A Quick Guide for the cubfits Package (Ver. 0.1-0)

Wei-Chen Chen[1], Russell Zaretzki[2], William Howell[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

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 0.1-0. 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 6 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 techniques

to speed up computation including **parallel** (R Core Team 2012) and **pbdMPI** (Chen *et al.* 2012b).

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("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 cubfits_0.1-0.tar.gz
```

provided other requirements 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:

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.

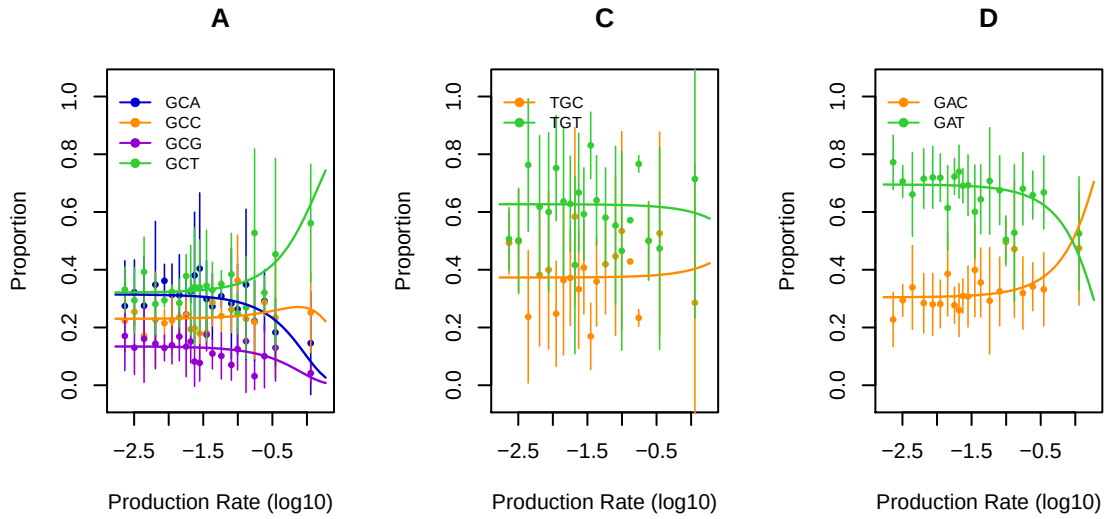


Figure 1: A simple demo plot. Figures show the empirical binning results by expression levels where dots are mean proportions of synonymous codons of 100 sequences in every 5% expression windows, and horizontal lines are 90% empirical intervals. The curves are theoretical prediction of synonymous codon usages (Shah and Gilchrist 2011). See Section 6 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.

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) model (Shah and Gilchrist 2011) which is shown 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),

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

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. Generic Functions (Aside)

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 6 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. Speedup (Aside)

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

- models assume conditional independent of amino acids, and
- tedious to parallelize within `VGAM::vglm()` function.[2]

Fortunately, carefully utilize summarized data structures can avoid some burdens and boost the MCMC computing a lot. For example, sorting by ORF's names before MCMC may avoid subset a `data.frame` in some computing, and further computing can be moved to `C` easily.

Further, **cubfits** consider three ways of parallel computations to speedup computations, including `parallel::mclapply()`, `pbdMPI::task.pull()`, and `pbdMPI::pbdLapply()`. Note that only **pbdMPI** functions are tested thoroughly in **cubfits** by default.

3.1. mclapply

The function `mclapply()` is a default function in **parallel** and particularly useful for multicores and shared memory machines, but only for Unix-like systems such as Linux and Mac OS. In Windows system, this function is the same as `lapply()` since forking mechanism differs from Unix-like systems. `mclapply()` is a parallel version of `lapply()`, and is easy to migrate from `lapply()`. So, this is first basic way to speedup **cubfits**.

We consider to split jobs (tasks) in the level of 20 amino acids since the conditional independence. Even though, the computation bottleneck held by some amino acids that have more synonymous codons than others, e.g. Leucine (Leu/L) has 6 synonymous codons. This means that 20 jobs are not homogeneous and Leu probably needs the longest computing in those `fitMultinom*()` calls. Therefore, we consider to use the option `mc.preschedule = FALSE` set to `mclapply()`, and limit the number of cores to 4 or 5. In such a way, we gain the performance economically.

However, in the warning section of `mclapply()` help page, it says *_strongly discouraged_* to use these functions in GUI or embedded environments, ... For small tests, `mclapply()` is working well and efficient in **cubfits**. Normally, less sequences and shorter MCMC iterations. **cubfits** did observe some crashes occasionally when using `mclapply()` for longer runs of MCMC, such as in a work flow.

3.2. task.pull

If GUI or embedded environments were not the main targets, then a stable way to speedup **cubfits** would be an idea way to aim for. **pbdMPI** is designed in single program multiple

[2] As long as number of sequences is not too large and summarized statistics is used, there is no need to consider this approach. Although this can lead to huge improvement of speedup, the cost may be too high that needs to rewrite whole `vglm()` and reorganize data structures.

data (SPMD) framework, and is a simpler approach to parallelize codes. Although only batch model is allowed in **pbdMPI**, one of the benefits is that it is not limited in multi-cores and shared memory machines, and can be applied on most systems provided a MPI library is installed. See Chen *et al.* (2014a) for more details of installation of a MPI library and **pbdMPI**.

The task pull is a simple way and has similar idea as `mc.preschedule = FALSE` in `mclapply()` that gain performance economically for non-homogeneous jobs. `task.pull()` in **pbdMPI** is a simple implementation that can boost `fitMultinom*()` functions, and has similar syntax as `mclapply()`. Currently, **cubfits** is mainly tested under this way.

3.3. pbdLapply

For some cases, task pull is not economic nor efficient. For example, in **cubfits** needs to initial expected expression level of each sequences via a call to `estimatePhiAll*()`. On average, this is a pretty much spending the same computation on all sequences. Therefore, task pull on 5,000 sequences may not a good idea since extra 5,000 calls for requesting unfinished sequences are unavoidable in the task pull way.

The function `pbdLapply()` is similar to set `mc.preschedule = TRUE` in `mclapply()`. The 5,000 sequences are divided by number of cores equally likely and processed by each core simultaneously. In such a way, there is no call for requesting sequences. Ideally, a good choice of number of cores is related to number of sequences. In general, it needs to be tuned with a few empirical runs.

3.4. Performance

Three different ways (`mclapply()`, `task.pull()`, and `task.pull()`) are tested on Yeast genome which has about 5346 genes and corresponding gene expression. Average run time of 200 iterations of `cubfits()` are recorded in different number of cores. Then, the speedup of parallelization is defined as each run time of given cores divides the run time of one core of `mclapply()`.

Figure 2 shows `task.pull()` is the best approach although it needs one more core for task management. There is no best choice for number of cores, but seven cores of `task.pull()` is pretty much reach a bottleneck. The performance degradation of eight cores of `pbdLapply()` is due to the unbalance pre-assignment of amino acid, e.g. Leu and Arg are probably assigned to the same core. The performance of `mclapply()` is surprisingly not well which may be due to memory copying between forking mechanism and `mc.preschedule = FALSE`.

4. Work Flows

Warning: All work flows are supposed to be run under batch mode and Unix-like systems due to performance.

cubfits is tested and built with some useful work flows, including

1. `simu` for simulation studies,
2. `wphi` for sequences with expression levels and measurement errors, and

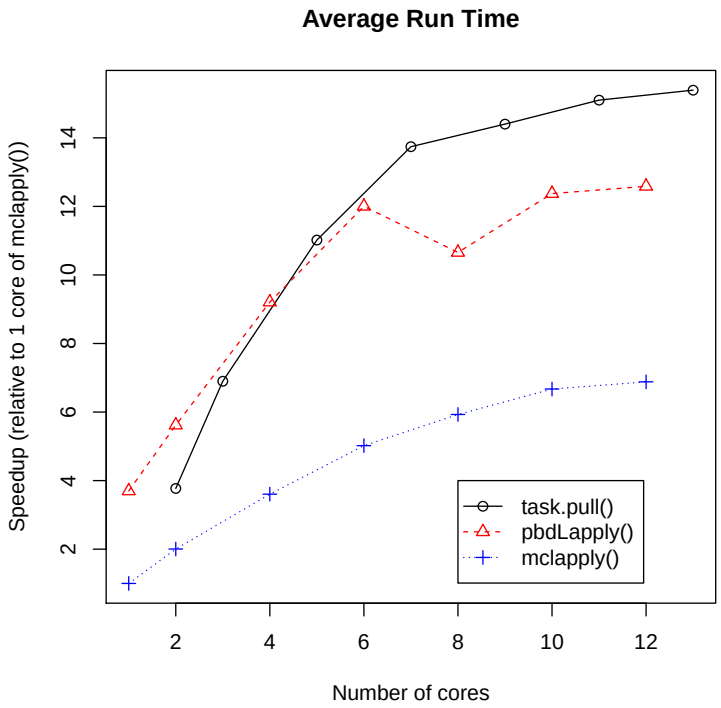


Figure 2: Speedup of three different parallelizations.

3. `wophi` for sequences without expression levels.

Upon the **cubfits** package release, we also share these work flows as templates which may be useful for further researches. A quick way to obtain a default work flow is via an internal function call `cubfits::cp.workflow()`. There are currently three work flows built within this call. Note that all of these are only tested privately under the Linux system, merely served as templates, none of them are checked by CRAN.

The work flows are stored in the package directory `cubfits/inst/workflow/` and installed in `${R_HOME}/library/cubfits/workflow/`. Therefore, execute the follow script can regenerate those work flows in three directories `./01-simu/`, `./02-wphi/`, and `./03-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
$ cd 03-wophi/
$ Rscript -e "cubfits::cp.workflow('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/code/` and installed in `${R_HOME}/library/cubfits/workflow/code/`. Those are served as templates and can be altered by users (copy scripts to local directory and change to what they should be.)

After regenerating those work flows, three shell scripts and one R script are created within each directories as

- `run_0.sh` is to create subdirectories for storages,
- `run_1.sh` is the main script of analysis,[3]
- `run_2.sh` is for post processes, and
- `00-set_env.r` is the R script for configurations.

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

Examples of post processes (`run_2.sh`) from `simu` work flow are given in Figure 3. 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`.

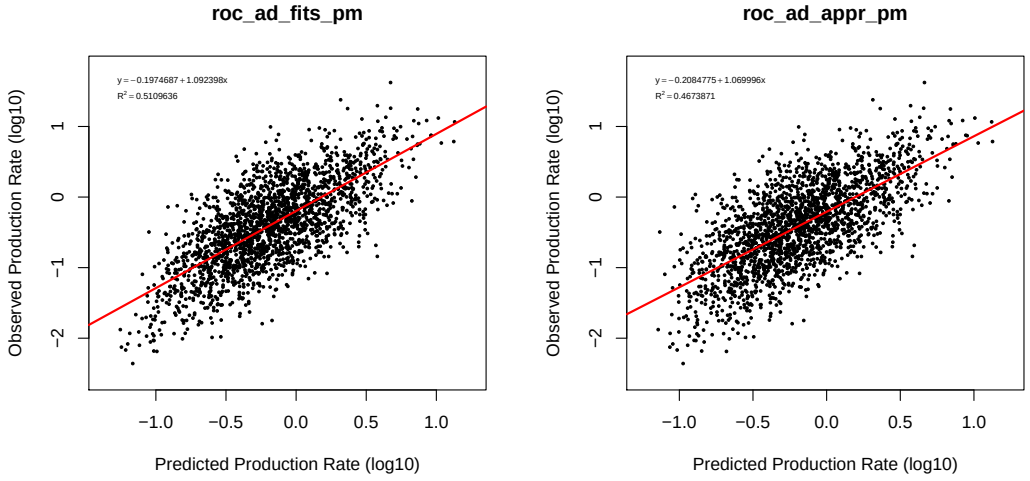


Figure 3: The left plot is predicted expression (MCMC posterior mean of expected expression) from model fits against observed expression (with measurement errors). In the right plot, the predicted expression is approximated from a simulation (no model fits.)

5. Utilities

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

The **cubfits** 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 standard format. As long as data are in 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 format after read in from disk or before write out to disk.

5.1. Data I/O Functions

- `read.seq()` is to read sequence data in FASTA format, and
- `write.seq()` is to write sequence data in FASTA format.

Both functions are simplified wrapper functions of **seqinr**. The writing function is mainly for simulation studies.

- `read.phi()` is to read in expression values in tsv/csv format, and
- `write.phi()` it to write out expression values in tsv/csv format.

Default data structure of these is a `data.frame`. The writing function is mainly for simulation studies.

5.2. Main Data Structures

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

- `reu13.df` is a data structure (used by REU13 students) contains codon positions and expression levels,
- `y` is a data structure (used by REU12 students and Wallace *et al.* (2013)) contains synonymous codon counts for each sequence,
- `n` is a data structure (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 ids 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 parallelize code is not enough. In R, a good data structure can be very different for computing performance. In practice, it is the best idea that not to change/modify/subset existing data structures within computations/iterations. As long as, data are not too big and in manageable size, for example in summarized statistics as Section 5.2. It is worth to generate extra data structures which is efficient for some specific functions or particular computation even the information are redundant.

However, managing different data structures could be a nightmare such that to be consistent 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.

Also, **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 to main data structures can be called by three main functions introduced in Section 2. The files are `seq_200.fasta` containing 200 sequences and `phi_200.tsv` containing 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. Miscellaneous

6.1. Multinomial Logistic Regression

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

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 4

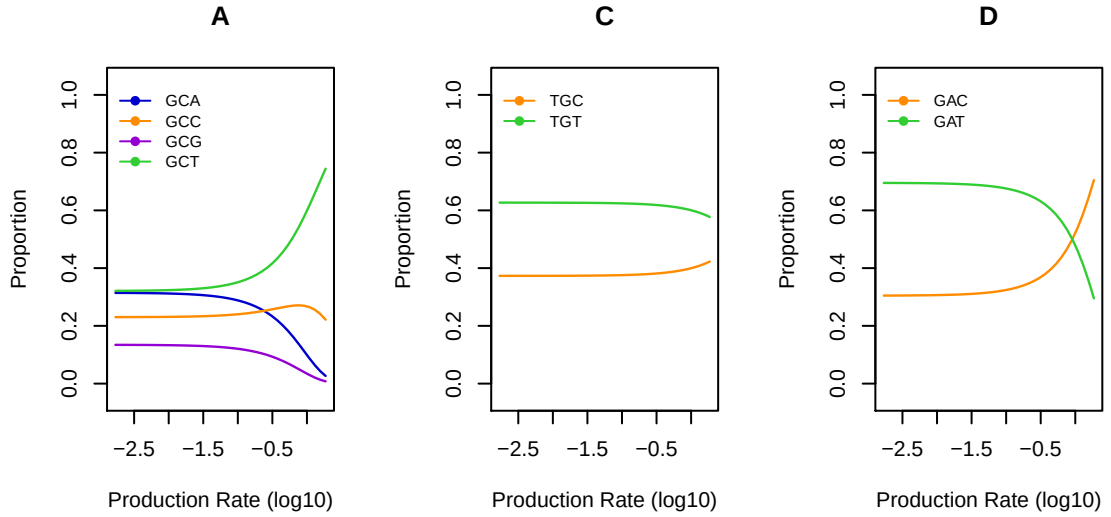


Figure 4: A simple prediction plot, similar to Figure 1 except empirical binning.

Important: Note that `init.function(model = "roc")` is to initial generic functions for the ROC model and `fitMultinom()` can access corresponding generic functions from environment `.cubfitsEnv`. The generic function then perform multinomial logistic regression on the input summarized statistics, and `VGAM::vglm()` can fit and return parameter estimations. Therefore, there is only need to initial once before MCMC iterations, such as the three main

function in Section 2. For performance issue, all calls within MCMC iterations should access generic functions, such as via `.cubfitEnv$fitMultinom()` regardless models. Do *NOT* access this function, `fitMultinom()`, within any MCMC iteration.

6.2. Asymmetric Laplace Distribution

As the regular R functions for different distributions, **cubfits** provides analog functions for asymmetric Laplace distribution (ASL) such as `rasl()`, `dasl()`, `pasl()`, and `qasl()`. See (Kotz *et al.* 2001) for more details of ASL. 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 5a which can be done as the next.

R Script

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

Further, I also used mixture normal models to fit the same data using the EM algorithm implemented in **EMCluster** (Chen *et al.* 2012a). Figures 5b shows the fits for $K = 1, 2, \ldots, 6$ components of mixture normal models which can be done as the next.

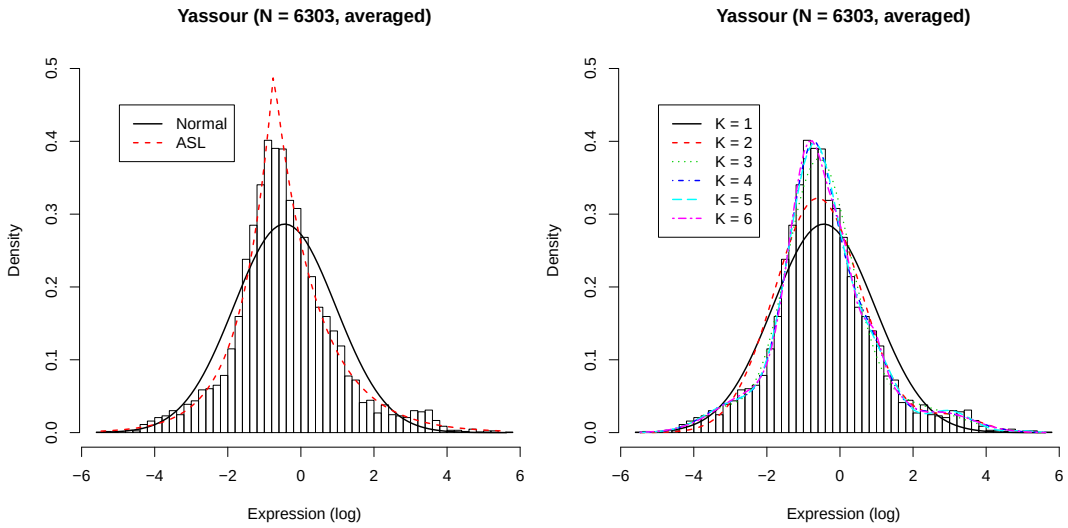
R Script

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

Table 1 provides some details for model comparison. Note that $K = 6$ has a smaller log likelihood than $K = 5$ that means 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 5: 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.

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.

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.

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.