Adapting HyPhy to function in a genomics pipeline

Sergei L Kosakovsky Pond

June 19, 2007

1 Overview

This documents presents an approach to script the HyPhy package (`http://www.hyphy.org/`) for automated analysis of multiple sequence alignments, especially in the context of applying the same analysis to a collection of several alignments (e.g. genes from a multiple species genome-wide alignments). All of the components of the analytic pipeline are written in HyPhy batch language (HBL). The pipeline is assembled from the following components

1. **Data readers** (shared by all analyses). These modules split multiple alignment files (e.g. mutliple alignment FASTA files, or individual alignment files accessed via a list file) into a individual alignments, send each to a (user-configurable) analysis module, and pass off the results to (user-configurable) output modules.

2. **Analysis modules** (individual to each of the analyses). Here the actual analysis is performed (one alignment at a time), based on user configured inputs, results are generated and passed on to an output module.

3. **Output writers** (shared by all analyses). These modules convert the output of Analysis modules into a format suitable for export (e.g. comma or tab separated values).

4. **Configurations** (individual to each of the analyses). Each configuration file specifies which of the above three components to use, and also sets configurable options for each analysis.

5. **Utility files** (shared by all analyses). Small HBL files that provide useful functions for use by any of the modules/analyses.

Each kind of file resides in its own top level directory in the genome analyses directory and uses relative paths to link to each other; hence the directories must not be moved or renamed. However, the entire genome analysis directory can be moved wherever is convenient.

A typical invocation of an analysis from the command line may look like this:

```
$/path/to/HYPHY /path/to/genome/analyses/Configurations/myConfig.bf
```

Properly written modules will not require the standard input, hence the entire analysis can be queued or backgrounded, e.g.

```
$nohup /path/to/HYPHY /path/to/genome/analyses/Configurations/myConfig.bf \
> /dev/null &
```

2 Data readers

The function of the data readers is to include all other analysis components (prompted for via filenames and supplied by the configuration file), execute pre-, normal and post-processors defined in the output and analysis modules. They reside inside the *DataReaders* directory. All data readers MUST perform the following steps (see `DataReaders/FASTAReader.bf` for a well-commented example).

- Read the paths of analysis and output modules from the standard input (those will be provided by the configuration file) using `fscanf`. Execute files at those paths using `ExecuteAFile()`. Paths in HyPhy can be defined either via an absolute specification, or (preferably) relative to the location of the file using these paths. For relative paths, UNIX path syntax is recommended (it is automatically converted to the appropriate path specification on other platforms).
- Read the path for the output file using `fscanf`, storing it in the `_outPath` variable. Read optional inputs specific to the reader (e.g. a tree string)
- Execute `_prepareFileOutput (_outPath);` prior to executing the analysis.
- Read input alignments one at a time, decide whether a given alignment is valid or not. If the alignment is valid, create a `DataSet` object named `ds` containing the alignment. For each alignment (valid or not), call `_processAGene(valid, ID, errorTag)` using the first booleanargument to specify if the alignment is valid or not, the second to provide the integer index of the alignment and the third - to specify how to annotate invalid genes in the output.
- Execute `_finishFileOutput (0);` following the completion of all analyses.

3 Analysis modules

The function of the analysis modules is to perform the statistical analysis on each alignment. They reside inside the *AnalysisModules* directory. All data analysis modules MUST perform the following steps (see `AnalysisModules/SimpleGlobalFitter.bf` for a well-commented example).

- Define the `returnResultHeaders` function with one dummy argument (since HyPhy currently does not allow argument-free functions). The function must return a $0 \ldots N-1$ indexed associative array (N is the number of results returned per analysis), defining the ordering and names of every output option for the analysis. For example, if the analysis returns the index of the gene, the log likelihood score of a model fit, the tree string with branch lengths, and the estimate of transition/transversion ratio, the function may look like this:

```
function returnResultHeaders (dummy)
{
        _analysisHeaders = {};
        _analysisHeaders[0]  = "GENE";
        _analysisHeaders[1]  = "LogL";
        _analysisHeaders[2]  = "Tree";
        _analysisHeaders[3]  = "Transition/Transversion Ratio";
        return _analysisHeaders;
}
```

  This array will be used by output modules to decide which return values should be output, how to label, and how to order them. This arrangement assumes that the results returned from every gene will have the same number of entries.

- Define the `runAGeneFit(ID)` function with one argument - the index of the current gene being processed. The function performs the actual data analysis and must make use of the data stored in `DataSet ds` and return an associative array with results indexed by the keys defined in `returnResultHeaders`. For the same example as above the function may look like this:

```
_fileCount = 0;
function runAGeneFit (geneID)
{
```

```
        if (_fileCount == 0)
        {
                ... include some auxiliary files ...
                ... do initalizations ...
        }

        ....
        run the analysis
        ...

        _analysisResults = {};
        _analysisResults[``GENE''] = geneID;
        _analysisResults[``LogL''] = logLikScore;
        _analysisResults[``Tree''] = Format(treeVariable,0,1);
        _analysisResults[``Transition/Transversion Ratio''] = kappa;

        _fileCount = _fileCount + 1;

        return _analysisResults;
}
```

Note that the function may also need to perform some initializations (before the first file is analyzed) and include some auxiliary files.

4 Output modules

The function of the output modules is to format and write the output from each alignment analysis. They reside inside the *Writers* directory. All data writers MUST implement the following three functions (they may assume that the data reader and the analysis modules have been read in and all the functions defined therein are available). See `Writers/TAB.bf` for an example.

- `_prepareFileOutput(outPath)` - the prefix function which takes the path of the output file as input, and does all the work necessary to start outputting results, e.g. opens the file, clears it, writes headers etc.
- `_processAGene(valid,ID,errorTag)` - this function is responsible for calling the analysis function `runAGeneFit` and writing out the results of each analysis;

the arguments are: is the alignment valid (if yes, run the analysis, if not - write out `errorTag`), the ID of the gene, and finally the error message to write for invalid alignments.

- `_finishFileOutput(dummy)` - the postifx function which takes no arguments and finalizes the writing of results, e.g. closing the files.

5 Configuration files

These reside in `Configurations` directory, and configure each individual analysis. The job of a configuration file is twofold:

- Select (via paths) the data reader, analysis module, and writer. The latter two are supplied as overloaded stdin arguments to the data reader, and the data reader itself is called by using `ExecuteAFile (/path/to/reader, stdinOverload`
- Populate the stdin overload associative array with all other options required by the selected data reader and analysis modules (look at the comments at the beginning of each .bf file to compile the necessary list of inputs).
- To define overloaded standard input, one populates an associative array (all values are strings) with the input options in the order they are requested. The keys for each option must also be strings, lexicographically ordered (e.g. "00" for the first option, "01" - for the second, etc).