local clocks are not present.

Chapter 4

Simulation Tools

The use of simulation in molecular evolutionary analysis has always been important. As computing speed has increased, the role of simulation-based procedures has become even more prominent. Simulation allows us to test statistical properties of methods, to assess the validity of theoretical asymptotic distributions of statistics, and to study the robustness of procedures to underlying model assumptions. More recently, methods invoking simulation have seen increased use. These techniques include numerical resampling methods for estimating variances or for computing confidence intervals, and also the parametric bootstrap procedures for estimating p-values of test statistics. ***Hy-Phy*** provides both parametric and nonparametric simulation tools, and examples of both are illustrated in the following sections.

4.1 The Bootstrap: *bootstrap.bf*

The bootsrap provides, among other things, a simple nonparametric approach for estimating variances of parameter estimates. Consider *bootstrap.bf*. The relevant commands from the batch file are (note that some lines creating output have been deleted for clarity):

```
LikelihoodFunction theLikFun = (myFilter, myTree, obsFreqs);
Optimize (paramValues, theLikFun);
fprintf  (stdout, theLikFun);

reps = 10;
Tree bsTree = (a,b,og);

for (simCounter = 1; simCounter<=reps; simCounter = simCounter+1)
{
        DataSetFilter simFilter = Bootstrap(myFilter,1);
        HarvestFrequencies (simFreqs, simFilter, 1, 1, 1);
        LikelihoodFunction simLik = (simFilter, bsTree, simFreqs);
        Optimize (simParamValues, simLik);
}
```

The first section of code is simply the completion of a typical data analysis, storing and printing results from the analysis of data in *myFilter*. We then create a variable containing the desired number of bootstrap replicates and a copy of the structure *myTree* for use in the bootstrap replicates. (The latter is not necessary, but because the original structure will often be reused after bootstrap replicates are generated, it is a good programming habit to develop.)

The *for* loop is the meat of the batch file. For each of the *reps* replicates, we generate a new *DataSetFilter* named *simFilter*. We do this by creating a bootstrap replicate from the existing *DataSetFilter* named *myFilter*, which was created in the normal fashion. *simFilter* will contain the same number of columns as *myFilter*. Once the new filter has been created, we use it in an appropriate *LikelihoodFunction* command and find MLEs of the parameters. Notice in the complete batch file (not shown in the code above) how we use the matrix variable *BSRes* to tabulate and report the average of all bootstrap replicates. More complex analyses can be programmed with relative ease, or the bootstrap replicates can be saved and imported into a spreadsheet for statistical analyses.

The *Permute* function, with syntax identical to *Bootstrap*, exists for applications where the columns in the existing *DataSetFilter* must appear exactly once in each of the simulated datasets. This feature may be useful for comparisons between the three codon positions or for studies investigating spatial correlations or spatial heterogeneity.

4.2 The Parametric Bootstrap: *parboot.bf*

Another useful simulation tool is the parametric bootstrap. ***Hy-Phy*** provides the *SimulateDataSet* command to provide the type of model-based simulation required. In *parboot.bf* we find the following lines of code. Again, some lines have been deleted for clarity.

```
for (simCounter = 1; simCounter<=reps; simCounter = simCounter+1)
{
        DataSet simData = SimulateDataSet(theLikFun);
        DataSetFilter simFilter = CreateFilter (simData,1);
        HarvestFrequencies (simFreqs, simFilter, 1, 1, 1);
        LikelihoodFunction simLik = (simFilter, bsTree, simFreqs);
        Optimize (simParamValues, simLik);
}
```

The end result is analagous to that of *bootstrap.bf*: we simulate *reps* datasets, find MLEs, and tabulate results. The fundamental difference is that the datasets are formed by simulating using the tree structure, evolutionary model, and parameters in *theLikFun* via the function *SimulateDataSet*. An important technical difference is that *SimulateDataSet* generates a *DataSet* as opposed to the *DataSetFilter* created by *Bootstrap*. Thus, we must use the *CreateFilter* command to create an appropriate filter.

Again note the use of *BSRes* for tabulating results, and also the use of *fscanf* for acquiring input from the user (see the *Batch Language Command Reference* for details).

Chapter 5

Putting It All Together: *positions.bf*

As an example of the type of analysis ***Hy-Phy*** was designed to implement, we now describe the batchfile *positions.bf*. This file illustrates some of the features of the *CreateFilter* command by ignoring species *C* and by creating separate filters for each of the three codon positions. The HKY85 model is used as the basic substitution model. First, the entire data set is analyzed in the normal way, treating all sites in the same way. A second *LikelihoodFunction* is then created, which splits the data according to codon position. Each of the three partitions is allowed to evolve with a separate rate. However, the transition/transversion ratio is constrained to be the same for all three codon positions as well as for all lineages. The likelihood ratio test statistic comparing these two models is computed, and the statistical significance of the test is reported using both the chi-squared approximation and nonparametric bootstrapping.

The file *positions.bf* is rather complicated, so we will focus only on some of its key features.

Using “combs” when filtering data.

It is often the case that molecular data sets have some repeating underlying structure that we would like to exploit or study. For instance, coding regions might be described with the repeating structure 123123123 . . . . In *positions.bf* we create separate *DataSetFilters* for first, second, and third codon positions. The command:

```
DataSetFilter myFilter1 = CreateFilter (myData,1,"<100>","0,1,3");
```

creates a *DataSetFilter* named *MyData1* that includes only the first nucleotide of every triplet. Likewise, the statement

```
DataSetFilter myFilter3 = CreateFilter (myData,1,"<001>","0,1,3");
```

creates a *DataSetFilter* named *MyData3* that includes only the third nucleotide of every triplet. Had we wished to create a filter consisting of both first and second positions, we would have used a statement like

```
DataSetFilter myFilter12 = CreateFilter (myData,1,"<110>","0,1,3");
```

Define a substitution model for each position.

The next portion of *positions.bf* creates a vector of observed frequencies for each of the filters using standard syntax.

Next, the basic substitution model is defined. We use the HKY85 model, with trnasversion parameter *b* and global transition/transversion ratio *R*. A separate *Model* is created for each partition, since they each use different frequencies:

```
global R;
HKY85RateMatrix = {{*,b,R*b,b}{b,*,b,R*b}{R*b,b,*,b}{b,R*b,b,*}};
Model HKY85 = (HKY85RateMatrix, obsFreqs);
Tree myTree = (a,b,og);
Model HKY851 = (HKY85RateMatrix, obsFreqs1);
Tree myTree1 = (a,b,og);
Model HKY852 = (HKY85RateMatrix, obsFreqs2);
Tree myTree2 = (a,b,og);
Model HKY853 = (HKY85RateMatrix, obsFreqs3);
Tree myTree3 = (a,b,og);
```

Define two likelihood functions

We are now ready to set up *LikelihoodFunctions* and *Optimize* them. The analysis of the combined data set is routine:

```
LikelihoodFunction theLikFun = (myFilter,myTree);
Optimize (paramValues, theLikFun);
```

We also store some results for later use:

```
lnLik0 = paramValues[1][0];
npar0 = paramValues[1][1]+3;
fprintf  (stdout, theLikFun, "\n\n");
```

The statement `npar0 = paramValues[1][1]+3;` requires some explanation. The *Optimize* function always returns the number of parameters that were optimized as the `[1][1]` element of its returned matrix of results. Typically, we do not optimize over base frequency values, electing instead to simply use observed frequencies, which are usually very close to the maximum likelihood estimates. Since the frequencies are, in fact, estimated from the data, they need to be included in the parameter count. The value of *npar0*, therefore, includes the count of independent substitution parameters in the model (the number of which is returned by *Optimize*) along with the three independent base frequencies estimated from the data.

The *LikelihoodFunction* for the "partitioned" analysis simply uses the extended form of the *LikelihoodFunction* command:

```
LikelihoodFunction theSplitLikFun = (myFilter1,myTree1,
                                     myFilter2,myTree2,
                                     myFilter3,myTree3);
Optimize (paramValues, theSplitLikFun);
lnLik1 = paramValues[1][0];
npar1 = paramValues[1][1]+9;
```

Note the addition of the 9 estimated frequencies to the model's parameter count.

Finally, we compute the P-value for the test of the combined analysis (null hypothesis) against the split model (alternative hypothesis). Two approaches are used. First, the normal chi-squared approximation to the LRT statistic:

```
LRT = 2*(lnLik1-lnLik0);
pValueChi2 = 1-CChi2 (LRT, npar1-npar0).
```

Estimate the P-value via parametric bootstrapping

One can also estimate the P-value using the parametric bootstrap. The statement for simulating a random dataset based on *theLikFun* is

```
DataSet simData = SimulateDataSet(theLikFun);
```

The remaining part of the loop is basically a copy of the original analysis, with variable names adjusted to indicate that they are coming from simulated data. For each simulated dataset we compute the LRT, named *simLRT*, and compare it to the observed LRT. The estimate of the P-value is the proportion of simulated datasets with a LRT larger than that of the observed data. We simply keep track of the number of such events using the variable *count*:

```
simLRT = 2*(simlnLik1-simlnLik0);
if (simLRT > LRT)
{
        count = count+1;
}
```

and report the results:

```
fprintf(stdout,"\n\n*** P-value (Parametric BS)= ",count/reps,"\n");
```

positions.bf provides a good example of the flexibility of ***Hy-Phy***, and many of the same ideas could be used to develop analyses of multiple genes. Of particular importance for multilocus is the ability to mix local and global variables.

5.1 Exercises

1. Modify *positions.bf* so that the transition/transversion ratio is different from lineage to lineage, but equal among codon positions (ie, all branches leading to species A have the same ts/tv ratio, regardless of the codon position, but that ratio can differ from the ratio for branches leading to species B).

2. Modify *positions.bf* so that (i) all four species in *demo.seq* are analyzed, and (ii) first and second codon positions are combined into a single partition.