A Quick Guide for the cubfits Package (Ver. 0.1-2)

Wei-Chen Chen[1], Russell Zaretzki[2,4], William Howell[1], Cedric Landerer[1], Drew Schmidt[3], and Michael A. Gilchrist[1,4]

[1]Department of Ecology and Evolutionary Biology,
University of Tennessee,
Knoxville, TN, USA

[2]Department of Statistics, Operations, and Management Science,
University of Tennessee,
Knoxville, TN, USA

[3]National Institute for Computational Sciences,
University of Tennessee,
Knoxville, TN, USA

[4]National Institute for Mathematical and Biological Synthesis,
Knoxville, TN, USA

Contents

This publication was typeset using LaTeX.

Acknowledgement

Support for this project was provided by the Department of Ecology and Evolutionary Biology at the University of Tennessee, Knoxville; the National Institute for Mathematical and Biological Synthesis; and the Tennessee Science Alliance, and a grant from the National Science Foundation (MCB-1120370).

Warning: The findings and conclusions in this article have not been formally disseminated and should not be construed to represent any determination or policy of University, Agency, and National Laboratory.

This document is written to explain the main functions of **cubfits** (Chen *et al.* 2014b), version 0.1-1. Every effort will be made to ensure future versions are consistent with these instructions, but features in later versions may not be explained in this document.

1. Introduction

Coding sequences or open reading frames (ORFs) are regions of genome encoding proteins required to properly function any living biological organism. In order to adopt to environment, growth in limited resource, or maintain population size, the protein production needs to be reflect quickly under those condition. The efficiency of protein translation is essentially critical for some tiny organism, such as yeast. For example, we may model the translation rates, from a codon sequence to a protein, with several factors and in several ways. Once the codon information encoded behind a genome is estimated/approximated appropriately, then the model can be used to predict gene expression levels or protein production rates. Therefore, it is interesting to know what factors may affect production rates, how strong they are, or whether genomes evolve accordingly.

In a simplified example, Cysteine (Cys/C) is one of 20 amino acids and can be encoded by two synonymous codons, TGT and TGC. Suppose Cysteine are observed in two coding sequences and used to encode two proteins. Suppose further one has lower expression level, but the other one has higher expression level. Assume TGT and TGC both have no effect to expression level, then the chance of observing TGT and TGC is roughly 50% and 50% among both coding sequences. Suppose the chance of observing TGC is much higher than observing TGT in the protein has higher expression level, then we may guess that TGT is more efficient than TGC for that protein translation.

cubfits (Chen *et al.* 2014b) models the biased patterns of codon usage among other information and fits model parameters. The implemented models are introduced in (Gilchrist 2007; Shah and Gilchrist 2011; Wallace *et al.* 2013) that combines several modeling techniques of Population Genetics and Statistics to predict protein production rates.

Package requirements and installation of **cubfits** are described next. Section 2 gives short examples for main functions. Section 3 provides parallel implementations that simply speedup computations, and discusses technical issues. In Section 4, useful work flows built in **cubfits** are introduced. Section 5 introduces implemented methods of data input/output and conversion between different data structures with examples. Finally, Section 7 introduces some useful functions.

1.1. Requirements

cubfits is a package of R (R Core Team 2013) built and test on R 3.0.0, but may be compatible back to 2.15.0 or early. It requires **methods** package which is default in R, and suggests **seqinr** (Charif and Lobry 2007) and **VGAM** (Yee 2013) packages that both are available and can be installed from CRAN or it's mirror sites. The **seqinr** takes care input and output of DNA sequence data in FASTA format, and the **VGAM** takes care several core functions of model fitting used by **cubfits**. The **cubfits** also uses several high performance computing tech-

niques to speed up computation including **parallel** (R Core Team 2012) and **pbdMPI** (Chen *et al.* 2012b). Further, the **cubfits** provides some diagnoses for multiple chains of MCMC results and restart MCMC chains utilizing packages such as **doSNOW** (Revolution Analytics and Weston 2014) and **coda** (Plummer *et al.* 2006), and their dependent packages. Other utilities in **cubfits** also include **EMCluster** (Chen *et al.* 2012a) for a mixture distribution in one dimension.

The **cubfits** uses a core function vglm() of **VGAM** for model fitting, and (optionally) parallelizes via mclapply() of **parallel** package and task.pull() or pbdLapply() of **pbdMPI** package. The parallelization is designed across the 20 amino acids. Note that mclapply() is only supported on Unix-like systems and on shared memory machines, while tak.pull() and pbdLapply() are supported on most systems (Linux/Unix/MacOS/Windows) and on both of shared memory machines and distributed clusters provided a MPI (Gropp *et al.* 1994) library is installed. See Chen *et al.* (2014a) for more details of installation of a MPI library and **pbdMPI**.

1.2. Installation and Test

The **cubfits** can be either installed within an R session as

R Script

```
> install.packages("seqinr")
> install.packages("VGAM")
> install.packages("doSNOW")
> install.packages("coda")
> install.packages("EMCluster")
> install.packages("cubfits")
```

or installed from a command line as

Shell Command

```
$ R CMD INSTALL seqinr_3.7-0.tar.gz
$ R CMD INSTALL VGAM_0.3-9.tar.gz
$ R CMD INSTALL doSNOW_1.0.12.tar.gz
$ R CMD INSTALL coda_0.16-1.tar.gz
$ R CMD INSTALL EMCluster_0.2-4.tar.gz
$ R CMD INSTALL cubfits_0.1-0.tar.gz
```

provided other requirements or dependent packages are installed correctly. Note that **parallel** and **pbdMPI** are optional.

A simple test can be used to see if the **cubfits** installed correctly as

R Script

```
> demo(plotbin, 'cubfits')
```

within an R session, and this demo provides a plot as in Figure 1.

2. Main Functions

The **cubfits** package is composed of three main functions:

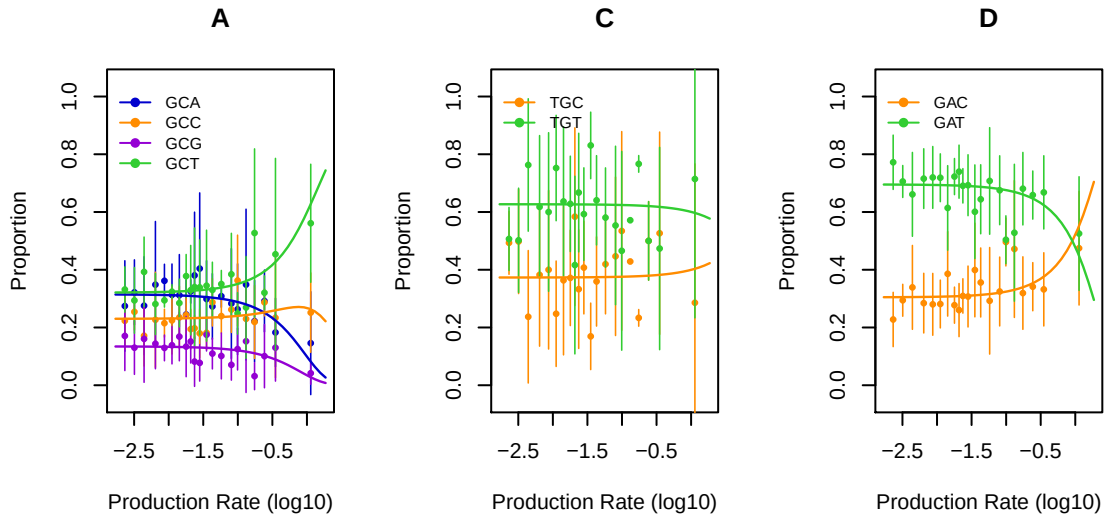


Figure 1: A simple demo plot. Figures show the empirical binning results by expression levels where dots are mean proportions of synonymous codons of 100 sequences in every 5% expression windows, and horizontal lines are 90% empirical intervals. The curves are theoretical prediction of synonymous codon usages (Shah and Gilchrist 2011). See Section 7 for details. For amino acid A, the selection effect dominates the mutation effect for some codons at expression level larger than −1.5. For C, it is mainly mutation effect. For D, the selection effect can be observed at expression level larger than −0.75.

1. `cubfits()` fits models for sequences with observed ϕ (expression level) values and measurement errors,

2. `cubappr()` approximates models for sequences without any observed ϕ values, and

3. `cubpred()` fits models for sequences with observed ϕ values and measurement errors, and then predicts ϕ values for sequences without observations.

See package's help pages for details of other input options.[1]

2.1. Demonstrations

The **cubfits** package provides quick examples for its three main functions:

R Script

```
> demo(roc.train, 'cubfits')     # for cubfits()
> demo(roc.appr, 'cubfits')      # for cubappr()
> demo(roc.pred, 'cubfits')      # for cubpred()
```

These **cubfits** demos perform short MCMC runs and analyze toy datasets (`ex.train` and `ex.test`) for the Ribosome Overhead Cost (ROC) (Shah and Gilchrist 2011) which is shown

[1] `?cubfits::cubfits`, `?cubfits::cubappr`, and `?cubfits::cubpred`.

in Figure 1. The toy datasets have only 100 short sequences, and only 3 amino acids are considered.

For a standard data analysis, the process essentially consists of:

1. reading sequences and expressions files,
2. converting to appropriate data structures,
3. running a main function (MCMC),
4. summarizing MCMC outputs, and
5. plotting predictions.

The **cubfits** package also provides an example using simulated data:

R Script

```
> demo(simu.roc, 'cubfits')      # cubfits() is called.
```

Note that this demo will generate a fake sequence file (`toy_roc.fasta`) in FASTA format in the working directory, and (for testing) read it back. Also, it converts the data into the correct format needed by `cubfits()`, and finally runs MCMC and generates a plot.

2.2. Modeling ROC

cubfits (Chen *et al.* 2014b) utilizes a Bayesian hierarchical model for codon useage bias (CUB) (Gilchrist 2007; Shah and Gilchrist 2011; Wallace *et al.* 2013; Gilchrist *et al.* 2014). The CUB model for ROC is

$$\begin{aligned}
\vec{y}_{ga} &\sim Multinom(n_{ga}, \vec{p}_a), && (1)\\
\vec{p}_a &= mlogit^{-1}(\log(\vec{\mu}_a) + \vec{\Delta}_{ta}\phi_g), && (2)\\
\log(\phi_{g,obs}) &\sim N(\log(\phi_g) + \log(K_{bias}), \sigma^2_{obs}), && (3)\\
\phi_g &\sim \log N(-\sigma^2_\phi/2, \sigma_\phi), && (4)\\
\vec{\mu}_a, \vec{\Delta}_{ta}, \sigma_\phi, K_{bias} &\propto 1, \text{ and} && (5)\\
f(\sigma^2_{obs}) &\propto 1/\sigma^2_{obs}. && (6)
\end{aligned}$$

Given gene g and amino acid a, $\vec{y}_{ga}$ is synonymous codon counts following a multinomial distribution with parameters n_{ga} for total codons and $\vec{p}_a$ for probability of synonymous codons. The $mlogit^{-1}$ is an inversed multivariate logit function of $\vec{\mu}_a$ mutation rate and $\vec{\Delta}_{ta}$ elongation time. ϕ_g is protein production rate and $\phi_{g,obs}$ is observed expression. K_{bias} and σ^2_{obs} are measurement bias and error, respectively. σ^2_ϕ is dispersion of $\log(\phi_g)$. We assume independence of amino acids and independence of genes, and both contribute information to the CUB model.

The package implements MCMC algorithms of CUB model in three main functions. `cubfits()` estimate model parameters for sequence with $\phi_{g,obs}$ and implements Equations (1) to (4). `cubappr()` predicts expression for sequences without $\phi_{g,obs}$ that skips Equation (3). `cubpred()` estimates parameters for sequences with $\phi_{g,obs}$ and predicts expression for sequences without

$\phi_{g,obs}$. All functions can take options set by global objects such as MCMC conditions and auxiliary control variables, especially the prior and hyper-parameters as Equations (5) and (6) Figure 2 displays dependence of the model. The extended Bayesian hierarchical model with measurement errors for predicting protein synthesis rates "with" or "without" gene expression (ϕ) is abstracted in the dependence graph:

- data hierarchy is in blue boxes,
- observed sequences are in a green solid circle,
- gene expressions (may be ignored) are in a green dashed circle,
- unknown parameters to be estimated are in red dashed circles,
- prior parameters (may be restricted) are in the black circle, and
- parallelization within MCMC is in red dashed lines.

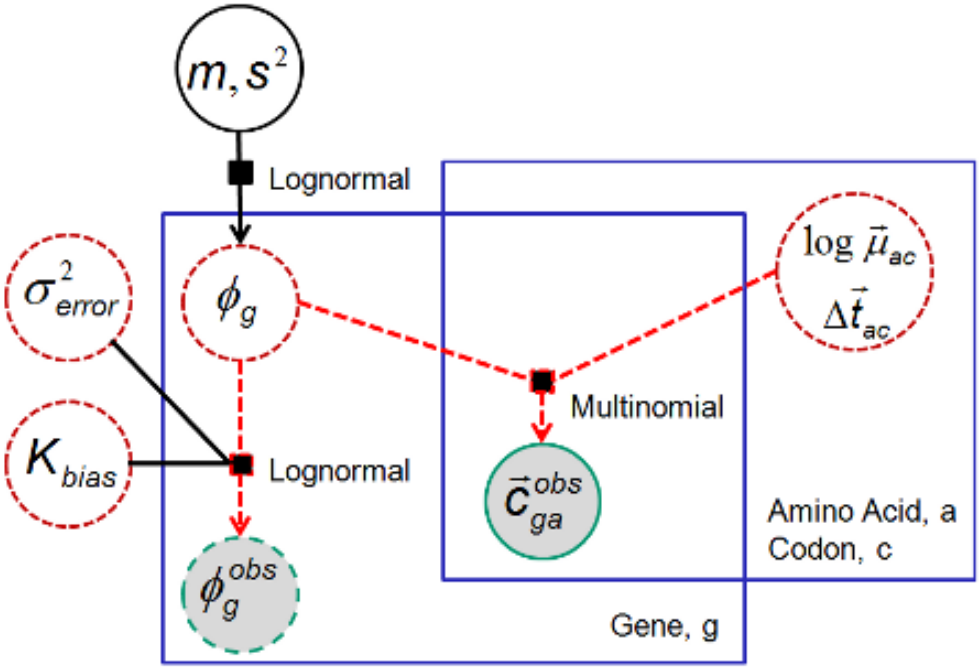


Figure 2: Dependence plot of ROC model (after (Wallace *et al.* 2013)).

2.3. Generic Functions

Note that the three main functions are wrappers of other generic functions that perform parameter initializations, propose new parameters, compute MCMC acceptance/rejection ratio, and more. The function `init.function()` initializes generic functions that will be called by the three main functions. Although `init.function()` is called within each of three main functions to setup the generic functions, it also needs to be called before using other utility functions, such as `fitMultinom()`, see Section 7 for examples.

Accompanying control variables such as `.CF.CT` and `.CF.OP`, the `init.function()` will dispatch the corresponding generic functions into a default environment `.cubfitsEnv`, allowing other main functions may call those generic functions dynamically.

Note that generic functions in R typically only depend on input object types. However, the design in **cubfits** has several advantages:

- functions are clearer by making good use of data structures and simplifying options,
- extensions are easier to create; simply add more generic functions rather than changing main functions, and
- performance is more efficient by avoiding extra conditional checks such as `if(...){...} else{...}` in every iteration.

Also, the design can avoid tedious CRAN checks since there are some restrictions in accessing `.GlobalEnv`. For example, `.cubfitsEnv$my.fitMultinomAll()` is called in several internal functions to fit multinomial logistic regression in every MCMC iterations. In particular, it has four generic functions:

1. `my.fitMultinomAll.lapply()` ueses `lapply()` in the serial version,
2. `my.fitMultinomAll.mclapply()` uses `parallel::mclapply()` in shared memory machines,
3. `my.fitMultinomAll.task.pull()` uses `pbdMPI::task.pull()` in distributed clusters, and
4. `my.fitMultinomAll.pbdLapply()` uses `pbdMPI::pbdLapply()` in distributed clusters, but is only efficient for homogeneous tasks.

Through `init.function()`, there is no need to check which generic function should be called within the MCMC step, and there is no need to worry about serial or parallel details when designing a MCMC algorithm.

3. Parallel Speedup

The main bottleneck of **cubfits** is in calls to `fitMultinom*()` or `my.fitMultinom*()`. The reasons are that

- models assume conditional independence of amino acids, and
- it is tedious to extract parallelism within the `VGAM::vglm()` function.[2]

Fortunately, by carefully utilizing summarized data structures, one can avoid some burdens and boost the MCMC computation significantly. For example, sorting by ORF's names before

[2] As long as the number of sequences is not too large and summary statistics are used, there is no need to consider this approach. Although this can lead to huge improvement in speed, it may require the need to rewrite all of `vglm()` and reorganize data structures, which is highly non-trivial.

MCMC may avoid subsetting a `data.frame` in some cases, and further computation can be easily moved to C.

Further, **cubfits** offers three ways of parallelizing computations to speedup computations, including `parallel::mclapply()`, `pbdMPI::task.pull()`, and `pbdMPI::pbdLapply()`. Note that only **pbdMPI** functions have received the most thorough testing in **cubfits**. We now briefly describe these three approaches.

3.1. `mclapply()`

The function `mclapply()` is a function in the **parallel** package and is particularly useful for multi-core/shared memory machines; however, as it relies on the system's `fork` utility, it is only available for Unix-like systems such as Linux and Mac OS. On Windows systems, this function is identical to `lapply()`. As the name implies, `mclapply()` is a parallel version of R's own `lapply()`, and as such it is usually trivial to migrate to `mclapply()` from code using `lapply()`.

With this approach in **cubfits**, we split the work into jobs (tasks) at the level of 20 amino acids, because of the conditional independence assumption. However, the computation is not evenly distributed across amino acids, as some have more synonymous codons than others; e.g. Leucine (Leu/L) has 6 synonymous codons. In other words, the 20 jobs are not homogeneous (and as an aside, Leucine generally needs the most computing in those `fitMultinom*()` calls). Therefore, we use the `mclapply()` option `mc.preschedule = FALSE`, and limit the number of cores to 4 or 5. In doing so, we are able to somewhat overcome this computational imbalance.

The warning section of the `mclapply()` help page notes that it is _*strongly discouraged*_ to use these functions in GUI or embedded environments. For small tests, i.e. those with fewer sequences and shorter MCMC iterations, the `mclapply()` approach works well and with reasonable efficiency in **cubfits**. However, we did observe some occasional crashes when using this approach for longer runs of MCMC, such as in a work flow.

3.2. `task.pull()`

The **pbdMPI** package is designed in single program multiple data (SPMD) framework, and is a simpler approach to parallelize codes than the perhaps more familiar manager/worker paradigm. Although only batch computing (i.e., no interactivity) is allowed in **pbdMPI**, one of the benefits is that it is not strictly limited to shared memory machines, and can be applied on most systems with an MPI library installed. See Chen *et al.* (2014a) for details on installing an MPI library and **pbdMPI**.

The task pull is a simple parallelism strategy and is very similar to `mc.preschedule = FALSE` in `mclapply()`; in particular, this approach gains performance economically for non-homogeneous jobs. `task.pull()` in **pbdMPI** is a convenient implementation that can boost the performance of `fitMultinom*()` functions, and has similar syntax as `mclapply()`.

3.3. `pbdLapply()`

For some cases, task pull is neither economical nor efficient. For example, in **cubfits**, one needs to initialize the expected expression level of each sequence via a call to `estimatePhiAll*()`. Generally, this performs the same computation on all sequences. Therefore, a task pull on

5,000 sequences may not be a good idea, since the extra 5,000 calls for requesting unfinished sequences are unavoidable in this approach.

The function `pbdLapply()` is similar to set `mc.preschedule = TRUE` in `mclapply()`. The 5,000 sequences are equally divided by the number of cores available for processing and processed by each core simultaneously. As such, there is no call for requesting sequences. A good choice of the number of cores to use is generally related to the number of sequences. However, this needs to be tuned with a few empirical runs for optimal performance.

3.4. Performance

The three methods of parallelizing **cubfits** calls, (`mclapply()`, `task.pull()`, and `pbdLapply()`), have all been tested on a Yeast genome which has about 5346 genes and corresponding gene expression. The average run times of 200 iterations of `cubfits()` are recorded in different numbers of cores. The speedup via parallelization is defined as the run time of one core of `mclapply()` divided by the run time of the parallel version.

Warning: the next test was done with `vglm()` fits for each MCMC iteration.

Figure 3 shows that for this test, `task.pull()` is the best approach, although it needs an additional core for task management. There is no obvious best choice for number of cores, but after about seven cores, `task.pull()` is beginning to level out. The performance degradation of `pbdLapply()` at 8 cores is due to the unbalanced pre-assignment of amino acids; e.g. Leu and Arg, both computationally expensive amino acids, may be assigned to the same core. The performance of `mclapply()` is surprisingly poor by comparison, which may be due to memory copying between th eforking mechanism and `mc.preschedule = FALSE`.

4. Work Flows

Warning: All work flows are intended to be run in batch mode and on Unix-like systems for optimum performance.

The **cubfits** package is built with some useful work flows, including

1. `simu` for simulation studies,
2. `wphi` for sequences with expression levels (ϕ) and measurement errors,
3. `wophi` for sequences without expression levels, and
4. `wphi_wophi` for model validation.

We share these work flows as templates which may be useful for further research. A quick way to obtain a default work flow is by calling `cubfits::cp.workflow()`, which contains the three aforementioned work flows. Note however that these are only tested privately under a Linux environment, and are merely served as templates. None of them are checked by CRAN.

Figure 4 depicts the analysis process and how **cubfits** functions play behind work flows.

- input: FASTA file and gene expression for "with" ϕ work flow (`wphi`),
- execute: major functions `cubfits()` and `cubappr()` in parallel,

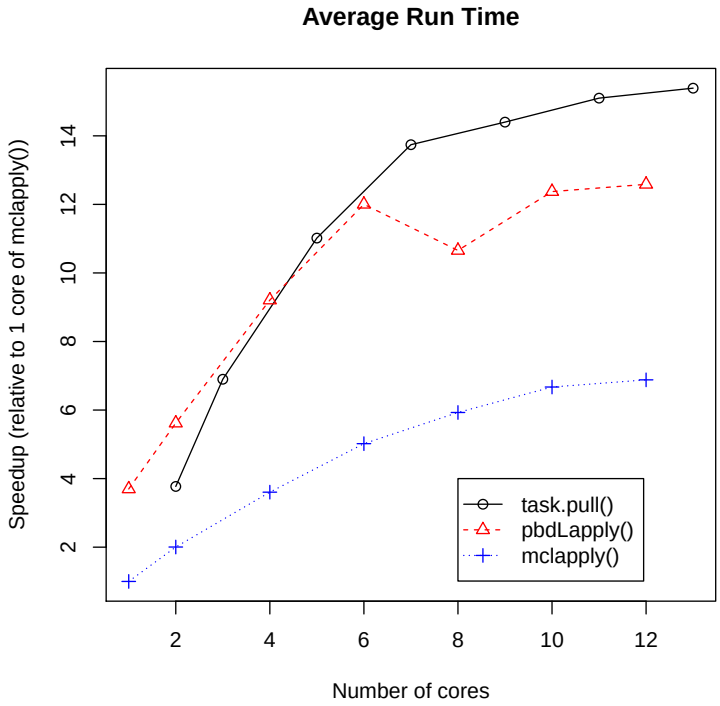


Figure 3: Speedup of three different parallelizations.

- output: save `.rda` files for post processing,
- post process: subset MCMC iterations and precompute statistics,
- diagnosis: generate plots,
- summary: obtain parameters for "with" ϕ work flow, and
- feedback: utilize "with" ϕ results to initial "without" ϕ work flow (`wophi`).

The work flows are stored in the package directory `cubfits/inst/workflow/` and installed in `${R_HOME}/library/cubfits/workflow/`. Therefore, executing the following script can regenerate those work flows in three directories `./01-simu/`, `./02-wphi/`, `03-wophi` and `./04-wphi_wophi/`.

Shell Command

```
$ mkdir 01-simu
$ cd 01-simu/
$ Rscript -e "cubfits::cp.workflow('simu')"
$ cd ../
$ mkdir 02-wphi
$ cd 02-wphi/
$ Rscript -e "cubfits::cp.workflow('wphi')"
$ cd ../
$ mkdir 03-wophi
```

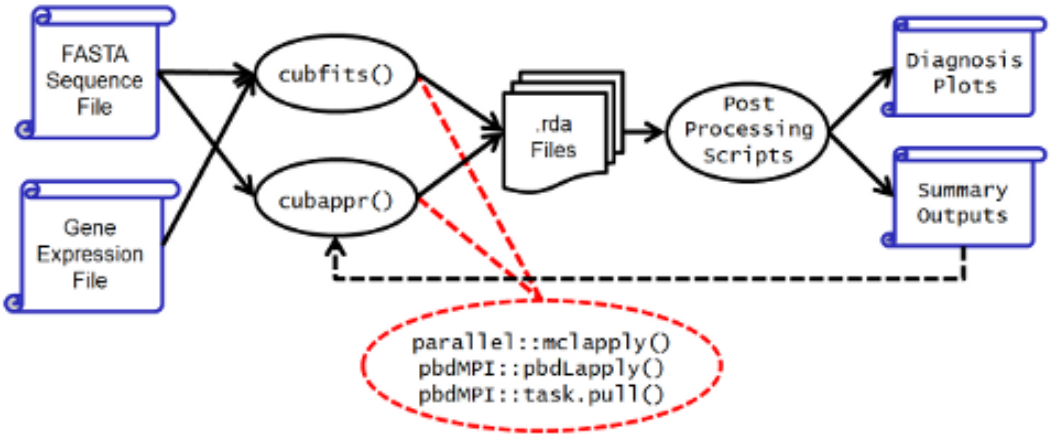


Figure 4: `wphi` work flow is mainly based on `cubfits()` and it's outputs can be initial values for `wophi` work flow mainly based on `cubappr()`. Note that `wphi` needs both sequence and expression data, while `wophi` only needs sequence. The dashed black line indicate `wphi_wophi` work flow combining both `wphi` and `wophi`.

```
$ cd 03-wophi/
$ Rscript -e "cubfits::cp.workflow('wophi')"
$ cd ../
$ cd 04-wphi_wophi/
$ Rscript -e "cubfits::cp.workflow('wphi_wophi')"
$ cd ../
```

Note that a work flow is mainly coded in shell script and sequentially executes analyses via several shell commands and R scripts. The analysis R scripts are managed and stored in `cubfits/inst/workflow/` and installed in `${R_HOME}/library/cubfits/workflow/`. Those are served as templates and can be altered by users (copy scripts to local directory and modify as needed.)

After regenerating those work flows, three shell scripts and one R script are created within each directory. These scripts consist of:

- `run_0.sh` creates the needed subdirectories,
- `run_1.sh` is the main script of analysis,[3]
- `run_2.sh` is for post processing,
- `run_2_ps.sh` is for post scaling, and
- `00-set_env.r` is the configuration R script for **cubfits**.

For `simu` work flow, one can sequentially execute those shell scripts (`run_*.sh`) without further changes, and may learn to adjust scripts for further studies. For `wphi` and `wophi`

[3] This may require up to 30 cores and may take a few hours to finish depending on the number of sequences. Please do some testing and adjust this file and `00-set_env.r` accordingly before a hero run.

work flows, one needs to provide genome sequences (e.g. `./02-wphi/param/genome.fasta`) and expression data (e.g. `./02-wphi/param/genome.phi.tsv`).

Examples of post processing from the `simu` work flow (as in (`run_2.sh`) are given in Figure 5. The plots are from files `./01-simu/all.out/plot/prxy_roc_ad_fits_pm_5k-10k.pdf` and `./01-simu/all.out/plot/prxy_roc_ad_appr_pm_5k-10k.pdf`.

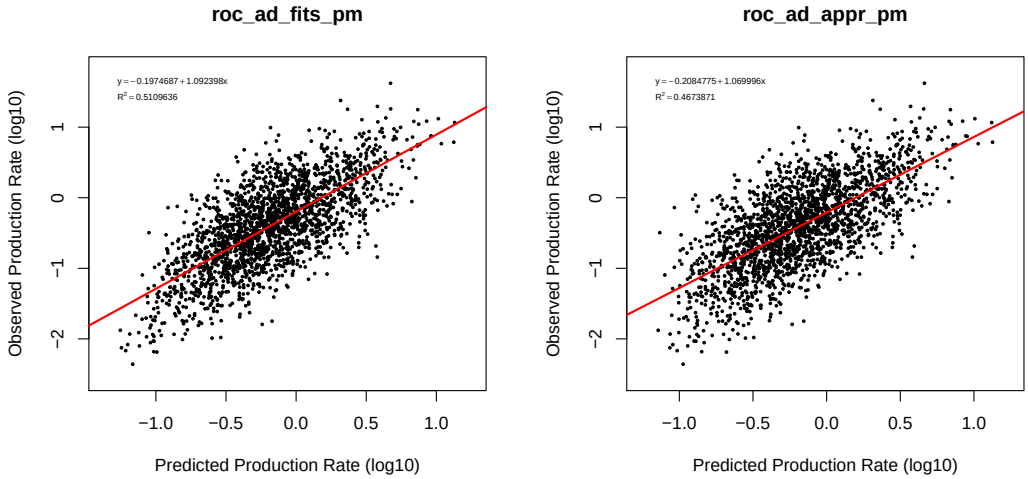


Figure 5: The left plot is predicted expression (MCMC posterior mean of expected expression) from model fits against observed expression (with measurement errors). In the right plot, the predicted expression is approximated from a simulation (no model fits.)

5. Utilities

The **cubfits** package provides simplified data input and output utilities that can quickly import sequence and expression values, turn them into R data structures, and dump analysis results in a standard format. As long as data are in this common format, the following functions simplify analysis work flows as in Section 4. Further, the R data structures used in **cubfits** can be converted to different formats after reading in from disk and/or before writing out to disk.

5.1. Data I/O Functions

- `read.seq()` reads sequence data in FASTA format, and
- `write.seq()` writes sequence data in FASTA format.

Both functions are simplified wrapper functions of package **seqinr**. The writing function is mainly intended for simulation studies, as described in the section above.

- `read.phi()` reads in expression values in tsv/csv format, and

- `write.phi()` writes out expression values in tsv/csv format.

The default data structure for these is an R `data.frame`. The writing function is mainly for simulation studies.

Warning: ideally, the input objects ϕ and ORF sequences should be

1. matched with each other correspondingly by name, and
2. sorted by name.

If not, `%in%`, subsetting, and `sort()` functions can post process input data before converting to main data structures. We next introduce main data structures followed by conversion functions provided in **cubfits**.

5.2. Main Data Structures

The main data structures used in **cubfits** are:

- `reu13.df` (used by REU13 students) contains codon positions and expression levels,
- `y` (used by REU12 students and Wallace *et al.* (2013)) contains synonymous codon counts for each sequence,
- `n` (used by REU12 students and Wallace *et al.* (2013)) contains total codon count for each sequence,
- `reu13.list` is a list version of `reu13.df` (adopted by WCC),
- `y.list` is a list version of `y` (adopted by WCC), and
- `n.list` is a list version of `n` (adopted by WCC).

These are typically fixed after data input steps and mainly used in the three main functions without further changes. Note that the objects with these data structures are normally sorted by ORF id's or names. The list versions are also useful for some models and speedup model fitting. Some conversion functions between data structures are provided in **cubfits**, see Section 5.3 for details.

Also, there are other data structures for parameters, but those are sometimes model dependent. For example, data structure `b` may contain model parameters in different dimension. The parameters of each amino acid are $(log(\mu), \Delta t)$ for ROC model, $(log(\mu), \omega)$ for NSEf model, and $(log(\mu), \Delta t, \omega)$ for ROC+NSEf model. See help pages (`?cubfits::AllDataFormats`) for details.

The following is an example of `reu13.df` data structure containing three amino acids A, C, and D. Each amino acid is in `data.frame` with four required columns `ORF` (sequence id), `phi` (expression level), `Pos` (codon position), and `Codon`, and one optional column `Codon.id` (created by `rearrange.reu13.df()` within `gen.reu13.df()` call.)

Example of `reu13.df`

```
> str(ex.train$reu13.df)
```

```
List of 3
 $ A:'data.frame':      2682 obs. of  5 variables:
  ..$ ORF     : chr [1:2682] "YBL023C" "YBL023C" "YBL023C" "YBL023C"
     ...
  ..$ phi     : num [1:2682] 0.0186 0.0186 0.0186 0.0186 0.0186 ...
  ..$ Pos     : num [1:2682] 109 353 123 294 1133 ...
  ..$ Codon   : chr [1:2682] "GCA" "GCA" "GCA" "GCA" ...
  ..$ Codon.id: int [1:2682] 0 0 0 0 0 0 0 0 0 0 ...
 $ C:'data.frame':      662 obs. of  5 variables:
  ..$ ORF     : chr [1:662] "YBL023C" "YBL023C" "YBL023C" "YBL023C"
     ...
  ..$ phi     : num [1:662] 0.0186 0.0186 0.0186 0.0186 0.0186 ...
  ..$ Pos     : num [1:662] 387 862 813 248 226 40 82 477 922 87 ...
  ..$ Codon   : chr [1:662] "TGC" "TGC" "TGC" "TGT" ...
  ..$ Codon.id: int [1:662] 0 0 0 1 1 1 1 1 1 1 ...
 $ D:'data.frame':      3164 obs. of  5 variables:
  ..$ ORF     : chr [1:3164] "YBL023C" "YBL023C" "YBL023C" "YBL023C"
     ...
  ..$ phi     : num [1:3164] 0.0186 0.0186 0.0186 0.0186 0.0186 ...
  ..$ Pos     : num [1:3164] 209 199 255 89 273 141 263 158 112 306
     ...
  ..$ Codon   : chr [1:3164] "GAC" "GAC" "GAC" "GAC" ...
  ..$ Codon.id: int [1:3164] 0 0 0 0 0 0 0 0 0 0 ...
```

The following is an example of corresponding `y` which has only synonymous codon counts for each amino acids and sequences.

Example of `y`

```
> str(ex.train$y)
List of 3
 $ A: int [1:100, 1:4] 16 4 5 17 5 7 4 0 20 13 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:100] "YBL023C" "YBL074C" "YBR060C" "YBR068C" ...
  .. ..$ : chr [1:4] "GCA" "GCC" "GCG" "GCT"
 $ C: int [1:100, 1:2] 3 0 1 5 4 2 4 2 0 11 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:100] "YBL023C" "YBL074C" "YBR060C" "YBR068C" ...
  .. ..$ : chr [1:2] "TGC" "TGT"
 $ D: int [1:100, 1:2] 15 10 10 11 6 12 8 2 17 27 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:100] "YBL023C" "YBL074C" "YBR060C" "YBR068C" ...
  .. ..$ : chr [1:2] "GAC" "GAT"
```

The following is an example of corresponding `n` which has only total codon counts for each amino acids and sequences.

Example of `n`

```
> str(ex.test$n)
List of 3
 $ A: Named int [1:100] 60 22 19 16 22 28 9 22 35 8 ...
  ..- attr(*, "names")= chr [1:100] "YAL017W" "YBL033C" "YBL102W"
     "YBR010W" ...
```

```
$ C: Named int [1:100] 12 7 5 0 1 7 5 4 4 4 ...
 ..- attr(*, "names")= chr [1:100] "YAL017W" "YBL033C" "YBL102W"
    "YBR010W" ...
$ D: Named int [1:100] 96 26 6 4 24 21 5 28 28 3 ...
 ..- attr(*, "names")= chr [1:100] "YAL017W" "YBL033C" "YBL102W"
    "YBR010W" ...
```

5.3. Conversion of Data Structures

In order to improve performance, simply parallelizing the code is not enough. In R, a good data structure can have massive impact on performance. In practice, it is best that not to change/modify/subset existing data structures within computations/iterations. It can be worth generating extra data structures which are efficient for some specific functions or particular computation, even if the information is at times redundant.

However, managing different data structures could be a nightmare in maintaining consistency across function calls. **cubfits** provides several utilities to generate (`gen*()`) or convert (`convert*()`) extended data structures, such as `reu13.list`, `n.list`, and `y.list`. See help pages (`?cubfits::DataGenerating` and `?cubfits::DataConverting`) for details.

Finally, **cubfits** provides a simple demo

R Script

```
> demo(basic, 'cubfits')
```

and shows how those generating and converting functions work. Note that this demo reads two example files from **cubfits** and turns them into main data structures which can be called by the three main functions introduced in Section 2. The files are `seq_200.fasta` containing 200 sequences and `phi_200.tsv` containing the corresponding expression levels. Both files are stored in the package directory `cubfits/inst/ex_data/` and installed in `${R_HOME}/library/cubfits/ex_data/`.

6. FAQ

1. **Q:** Should the amino acids and ORFs be sorted for **cubfits**?
   A: Yes. For performance issue, amino acids are coded with one character code, and ORFs and their expression data (if any) should be named. Both should be sorted by their names.

2. **Q:** What is the main difference of `cubfits()`, `cubappr()`, and `cubpred()`?
   A: `cubfits()` is the usual MCMC method to estimate parameters where some people call backward simulation. On the other hand, `cubappr()` is pure simulation without observations from equilibrium states (if they have been reached.) `cubpred()` is more like machine learning techniques, such as cross-validation methods, use part of observations to estimate parameters, and predict the other parts of data.

3. **Q:** Should `phi.Obs` be scaled?
   A: For `cubappr()`, it must be scaled to mean 1 since the function uses as `phi.Obs` as

initial values of ϕ_g. For `cubfits()` and `cubpred()`, it is not necessary. The bias terms should be also estimated if `phi.Obs` were not scaled to mean 1. Post scaling of each MCMC iteration should be performed on parameters and prediction of ϕ_g to make them comparable with `cubappr()` results. However, this may induce bias as well.

4. **Q:** What are configurations to enforce estimating bias of ϕ_g, K_{bias}?
   A: The only options currently **cubfits** have are in next.

Configuration

```
.CF.CT$type.p <- "lognormal_bias"
.CF.CT$scale.phi.Obs <- FALSE
.CF.CONF$estimate.bias.Phi <- TRUE
```

5. **Q:** What is the default environment of **cubfits**?
   A: `.cubfitsEnv` will dynamicly store generic functions, while all data are still located in `.GlobalEnv`. See Section 7 for examples.

6. **Q:** What are the ways to access **cubfits** functions?
   A: There are at least three levels of functions developed in **cubfits**:

   - `cubfits::function_name()` or simply `function_name()` (if **cubfits** is loaded) can access exported major functions of **cubfits**,
   - `cubfits:::function_name()` can access unexported internal functions and objects of **cubfits** (even **cubfits** is not loaded), and
   - `.cubfitsEnv$function_name()` can access generic functions dispatched in the environment `.cubfitsEnv` if `init_function()` has been called. (`ls(.cubfitsEnv)` can see what have been dispatched.)

7. **Q:** What is the way to debug **cubfits** functions?
   A: Debugger of negative R may not work well in some cases, so we suggest:

   - `cat()` or `print()` and `source()` can be useful for non-generic functions, and
   - `browser()` would be better for generic functions in `.cubfitsEnv`.

   Note that if you `source()` a new code into R, you may still have to overwrite functions in `.cubfitEnv` since MCMC iterations heavily access functions from there instead of `.GlobalEnv`.

7. Miscellaneous

7.1. Multinomial Logistic Regression

`fitMultinom()` is a utility function which can fit an ROC model with ϕ values, assuming no measurement errors as in Shah and Gilchrist (2011). A typical usage is:

R Script

```
> demo(fitMultinom, 'cubfit')
```

or

R Script

```
library(cubfits, quietly = TRUE)

# fit Shah & Gilchrist (2011)
init.function(model = "roc")
fitlist <- fitMultinom(ex.train$reu13.df, ex.train$phi.Obs,
                       ex.train$y, ex.train$n)
ret.fit <- prop.model.roc(fitlist, phi.Obs.lim =
   range(ex.train$phi.Obs))
aa.names <- names(ex.train$reu13.df)

# plot.
par(mfrow = c(1, 3))
for(i.aa in 1:length(aa.names)){
  plotmodel(ret.model = ret.fit[[i.aa]], main = aa.names[i.aa])
}
```

where `fitlist` is an object of `b` data structure containing all estimations of $(log(\mu), \Delta t)$ for each synonymous codon and amino acid. This is a quick fit assuming no measurement error on expression levels. This demo returns plots in Figure 6

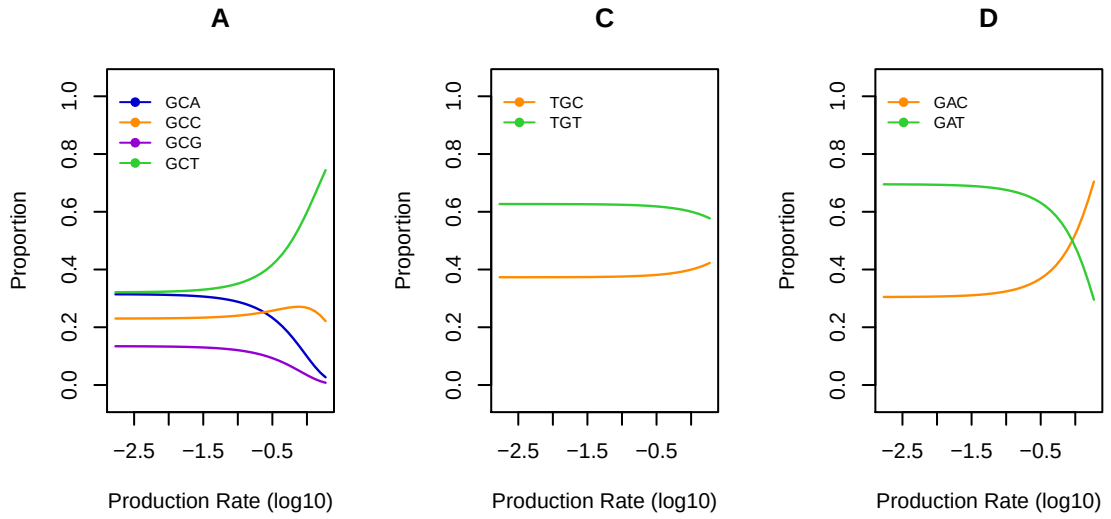


Figure 6: A simple prediction plot, similar to Figure 1 except empirical binning.

Important: Note that `init.function(model = "roc")` initializes generic functions for the ROC model and `fitMultinom()` can access corresponding generic functions from the `.cubfitsEnv` environment. The generic function then performs multinomial logistic regression on the input summarized statistics, and `VGAM::vglm()` can fit and return parameter estimations. Therefore, there is only need to initialize once before MCMC iterations, such as the three main function in Section 2. For performance issues, all calls within MCMC iterations should access generic functions, such as via `.cubfitEnv$fitMultinom()` regardless of

the model. Do **NOT** access `fitMultinom()` within any MCMC iteration.

7.2. Asymmetric Laplace Distribution

As with the regular R functions for different distributions (`dnorm()`, `pnorm()`, etc.), **cubfits** provides analogous functions for asymmetric Laplace distribution (ASL) such as `rasl()`, `dasl()`, `pasl()`, and `qasl()`. See (Kotz *et al.* 2001) for more details about the ASL distribution. Finding a MLE numerically is possible for ASL random samples and is implemented in the `asl.optim()`.

For example, Yassour dataset (Yassour *et al.* 2009) has 6303 gene expression measurements and four replicates. I took the geometric averaged mean of the replicates for each gene and fitted the ASL model to the means of 6303 genes as suggested by Wallace *et al.* (2013). The data distribution and the ASL fits are shown in Figures 7a which can be done as the next.

R Script

```
> demo(yassour.asl, 'cubfits')
```

Further, I also used normal mixture models to fit the same data using the EM algorithm implemented in the **EMCluster** (Chen *et al.* 2012a) package. Figure 7b shows the fits for $K = 1, 2, \ldots, 6$ components of normal mixture models, which can be done via a call to

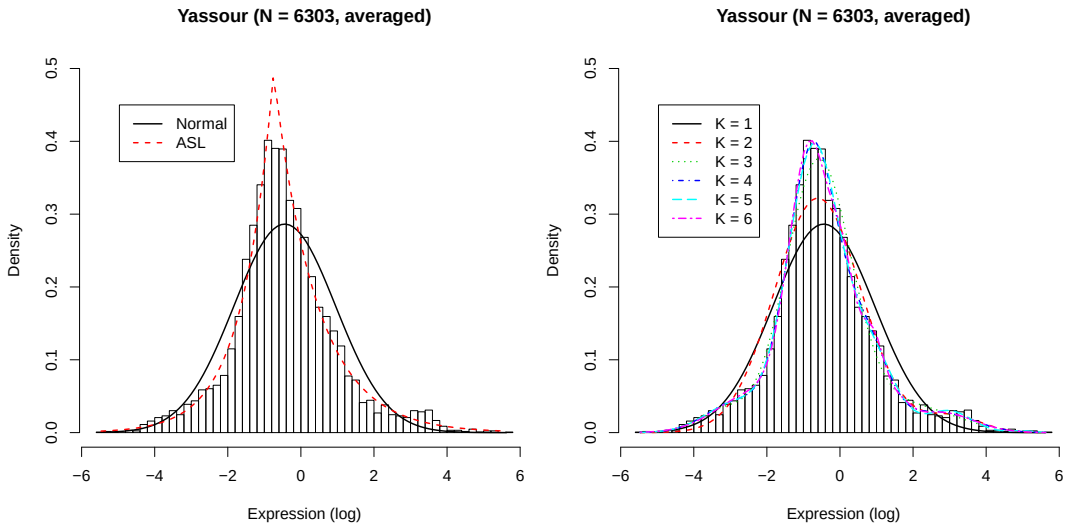
R Script

```
> demo(yassour.mixture, 'cubfits')
```

Table 1 provides some details for model comparison. Note that $K = 6$ has a smaller log likelihood than $K = 5$. This means that the EM algorithm converges to local optimum and may indicate an overestimated number of components. $K = 4$ is the best choice among all models by the smallest BIC.

Model	p	$\log L$	AIC	BIC
ASL	3	−10739.84	21485.68	21505.93
Normal / $K = 1$	2	−11033.55	22071.10	22084.60
$K = 2$	5	−10820.47	21650.93	21684.68
$K = 3$	8	−10687.06	21390.12	21444.11
$K = 4$	11	−10655.55	21333.10	21407.34
$K = 5$	14	−10649.69	21327.38	21421.87
$K = 6$	17	−10653.19	21340.38	21455.11

Table 1: K is the number of components for the mixture normal model and $K = 1$ is equivalent to normal model. p is the number of parameters for the fitted model, $\log L$ is the log likelihood, $AIC = -2\log L + 2p$, and $BIC = -2\log L + p\log(n)$ where $n = 6303$ is the number of samples.



(a) ASL model (b) Mixture normal model

Figure 7: Different distribution fits to Yassour dataset.

References

Charif D, Lobry J (2007). "SeqinR 1.0-2: a contributed package to the R project for statistical computing devoted to biological sequences retrieval and analysis." In U Bastolla, M Porto, H Roman, M Vendruscolo (eds.), *Structural approaches to sequence evolution: Molecules, networks, populations*, Biological and Medical Physics, Biomedical Engineering, pp. 207–232. Springer Verlag, New York. ISBN : 978-3-540-35305-8.

Chen WC, Maitra R, Melnykov V (2012a). "EMCluster: EM Algorithm for Model-Based Clustering of Finite Mixture Gaussian Distribution." R Package, URL http://cran.r-project.org/package=EMCluster.

Chen WC, Ostrouchov G, Schmidt D, Patel P, Yu H (2012b). "pbdMPI: Programming with Big Data – Interface to MPI." R Package, URL http://cran.r-project.org/package=pbdMPI.

Chen WC, Ostrouchov G, Schmidt D, Patel P, Yu H (2014a). *A Quick Guide for the pbdMPI package (Ver. 0.2-2)*. R Vignette, URL http://cran.r-project.org/package=pbdMPI.

Chen WC, Zaretzki R, Howell W, Schmidt D, Gilchrist M (2014b). "cubfits: Codon Usage Bias Fits." R Package, URL http://cran.r-project.org/package=cubfits.

Gilchrist M (2007). "Combining Models of Protein Translation and Population Genetics to Predict Protein Production Rates from Codon Usage Patterns." *Mol. Biol. Evol.*, **24**, 2362 – 2373.

Gilchrist M, Chen WC, Shah P, Zaretzki R (2014). "Quantifying expression and codon specific selection from genomic sequences." *Mol. Biol. Evol.* (in review).

Gropp W, Lusk E, Skjellum A (1994). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series.

Kotz S, Kozubowski T, Podgorski K (2001). *The Laplace distribution and generalizations: a revisit with applications to communications, economics, engineering, and finance*. Boston: Birkhauser.

Plummer M, Best N, Cowles K, Vines K (2006). "CODA: Convergence Diagnosis and Output Analysis for MCMC." *R News*, **6**(1), 7–11. URL http://CRAN.R-project.org/doc/Rnews/.

R Core Team (2012). "parallel: Support for Parallel Computation in R." R Package.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org.

Revolution Analytics, Weston S (2014). "doSNOW: Foreach parallel adaptor for the snow package." R Package version 1.0.12.

Shah P, Gilchrist M (2011). "Explaining Complex Codon Usage Patterns with Selection for Translational Efficiency, Mutation Bias, and Genetic Drift." *Proc. Natl. Acad. Sci. U.S.A.*, **108**, 10231 – 10236.

Wallace E, Airoldi E, Drummond D (2013). "Estimating Selection on Synonymous Codon Usage from Noisy Experimental Data." *Mol. Biol. Evol.*, **30**(6), 1438 – 1453.

Yassour M, Kaplan T, Fraser H, Levin J, Pfiffner J, Adiconis X, Schroth G, Luo S, Khrebtukova I, Gnirke A, Nusbaum C, Thompson D, Friedman N, Regev A (2009). "Ab initio construction of a eukaryotic transcriptome by massively parallel mRNA sequencing." *Proc. Natl. Acad. Sci. U.S.A.*, **106**, 3264 – 3269.

Yee T (2013). "VGAM: Vector Generalized Linear and Additive Models." R Package version 0.9-3, URL http://CRAN.R-project.org/package=VGAM.