Package 'cubfits'

May 8, 2026

Version 0.1-4

Date 2021-11-06

Title Codon Usage Bias Fits

Depends R(>= 4.0.0), methods, coda, foreach, parallel, stats, graphics, utils

Suggests seqinr, VGAM, EMCluster

Enhances pbdMPI (>= 0.3-1)

LazyLoad yes

LazyData yes

Description Estimating mutation and selection coefficients on synonymous codon bias usage based on models of ribosome overhead cost (ROC). Multinomial logistic regression and Markov Chain Monte Carlo are used to estimate and predict protein production rates with/without the presence of expressions and measurement errors. Work flows with examples for simulation, estimation and prediction processes are also provided with parallelization speedup. The whole framework is tested with yeast genome and gene expression data of Yassour, et al. (2009) <doi:10.1073/pnas.0812841106>.

License Mozilla Public License 2.0

BugReports https://github.com/snoweye/cubfits/issues

URL https://github.com/snoweye/cubfits

NeedsCompilation yes

Maintainer Wei-Chen Chen <wccsnow@gmail.com>

Author Wei-Chen Chen [aut, cre],
Russell Zaretzki [aut],
William Howell [aut],
Cedric Landerer [aut],
Drew Schmidt [aut],
Michael A. Gilchrist [aut],
Preston Hewgley [ctb],
Students REU13 [ctb]

Repository CRAN

Date/Publication 2021-11-07 17:20:02 UTC

Contents

cubfits-package . . . 2
Asymmetric Laplace Distribution . . . 4
Cedric Convergence Utilities . . . 6
Cedric IO Utilities . . . 7
Cedric Plot Utilities . . . 8
Codon Adaptation Index . . . 10
Controls . . . 11
Coverting Utility . . . 14
CUB Model Approximation . . . 16
CUB Model Fits . . . 18
CUB Model Prediction . . . 20
Data Formats . . . 23
Datasets . . . 24
Estimate Phi . . . 26
Fit Multinomial . . . 27
Generating Utility . . . 29
Initial Generic Functions . . . 30
Input and Output Utility . . . 33
Mixed Normal Optimization . . . 34
Plotbin . . . 36
Plotmodel . . . 37
Plotprxy . . . 39
Posterior Results of Yassour2009 . . . 40
Print . . . 41
Randomize SCUO Index . . . 42
Rearrangment Utility . . . 44
SCUO Index . . . 45
Selection on Codon Usage . . . 46
Simulation Tool . . . 47
Yassour2009 . . . 49

Index **51**

`cubfits-package` *Codon Bias Usage Fits*

Description

Estimating mutation and selection coefficients on synonymous codon bias usage based on models of ribosome overhead cost (ROC). Multinomial logistic regression and Markov Chain Monte Carlo are used to estimate and predict protein production rates with/without the presence of expressions and measurement errors.

Details

Package:	cubfits
Type:	Package
License:	Mozilla Public License 2.0
LazyLoad:	yes

The install command is simply as

```
> R CMD INSTALL cubfits_*.tar.gz
```

from a command mode or

```
R> install.packages("cubfits")
```

inside an R session.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>, Russell Zaretzki, William Howell, Drew Schmidt, and Michael Gilchrist.

References

https://github.com/snoweye/cubfits/

See Also

`init.function()`, `cubfits()`, `cubpred()`, and `cubappr()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

demo(roc.train, 'cubfits', ask = F, echo = F)
demo(roc.pred, 'cubfits', ask = F, echo = F)
demo(roc.appr, 'cubfits', ask = F, echo = F)

## End(Not run)
```

Asymmetric Laplace Distribution

The Asymmetric Laplace Distribution

Description

Density, probability, quantile, random number generation, and MLE functions for the asymmetric Laplace distribution with parameters either in $ASL(\theta, \mu, \sigma)$ or the alternative $ASL^*(\theta, \kappa, \sigma)$.

Usage

```
dasl(x, theta = 0, mu = 0, sigma = 1, log = FALSE)
dasla(x, theta = 0, kappa = 1, sigma = 1, log = FALSE)

pasl(q, theta = 0, mu = 0, sigma = 1, lower.tail = TRUE,
       log.p = FALSE)
pasla(q, theta = 0, kappa = 1, sigma = 1, lower.tail = TRUE,
        log.p = FALSE)

qasl(p, theta = 0, mu = 0, sigma = 1, lower.tail = TRUE,
       log.p = FALSE)
qasla(p, theta = 0, kappa = 1, sigma = 1, lower.tail = TRUE,
        log.p = FALSE)

rasl(n, theta = 0, mu = 0, sigma = 1)
rasla(n, theta = 0, kappa = 1, sigma = 1)

asl.optim(x)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If `length(n) > 1`, the length is taken to be the number required.
theta	center parameter.
mu, kappa	location parameters.
sigma	shape parameter.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are P[X <= x] otherwise, P[X > x].

Details

The density $f(x)$ of $ASL^*(\theta, \kappa, \sigma)$ is given as $\frac{\sqrt{2}}{\sigma}\frac{\kappa}{1+\kappa^2}exp(-\frac{\sqrt{2}\kappa}{\sigma}|x-\theta|)$ if $x \geq \theta$, and $\frac{\sqrt{2}}{\sigma}\frac{\kappa}{1+\kappa^2}exp(-\frac{\sqrt{2}}{\sigma\kappa}|x-\theta|)$ if $x < \theta$.

The parameter domains of ASL and ASL* are $\theta \in R$, $\sigma > 0$, $\kappa > 0$, and $\mu \in R$. The relation of μ and κ are $\kappa = \frac{\sqrt{2\sigma^2+\mu^2}-\mu}{\sqrt{2\sigma}}$ or $\mu = \frac{\sigma}{\sqrt{2}}(\frac{1}{\kappa} - \kappa)$.

Value

"dasl" and "dasla" give the densities, "pasl" and "pasla" give the distribution functions, "qasl" and "qasla" give the quantile functions, and "rasl" and "rasls" give the random numbers.

`asl.optim` returns the MLE of data `x` including `theta`, `mu`, `kappa`, and `sigma`.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

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

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

dasl(-2:2)
dasla(-2:2)
pasl(-2:2)
pasla(-2:2)
qasl(seq(0, 1, length = 5))
qasla(seq(0, 1, length = 5))

dasl(-2:2, log = TRUE)
dasla(-2:2, log = TRUE)
pasl(-2:2, log.p = TRUE)
pasla(-2:2, log.p = TRUE)
qasl(log(seq(0, 1, length = 5)), log.p = TRUE)
qasla(log(seq(0, 1, length = 5)), log.p = TRUE)

set.seed(123)
rasl(5)
rasla(5)

asl.optim(rasl(5000))

## End(Not run)
```

Cedric Convergence Utilities
Cedric Convergence Utilities

Description

This utility function provides convergence related functions by Cedric.

Usage

```
cubmultichain(cubmethod, reset.qr, seeds=NULL,
              teston=c("phi", "sphi"), swap=0, swapAt=0.05, monitor=NULL,
              min=0, max=160000, nchains=2, conv.thin=10,
              eps=0.1, ncores=2, ...)

cubsinglechain(cubmethod, frac1=0.1, frac2=0.5, reset.qr,
               seed=NULL, teston=c("phi", "sphi"), monitor=NULL,
               min=0, max=160000, conv.thin=10, eps=1, ...)
```

Arguments

cubmethod	String to choose method. Options are "cubfits", "cubappr", "cubpred"
reset.qr	recalculate QR decomposition matrix of covariance matrix until reset.qr samples are reached
swap	proportion of b matrix parameters to be swaped between convergence checks
swapAt	difference (L1-norm) between two consequtive convergence test leading to a swap in the b matrix
seeds	Vector of seed for random number generation
seed	Seed for random number generation
teston	Select data to test convergence on
monitor	A function to monitor the progress of the MCMC. The fucntions expects the result object and for cubmultichain an index i. (cubmultichain call: monitor(x,i), cubsinglechain call: monitor(x))
min	Minimum samples to be obtained. eps is ignored until number of samples reaches min
max	Maximum samples to be obtained. eps is ignored after max samples is obtained
eps	Convergence criterium
conv.thin	thinning of samples before performing convergence test
nchains	number of chains to run in parallel
ncores	number of cores to use for parallel execution of chains
frac1	fraction of samples at the beginning of set for Geweke test
frac2	fraction of samples at the end of set for Geweke test
...	named arguments for functions "cubfits", "cubappr" or "cubpred"

Details

under development

Value

under development

Author(s)

Cedric Landerer <cedric.landerer@gmail.com>.

References

https://github.com/clandere/cubfits/

See Also

cubfits, cubappr, cubpred

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))


## End(Not run)
```

Cedric IO Utilities *Cedric IO Utilities*

Description

This utility function provides basic IO by Cedric.

Usage

```
readGenome(fn.genome, ex.sh.aa = 0, rm.first.aa = 0)

normalizeDataSet(data)
```

Arguments

fn.genome	Fasta file with sequences
ex.sh.aa	Ignore sequences with a length less than ex.sh.aa. (After removal of the first rm.first.aa amino acids)
rm.first.aa	Remove the first rm.first.aa amino acids (after start codon)
data	Vector to be normalized. Means will be set to 1

Details

under development

Value

under development

Author(s)

Cedric Landerer <cedric.landerer@gmail.com>.

References

https://github.com/clandere/cubfits/

See Also

under development

Examples

```
## Not run:
  library(cubfits)
  seq.string <- readGenome("my_genome.fasta", 150, 10)

## End(Not run)
```

Cedric Plot Utilities *Cedric Plot Utilities*

Description

This utility function provides basic plots by Cedric.

Usage

```
plotPTraces(pMat, ...)

plotExpectedPhiTrace(phiMat, ...)

plotCUB(reu13.df.obs, bMat = NULL, bVec = NULL, phi.bin,
        n.use.samples = 2000, main = "CUB", model.label = c("True Model"),
        model.lty = 1, weightedCenters = TRUE)

plotTraces(bMat, names.aa, param = c("logmu", "deltaeta", "deltat"),
           main = "AA parameter trace")
```

Arguments

`reu13.df.obs`	under development
`bVec`	a parameter vector
`phi.bin`	phi values to bin for comparison
`n.use.samples`	under development
`main`	Main name for plotTraces
`model.label`	Name of model
`model.lty`	line type for model
`weightedCenters`	if centers are weighted.
`names.aa`	List of amino acids used for estimation
`param`	select to plot parameter trace for either log(mu) values or delta t
`phiMat`	phi matrix from the output of "cubmultichain", "cubsinglechain", "cubfits", "cubappr", or "cubpred"
`bMat`	b matrix from the output of "cubmultichain", "cubsinglechain", "cubfits", "cubappr", or "cubpred"
`pMat`	p matrix from the output of "cubmultichain", "cubsinglechain", "cubfits", "cubappr", or "cubpred"
`...`	other ploting options

Details

under development

Value

under development

Author(s)

Cedric Landerer <cedric.landerer@gmail.com>.

References

https://github.com/clandere/cubfits/

See Also

plot

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))


## End(Not run)
```

Codon Adaptation Index

Function for Codon Adaptation Index (CAI)

Description

Calculate the Codon Adaptation Index (CAI) for each gene. Used as a substitute for expression in cases of without expression measurements.

Usage

```
calc_cai_values(y, y.list, w = NULL)
```

Arguments

y	an object of format y.
y.list	an object of format y.list.
w	a specified relative frequency of synonymous codons.

Details

This function computes CAI for each gene. Typically, this method is completely based on entropy and information theory to estimate expression values of sequences according to their codon information.

If the input w is NULL, then empirical values are computed.

Value

A list with two named elements CAI and w are returned where CAI are CAI of input sequences (y and y.list) and w are the relative frequencey used to computed those CAI's.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

Sharp P.M. and Li W.-H. "The codon Adaptation Index – a measure of directional synonymous codon usage bias, and its potential applications" Nucleic Acids Res. 15 (3): 1281-1295, 1987.

See Also

calc_scuo_values(), calc_scu_values().

Examples

```
## Not run:
rm(list = ls())
library(cubfits, quietly = TRUE)

y <- ex.train$y
y.list <- convert.y.to.list(y)
CAI <- calc_cai_values(y, y.list)$CAI
plot(CAI, log10(ex.train$phi.Obs), main = "Expression vs CAI",
     xlab = "CAI", ylab = "Expression (log10)")

### Verify with the seqinr example.
library(seqinr, quietly = TRUE)
inputdatfile <- system.file("sequences/input.dat", package = "seqinr")
input <- read.fasta(file = inputdatfile, forceDNAtolower = FALSE)
names(input)[65] <- paste(names(input)[65], ".1", sep = "") # name duplicated.
input <- input[order(names(input))]

### Convert to cubfits format.
seq.string <- convert.seq.data.to.string(input)
new.y <- gen.y(seq.string)
new.y.list <- convert.y.to.list(new.y)
ret <- calc_cai_values(new.y, new.y.list)

### Rebuild w.
w <- rep(1, 64)
names(w) <- codon.low2up(rownames(caitab))
for(i in 1:64){
  id <- which(names(ret$w) == names(w)[i])
  if(length(id) == 1){
    w[i] <- ret$w[id]
  }
}
CAI.res <- sapply(input, seqinr::cai, w = w)

### Plot.
plot(CAI.res, ret$CAI,
     main = "Comparison of seqinR and cubfits results",
     xlab = "CAI from seqinR", ylab = "CAI from cubfits", las = 1)
abline(c(0, 1))

## End(Not run)
```

`Controls` *Default Controlling Options*

Description

Default controls of **cubfits** include for models, optimizations, MCMC, plotting, global variables, etc.

Usage

```
.cubfitsEnv
.CF.CT
.CF.CONF
.CF.GV
.CF.DP
.CF.OP
.CF.AC
.CF.PT
.CF.PARAM
.CO.CT
```

Format

All are in lists and contain several controlling options.

Details

See `init.function()` for use cases of these objects.

- `.cubfitEnv` is a default environment to dynamically save functions and objects.
- `.CF.CT` is main controls of models. It currently includes

`model`	main models
`type.p`	proposal for hyper-parameters
`type.Phi`	proposal for Phi
`model.Phi`	prior of Phi
`init.Phi`	initial methods for Phi
`init.fit`	how is coefficient proposed
`parallel`	parallel functions
`adaptive`	method for adaptive MCMC

- `.CF.CONF` controls the initial and draw scaling. It currently includes

`scale.phi.Obs`	if phi were scaled to mean 1
`init.b.Scale`	initial b scale
`init.phi.Scale`	initial phi scale
`p.nclass`	number of classes if mixture phi
`b.DrawScale`	drawing scale for b if random walk
`p.DrawScale`	drawing scale for p if random walk
`phi.DrawScale`	random walk scale for phi
`phi.pred.DrawScale`	random walk scale for phi.pred
`sigma.Phi.DrawScale`	random walk scale for sigma.Phi
`bias.Phi.DrawScale`	random walk scale for bias.Phi
`estimate.bias.Phi`	if estimate bias of phi during MCMC
`compute.logL`	if compute logL in each iteration

- `.CF.GV` contains global variables for amino acids and codons. It currently includes

`amino.acid`	amino acids
`amino.acid.3`	amino acids
`synonymous.codon`	synonymous codons of amino acids
`amino.acid.split`	amino acid 'S' is split
`amino.acid.split.3`	amino acid 'S' is split
`synonymous.codon.split`	synonymous codons of split amino acid

- `.CF.OP` controls optimizations. It currently includes

`optim.method`	method for `optim()`
`stable.min.exp`	minimum exponent
`stable.max.exp`	maximum exponent
`E.Phi`	expected Phi
`lower.optim`	lower of derivative of logL(x)
`upper.optim`	upper of derivative of logL(x)
`lower.integrate`	lower of integration of L(x)
`upper.integrate`	upper of integration of L(x)

- `.CF.DP` is for dumping MCMC iterations. It currently includes

`dump`	if dumping within MCMC
`iter`	iterations per dumping
`prefix.dump`	path and file names of dumping
`verbose`	if verbose
`iterThin`	iterations to thin chain
`report`	iterations to report
`report.proc`	iterations to report `proc.time()`

- `.CF.AC` controls adaptive MCMC. It currently includes

`renew.iter`	per renewing iterations
`target.accept.lower`	target acceptant rate lower bound
`target.accept.upper`	target acceptant rate upper bound
`scale.increase`	increase scale size
`scale.decrease`	decrease scale size
`sigma.lower`	lower bound of relative scale size
`sigma.upper`	upper bound of relative scale size

- `.CF.PT` controls the plotting format. It currently includes

`color`	color for codons.

- `.CF.PARAM` controls the parameters and hyperparameters of priors. It currently includes

`phi.meanlog`	mean of phi in loca scale
`phi.sdlog`	standard deviation of phi in loca scale

- `.CO.CT` controls the constrained optimization function. It currently includes

  debug message printing level of debugging.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

`init.function()`, `cubfits()`, `cubpred()`, `cubappr()`, and `mixnormerr.optim()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

.CF.CT
.CF.CONF
.CF.DP
.CF.GV
.CF.OP
.CF.AC
.CF.PT
.CF.PARAM
.CO.CT

ls(.cubfitsEnv)
init.function()
ls(.cubfitsEnv)

## End(Not run)
```

Coverting Utility *Convert Data Frame to Other Formats*

Description

These utility functions convert data of format divided by amino acids into list of format divided by ORFs, or convert data to other formats.

Usage

```
convert.reu13.df.to.list(reu13.df)
convert.y.to.list(y)
convert.n.to.list(n)

convert.y.to.scuo(y)
convert.seq.data.to.string(seq.data)

codon.low2up(x)
codon.up2low(x)

dna.low2up(x)
dna.up2low(x)

convert.b.to.bVec(b)
convert.bVec.to.b(bVec, aa.names, model = .CF.CT$model[1])
```

Arguments

reu13.df	a list of reu13.df data frames divided by amino acids.
y	a list of y data frames divided by amino acids.
n	a list of n vectors divided by amino acids.
seq.data	a vector of seq.data format.
x	a codon or dna string, such "ACG", "acg", or "A", "a".
b	a b object.
bVec	a bVec object.
aa.names	a vector contains amino acid names for analysis.
model	model fitted.

Details

convert.reu13.df.to.list(), convert.y.to.list(), and convert.n.to.list(): these utility functions take the inputs divided by amino acids and return the outputs divided by ORFs.

convert.y.scuo() converts y into scuo format.

convert.seq.data.to.string() converts seq.data into seq.string format.

codon.low2up() and codon.up2low() convert codon strings between lower or upper cases.

convert.bVec.to.b() and convert.b.to.bVec() convert objects b and bVec.

Value

All functions return the corresponding formats.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

AllDataFormats, rearrange.n(), rearrange.reu13.df(), rearrange.y(), and read.seq().

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

reu13.list <- convert.reu13.df.to.list(ex.train$reu13.df)
y.list <- convert.y.to.list(ex.train$y)
n.list <- convert.n.to.list(ex.train$n)

scuo <- convert.y.to.scuo(ex.train$y)

seq.data <- read.seq(get.expath("seq_200.fasta"))
seq.string <- convert.seq.data.to.string(seq.data)

codon.low2up("acg")
codon.up2low("ACG")

dna.low2up(c("a", "c", "g"))
dna.up2low(c("A", "C", "G"))

## End(Not run)
```

CUB Model Approximation

Codon Usage Bias Approximation for ORFs without Expression

Description

This function provides codon usage bias approximation with observed ORFs but without any expressions.

Usage

```
cubappr(reu13.df.obs, phi.pred.Init, y, n,
        nIter = 1000,
        b.Init = NULL, init.b.Scale = .CF.CONF$init.b.Scale,
            b.DrawScale = .CF.CONF$b.DrawScale,
            b.RInit = NULL,
        p.Init = NULL, p.nclass = .CF.CONF$p.nclass,
            p.DrawScale = .CF.CONF$p.DrawScale,
        phi.pred.DrawScale = .CF.CONF$phi.pred.DrawScale,
```

```
model = .CF.CT$model[1], model.Phi = .CF.CT$model.Phi[1],
adaptive = .CF.CT$adaptive[1],
verbose = .CF.DP$verbose,
iterThin = .CF.DP$iterThin, report = .CF.DP$report)
```

Arguments

`reu13.df.obs`	a `reu13.df` object, ORFs information.
`phi.pred.Init`	a `phi.Obs` object, temporarily initial of expression without measurement errors.
`y`	a `y` object, codon counts.
`n`	a `n` object, total codon counts.
`nIter`	number of iterations after burn-in iterations.
`b.Init`	initial values for parameters `b`.
`init.b.Scale`	for initial `b` if `b.Init = NULL`.
`b.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new `b`.
`b.RInit`	initial values (in a list) for `R` matrices of parameters `b` yielding from QR decomposition of `vglm()` for the variance-covariance matrix of `b`.
`p.Init`	initial values for hyper-parameters.
`p.nclass`	number of components for `model.Phi = "logmixture"`.
`p.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new `sigma.Phi`.
`phi.pred.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new Phi of predicted set.
`model`	model to be fitted, currently "roc" only.
`model.Phi`	prior model for Phi, currently "lognormal".
`adaptive`	adaptive method of MCMC for proposing new `b` and Phi.
`verbose`	print iteration messages.
`iterThin`	thinning iterations.
`report`	number of iterations to report more information.

Details

Total number of MCMC iterations is `nIter + 1`, but the outputs may be thinned to `nIter / iterThin + 1` iterations.

Temporary result dumping may be controlled by `.CF.DP`.

Value

A list contains three big lists of MCMC traces including: `b.Mat` for mutation and selection coefficients of `b`, `p.Mat` for hyper-parameters, and `phi.Mat` for expected expression values Phi. All lists are of length `nIter / iterThin + 1` and each element contains the output of each iteration.

All lists also can be binded as trace matrices, such as via `do.call("rbind", b.Mat)` yielding a matrix of dimension number of iterations by number of parameters. Then, those traces can be analyzed further via other MCMC packages such as **coda**.

Note

Note that `phi.pred.Init` need to be normalized to mean 1.

`p.DrawScale` may cause scaling prior if adaptive MCMC is used, and it can result in non-exits of equilibrium distribution.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

DataIO, DataConverting, `cubfits()` and `cubpred()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

demo(roc.appr, 'cubfits', ask = F, echo = F)

## End(Not run)
```

CUB Model Fits *Codon Usage Bias Fits for Observed ORFs and Expression*

Description

This function provides codon usage bias fits with observed ORFs and expressions which possibly contains measurement errors.

Usage

```
cubfits(reu13.df.obs, phi.Obs, y, n,
        nIter = 1000,
        b.Init = NULL, init.b.Scale = .CF.CONF$init.b.Scale,
            b.DrawScale = .CF.CONF$b.DrawScale,
            b.RInit = NULL,
        p.Init = NULL, p.nclass = .CF.CONF$p.nclass,
            p.DrawScale = .CF.CONF$p.DrawScale,
        phi.Init = NULL, init.phi.Scale = .CF.CONF$init.phi.Scale,
            phi.DrawScale = .CF.CONF$phi.DrawScale,
        model = .CF.CT$model[1], model.Phi = .CF.CT$model.Phi[1],
        adaptive = .CF.CT$adaptive[1],
        verbose = .CF.DP$verbose,
        iterThin = .CF.DP$iterThin, report = .CF.DP$report)
```

Arguments

`reu13.df.obs`	a `reu13.df` object, ORFs information.
`phi.Obs`	a `phi.Obs` object, expression with measurement errors.
`y`	a `y` object, codon counts.
`n`	a `n` object, total codon counts.
`nIter`	number of iterations after burn-in iterations.
`b.Init`	initial values for parameters `b`.
`init.b.Scale`	for initial `b` if `b.Init = NULL`.
`b.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new `b`.
`b.RInit`	initial values (in a list) for `R` matrices of parameters `b` yielding from QR decomposition of `vglm()` for the variance-covariance matrix of `b`.
`p.Init`	initial values for hyper-parameters.
`p.nclass`	number of components for `model.Phi = "logmixture"`.
`p.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new `sigma.Phi`.
`phi.Init`	initial values for Phi.
`init.phi.Scale`	for initial phi if `phi.Init = NULL`.
`phi.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new Phi.
`model`	model to be fitted, currently "roc" only.
`model.Phi`	prior model for Phi, currently "lognormal".
`adaptive`	adaptive method of MCMC for proposing new `b` and Phi.
`verbose`	print iteration messages.
`iterThin`	thinning iterations.
`report`	number of iterations to report more information.

Details

This function correctly and carefully implements a combining version of Shah and Gilchrist (2011) and Wallace et al. (2013).

Total number of MCMC iterations is `nIter + 1`, but the outputs may be thinned to `nIter / iterThin + 1` iterations.

Temporary result dumping may be controlled by `.CF.DP`.

Value

A list contains three big lists of MCMC traces including: `b.Mat` for mutation and selection coefficients of `b`, `p.Mat` for hyper-parameters, and `phi.Mat` for expected expression values Phi. All lists are of length `nIter / iterThin + 1` and each element contains the output of each iteration.

All lists also can be binded as trace matrices, such as via `do.call("rbind", b.Mat)` yielding a matrix of dimension number of iterations by number of parameters. Then, those traces can be analyzed further via other MCMC packages such as **coda**.

Note

Note that `phi.Init` need to be normalized to mean 1.

`p.DrawScale` may cause scaling prior if adaptive MCMC is used, and it can result in non-exits of equilibrium distribution.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

Shah P. and Gilchrist M.A. "Explaining complex codon usage patterns with selection for translational efficiency, mutation bias, and genetic drift" Proc Natl Acad Sci USA (2011) 108:10231–10236.

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

See Also

DataIO, DataConverting, `cubappr()` and `cubpred()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

demo(roc.train, 'cubfits', ask = F, echo = F)

## End(Not run)
```

`CUB Model Prediction` *Codon Usage Bias Prediction for Observed ORFs*

Description

This function provides codon usage bias fits of training set which has observed ORFs and expressions possibly containing measurement errors, and provides predictions of testing set which has other observed ORFs but without expression.

Usage

```
cubpred(reu13.df.obs, phi.Obs, y, n,
        reu13.df.pred, y.pred, n.pred,
        nIter = 1000,
        b.Init = NULL, init.b.Scale = .CF.CONF$init.b.Scale,
            b.DrawScale = .CF.CONF$b.DrawScale,
            b.RInit = NULL,
        p.Init = NULL, p.nclass = .CF.CONF$p.nclass,
            p.DrawScale = .CF.CONF$p.DrawScale,
        phi.Init = NULL, init.phi.Scale = .CF.CONF$init.phi.Scale,
            phi.DrawScale = .CF.CONF$phi.DrawScale,
        phi.pred.Init = NULL,
            phi.pred.DrawScale = .CF.CONF$phi.pred.DrawScale,
        model = .CF.CT$model[1], model.Phi = .CF.CT$model.Phi[1],
        adaptive = .CF.CT$adaptive[1],
        verbose = .CF.DP$verbose,
        iterThin = .CF.DP$iterThin, report = .CF.DP$report)
```

Arguments

`reu13.df.obs`	a `reu13.df` to be trained.
`phi.Obs`	a `phi.Obs` to be trained.
`y`	a `y` to be trained.
`n`	a `n` to be trained.
`reu13.df.pred`	a `reu13.df` to be predicted.
`y.pred`	a `y` to be predicted.
`n.pred`	a `n` to be predicted.
`nIter`	number of iterations after burn-in iterations.
`b.Init`	initial values for parameters `b`.
`init.b.Scale`	for initial `b` if `b.Init = NULL`.
`b.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new `b`.
`b.RInit`	initial values (in a list) for `R` matrices of parameters `b` yielding from QR decomposition of `vglm()` for the variance-covariance matrix of `b`.
`p.Init`	initial values for hyper-parameters.
`p.nclass`	number of components for `model.Phi = "logmixture"`.
`p.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new `sigma.Phi`.
`phi.Init`	initial values for Phi.
`init.phi.Scale`	for initial phi if `phi.Init = NULL`.
`phi.DrawScale`	scaling factor for adaptive MCMC with random walks when drawing new Phi.
`phi.pred.Init`	initial values for Phi of predicted set.
`phi.pred.DrawScale`	as `phi.DrawScale` but for predicted set.

model	model to be fitted, currently "roc" only.
model.Phi	prior model for Phi, currently "lognormal".
adaptive	adaptive method of MCMC for proposing new b and Phi.
verbose	print iteration messages.
iterThin	thinning iterations.
report	number of iterations to report more information.

Details

This function correctly and carefully implements an extension of Shah and Gilchrist (2011) and Wallace et al. (2013).

Total number of MCMC iterations is nIter + 1, but the outputs may be thinned to nIter / iterThin + 1 iterations.

Temporary result dumping may be controlled by .CF.DP.

Value

A list contains four big lists of MCMC traces including: b.Mat for mutation and selection coefficients of b, p.Mat for hyper-parameters, phi.Mat for expected expression values Phi, and phi.pred.Mat for predictive expression values Phi. All lists have nIter / iterThin + 1 elements, and each element contains the output of each iteration.

All lists also can be binded as trace matrices, such as via do.call("rbind", b.Mat) yielding a matrix of dimension number of iterations by number of parameters. Then, those traces can be analyzed further via other MCMC packages such as **coda**.

Note

Note that phi.Init and phi.pred.Init need to be normalized to mean 1.

p.DrawScale may cause scaling prior if adaptive MCMC is used, and it can result in non-exits of equilibrium distribution.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

Shah P. and Gilchrist M.A. "Explaining complex codon usage patterns with selection for translational efficiency, mutation bias, and genetic drift" Proc Natl Acad Sci USA (2011) 108:10231–10236.

See Also

DataIO, DataConverting, cubfits() and cubappr().

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

demo(roc.pred, 'cubfits', ask = F, echo = F)

## End(Not run)
```

`Data Formats` *Data Formats*

Description

Data formats used in **cubfits**.

Format

All are in simple formats as S3 default lists or data frames.

Details

- Format `b`:
  A named list `A` contains amino acids. Each element of the list `A[[i]]` is a list of elements `coefficients` (coefficients of log(mu) and Delta.t), `coef.mat` (matrix format of `coefficients`), and `R` (covariance matrix of `coefficients`). Note that `coefficients` and `R` are typically as in the output of `vglm()` of **VGAM** package. Also, `coef.mat` and `R` may miss in some cases.
  e.g. `A[[i]]$coef.mat` is the regression beta matrix of `i`-th amino acid.
- Format `bVec`:
  A vector simply contains all coefficients of a `b` object `A`. Note that this is probably only used inside MCMC or the output of `vglm()` of **VGAM** package.
  e.g. `do.call("c", lapply(A, function(x) x$coefficients))`.
- Format `n`:
  A named list `A` contains amino acids. Each element of the list `A[[i]]` is a vector containing total codon counts.
  e.g. `A[[i]][j]` is for `j`-th ORF of `i`-th amino acid `names(A)[i]`.
- Format `n.list`:
  A named list `A` contains ORFs. Each element of the list `A[[i]]` is a named list of amino acid containing total count.
  e.g. `A[[i]][[j]]` contains total count of `j`-th amino acid in `i`-th ORF.
- Format `phi.df`:
  A data frame `A` contains two columns `ORF` and `phi.value`.
  e.g. `A[i,]` is for `i`-th ORF.
- Format `reu13.df`:
  A named list `A` contains amino acids. Each element is a data frame summarizing ORF and expression. The data frame has four to five columns including `ORF`, `phi` (expression), `Pos` (amino acid position), `Codon` (synonymous codon), and `Codon.id` (synonymous codon id, for

computing only). Note that `Codon.id` may miss in some cases.
e.g. `A[[i]][17,]` is the 17-th recode of `i`-th amino acid.

- Format `reu13.list`:
A named list `A` contains ORFs. Each element is a named list `A[[i]]` contains amino acids. Each element of nested list `A[[i]][[j]]` is a position vector of synonymous codon.
e.g. `A[[i]][[j]][k]` is the `k`-th synonymous codon position of `j`-th amino acid in the `i`-th ORF.
- Format `scuo`:
A data frame of 8 named columns includes `AA` (amino acid), `ORF`, `C1`, ..., `C6` where `C*`'s are for codon counts.
- Format `seq.string`:
Default outputs of `read.fasta()` of **seqinr** package. A named list `A` contains ORFs. Each element of the list is a long string of a ORF.
e.g. `A[[i]][1]` or `A[[i]]` is the sequence of `i`-th ORF.
- Format `seq.data`:
Converted from `seq.string` format. A named list `A` contains ORFs. Each element of the list `A[[i]]` is a string vector. Each element of the vector is a codon string.
e.g. `A[[i]][j]` is `i`-th ORF and `j`-th codon.
- Format `phi.Obs`:
A named vector `A` of observed expression values and possibly with measurement errors.
e.g. `A[i]` is the observed phi value of `i`-th ORF.
- Format `y`:
A named list `A` contains amino acids. Each element of the list `A[[i]]` is a matrix where ORFs are in row and synonymous codons are in column. The element of the matrix contains codon counts.
e.g. `A[[i]][j, k]` is the count for `i`-th amino acid, `j`-th ORF, and `k`-th synonymous codon.
- Format `y.list`:
A named list `A` contains ORFs. Each element of the list `A[[i]]` is a named list `A[[i]][[j]]` contains amino acids. The element of amino acids list is a codon count vector.
e.g. `A[[i]][[j]][k]` is the count for `i`-th ORF, `j`-th amino acid, and `k`-th synonymous codon.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

`Datasets` *Datasets for Demonstrations*

Description

Examples of toy data to test and demonstrate **cubfits**.

Usage

```
b.Init
ex.test
ex.train
```

Format

All are in list formats.

Details

`b.Init` contains two sets (`roc` and `rocnse`) of initial coefficients including mutation and selection parameters for 3 amino acids 'A', 'C', and 'D' in `matrix` format. Both sets are in `b` format.

`ex.train` contains a training set of 100 sequences including 3 `reu13.df` (codon counts in `reu13` data frame format divided by amino acids), 3 `y` (codon counts in simplified data frame format divided by amino acids), 3 `n` (total amino acid counts in vector format divided by amino acids), and `phi.Obs` (observed phi values in vector format).

`ex.test` contains a testing set of the other 100 sequences in the same format of `ex.train`.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

`init.function()`, `cubfits()`, `cubpred()`, and `cubappr()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

str(b.Init)
str(ex.test)
str(ex.train)

## End(Not run)
```

Estimate Phi *Initialization of Phi (Generic)*

Description

This generic function estimates Phi (expression value) either by posterior mean (PM) or by maximum likelihood estimator (MLE) depending on options set by `init.function()`.

Usage

```
estimatePhi(fitlist, reu13.list, y.list, n.list,
  E.Phi = .CF.OP$E.Phi, lower.optim = .CF.OP$lower.optim,
  upper.optim = .CF.OP$upper.optim,
  lower.integrate = .CF.OP$lower.integrate,
  upper.integrate = .CF.OP$upper.integrate, control = list())
```

Arguments

`fitlist`	an object of format `b`.
`reu13.list`	an object of format `reu13.list`.
`y.list`	an object of format `y.list`.
`n.list`	an object of format `n.list`.
`E.Phi`	potential expected value of Phi.
`lower.optim`	lower bound to `optim()`.
`upper.optim`	upper bound to `optim()`.
`lower.integrate`	lower bound to `integrate()`.
`upper.integrate`	upper bound to `integrate()`.
`control`	control options to `optim()`.

Details

`estimatePhi()` is a generic function first initialized by `init.function()`, then it estimates Phi accordingly. By default, `.CF.CT$init.Phi` sets the method `PM` for the posterior mean.

`PM` uses a flat prior and `integrate()` to estimate Phi. While, `MLE` uses `optim()` to estimate Phi which may have boundary solutions for some sequences.

Value

Estimated Phi for every sequence is returned.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

`init.function()` and `fitMultinom()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

# Convert data.
reu13.list <- convert.reu13.df.to.list(ex.test$reu13.df)
y.list <- convert.y.to.list(ex.test$y)
n.list <- convert.n.to.list(ex.test$n)

# Get phi.pred.Init
init.function(model = "roc")
fitlist <- fitMultinom(ex.train$reu13.df, ex.train$phi.Obs, ex.train$y, ex.train$n)
phi.pred.Init <- estimatePhi(fitlist, reu13.list, y.list, n.list,
                         E.Phi = median(ex.test$phi.Obs),
                         lower.optim = min(ex.test$phi.Obs) * 0.9,
                         upper.optim = max(ex.test$phi.Obs) * 1.1)

## End(Not run)
```

`Fit Multinomial` *Fit Multinomial Model (Generic)*

Description

This generic function estimates `b` (mutation (log(mu)) and selection (Delta.t) parameters) depending on options set by `init.function()`.

Usage

```
fitMultinom(reu13.df, phi, y, n, phi.new = NULL, coefstart = NULL)
```

Arguments

`reu13.df`	an object of format `reu13.df`.
`phi`	an object of format `phi.Obs`.
`y`	an object of format `y`.
`n`	an object of format `n`.
`phi.new`	an object of format `phi.Obs` for MCMC only.
`coefstart`	initial value for `b` (mutation (log(mu)) and selection (Delta.t) parameters) only used in `vglm()`.

Details

`fitMultinom()` fits a multinomial logistic regression via vector generalized linear model fitting, `vglm()`. By default, for each amino acids, the last codon (order by characters) is assumed as a based line, and other codons are compared to the based line relatively.

In MCMC, `phi.new` are new proposed expression values and used to propose new `b`. The `coefstart` is used to avoid randomization of estimating `b` in `vglm()`, and speed up computation.

Value

A list of format `b` is returned which are modified from the returns of `vglm()`. Mainly, it includes `b$coefficient` (parameters in `vector`), `b$coef.mat` (parameters in `matrix`), and `b$R` (covariance matrix of parameters, *R* matrix in QR decomposition).

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

Shah P. and Gilchrist M.A. "Explaining complex codon usage patterns with selection for translational efficiency, mutation bias, and genetic drift" Proc Natl Acad Sci USA (2011) 108:10231–10236.

See Also

`init.function()` and `estimatePhi()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

# Convert data.
reu13.list <- convert.reu13.df.to.list(ex.test$reu13.df)
y.list <- convert.y.to.list(ex.test$y)
n.list <- convert.n.to.list(ex.test$n)

# Get phi.pred.Init
init.function(model = "roc")
fitlist <- fitMultinom(ex.train$reu13.df, ex.train$phi.Obs, ex.train$y, ex.train$n)
phi.pred.Init <- estimatePhi(fitlist, reu13.list, y.list, n.list,
                         E.Phi = median(ex.test$phi.Obs),
                         lower.optim = min(ex.test$phi.Obs) * 0.9,
                         upper.optim = max(ex.test$phi.Obs) * 1.1)

## End(Not run)
```

`Generating Utility` *Generating Data Structure*

Description

These utility functions generate and summarize sequence strings into several useful formats such as `reu13.df`, `y`, and `n`, etc.

Usage

```
gen.reu13.df(seq.string, phi.df = NULL, aa.names = .CF.GV$amino.acid,
             split.S = TRUE, drop.X = TRUE, drop.MW = TRUE,
             drop.1st.codon = TRUE)
gen.y(seq.string, aa.names = .CF.GV$amino.acid,
      split.S = TRUE, drop.X = TRUE, drop.MW = TRUE)
gen.n(seq.string, aa.names = .CF.GV$amino.acid,
      split.S = TRUE, drop.X = TRUE, drop.MW = TRUE)

gen.reu13.list(seq.string, aa.names = .CF.GV$amino.acid,
               split.S = TRUE, drop.X = TRUE, drop.MW = TRUE,
               drop.1st.codon = TRUE)
gen.phi.Obs(phi.df)
gen.scuo(seq.string, aa.names = .CF.GV$amino.acid,
         split.S = TRUE, drop.X = TRUE, drop.MW = TRUE)
```

Arguments

`seq.string`	a list of sequence strings.
`phi.df`	a `phi.df` object returned from `read.phi.df()`.
`aa.names`	a vector contains amino acid names for analysis.
`split.S`	split amino acid 'S' if any.
`drop.X`	drop amino acid 'X' if any.
`drop.MW`	drop amino acid 'M' and 'W' if any.
`drop.1st.codon`	if drop the first codon.

Details

These functions mainly take inputs of sequence strings `seq.string` or `phi.df` and turn them into corresponding format.

Value

The outputs are data structure in corresponding formats. See AllDataFormats for details.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

AllDataFormats, `read.seq()`, `read.phi.df()`, and `convert.seq.data.to.string()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

seq.data <- read.seq(get.expath("seq_200.fasta"))
phi.df <- read.phi.df(get.expath("phi_200.tsv"))
aa.names <- c("A", "C", "D")

# Read in from FASTA file.
seq.string <- convert.seq.data.to.string(seq.data)
reu13.df <- gen.reu13.df(seq.string, phi.df, aa.names)
reu13.list.new <- gen.reu13.list(seq.string, aa.names)
y <- gen.y(seq.string, aa.names)
n <- gen.n(seq.string, aa.names)
scuo <- gen.scuo(seq.string, aa.names)

# Convert to list format.
reu13.list <- convert.reu13.df.to.list(reu13.df)
y.list <- convert.y.to.list(y)
n.list <- convert.n.to.list(n)

## End(Not run)
```

Initial Generic Functions

Initial Generic Functions of Codon Usage Bias Fits

Description

Initial generic functions for model fitting/approximation/prediction of **cubfits**.

Usage

```
init.function(model = .CF.CT$model[1],
              type.p = .CF.CT$type.p[1],
              type.Phi = .CF.CT$type.Phi[1],
              model.Phi = .CF.CT$model.Phi[1],
```

```
                    init.Phi = .CF.CT$init.Phi[1],
                    init.fit = .CF.CT$init.fit[1],
                    parallel = .CF.CT$parallel[1],
                    adaptive = .CF.CT$adaptive[1])
```

Arguments

`model`	main fitted model.
`type.p`	proposal method for hyper-parameters.
`type.Phi`	proposal method for Phi (true expression values).
`model.Phi`	prior of Phi.
`init.Phi`	initial methods for Phi.
`init.fit`	how is coefficient initialed in `vglm()` of **VGAM**.
`parallel`	parallel functions.
`adaptive`	method for adaptive MCMC.

Details

This function mainly takes the options, find the according generic functions, and assign those functions to `.cubfitsEnv`. Those generic functions can be executed accordingly later within functions for MCMC or multinomial logistic regression such as `cubfits()`, `cubappr()`, and `cubpred()`. By default, those options are provided by `.CF.CT` which also leaves rooms for extensions of more complicated models and further optimizations.

It is supposed to call this function before running any MCMC or multinomial logistic regression. This function may affect `cubfits()`, `cubpred()`, `cubappr()`, `estimatePhi()`, and `fitMultinom()`.

- `model` is the main fitting model, currently only `roc` is fully supported.
- `type.p` is for proposing hyper-parameters in Gibb sampler. Currently, `lognormal_fix` is suggested where mean 1 is fixed for log normal distribution. Conjugated prior and flat prior exist and are easily available in this step
- `type.Phi` is for proposing Phi (expression values) in the random walk chain updates. Only, `RW_Norm` is supported. Usually, the acceptance ratio can be adapted within 25% and 50% controlled by `.CF.AC` if `adaptive = simple`.
- `model.Phi` is for the distribution of Phi. Typically, log normal distribution `lognormal` is assumed.
- `init.Phi` is a way to initial Phi. Posterior mean `PM` is recommended which avoid boundary values.
- `init.fit` is a way of initial coefficients to fit mutation and selection coefficients ($\log \mu$ and Δt or ω) in `vglm()`. Option `current` means the `b` (log(mu) and Delta.t) of current MCMC iteration is the initial values, while `random` means `vglm()` provides the initial values.
- `parallel` is a way of parallel methods to speed up code. `lapply` means `lapply()` is used and no parallel; `mclapply` means `mclapply()` of **parallel** is used and good for shared memory machines; `task.pull` means `task.pull()` of **pbdMPI** is used and good for heterogeneous machines; `pbdLapply` means `pbdLapply()` of **pbdMPI** is used and good for homogeneous machines. Among those, `task.pull` is tested thoroughly and is the most reliable and efficient method.

- adaptive is a way for adaptive MCMC that propose better mixing distributions for random walks of Phi. The simple method is suggested and only the proposal distribution of Phi (type.Phi = RW_Norm) is adjusted gradually.

Value

Return an invisible object which is a list contain all generic functions according to the input options. All functions are also assigned in the .cubfitsEnv for later evaluations called by MCMC or multinomial logistic regression.

Note

Note that all options are taken default values from the global control object .CF.CT, so one can utilize/alter the object's values to adjust those affected functions.

Note that phi.Obs should be scaled to mean 1 before applying to MCMC.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

.CF.CT, .CF.CT, cubfits(), cubpred(), and cubappr().

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

# Convert data.
reu13.list <- convert.reu13.df.to.list(ex.test$reu13.df)
y.list <- convert.y.to.list(ex.test$y)
n.list <- convert.n.to.list(ex.test$n)

# Get phi.pred.Init
init.function(model = "roc")
fitlist <- fitMultinom(ex.train$reu13.df, ex.train$phi.Obs, ex.train$y,
                       ex.train$n)
phi.pred.Init <- estimatePhi(fitlist, reu13.list, y.list, n.list,
                         E.Phi = median(ex.test$phi.Obs),
                         lower.optim = min(ex.test$phi.Obs) * 0.9,
                         upper.optim = max(ex.test$phi.Obs) * 1.1)

## End(Not run)
```

`Input and Output Utility`
Input and Output Utility

Description

These utility functions read and write data of FASTA and phi.df formats.

Usage

```
read.seq(file.name, forceDNAtolower = FALSE, convertDNAtoupper = TRUE)
write.seq(seq.data, file.name)

read.phi.df(file.name, header = TRUE, sep = "\t", quote = "")
write.phi.df(phi.df, file.name)

get.expath(file.name, path.root = "./ex_data/", pkg = "cubfits")
```

Arguments

`file.name`	a file name to read or write.
`forceDNAtolower`	an option passed to `read.fasta()` of **seqinr** package.
`convertDNAtoupper`	force everything in upper case.
`header`	an option passed to `read.table()`.
`sep`	an option passed to `read.table()`.
`quote`	an option passed to `read.table()`.
`seq.data`	a `seq.data` object.
`phi.df`	a `phi.df` object.
`path.root`	root path for the file name relatively to the pkg.
`pkg`	package name for the path of root.

Details

`read.seq()` and `write.seq()` typically read and write FASTA files (DNA ORFs or sequences).

`read.phi.df()` and `write.phi.df()` typically read and write phi.df files (expression values of ORFs or sequences).

`get.expath()` is only for demonstration returning a full path to the file.

Value

`read.seq()` returns an object of `seq.data` format which can be converted to `seq.string` format later via `convert.seq.data.to.string()`.

`read.phi.df()` returns an object of `phi.df` format which contains expression values.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

convert.seq.data.to.string().

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

seq.data <- read.seq(get.expath("seq_200.fasta"))
phi.df <- read.phi.df(get.expath("phi_200.tsv"))
aa.names <- c("A", "C", "D")

# Read in from FASTA file.
seq.string <- convert.seq.data.to.string(seq.data)

## End(Not run)
```

Mixed Normal Optimization

Mixed Normal Optimization

Description

Constrained optimization for mixed normal in 1D and typically for 2 components.

Usage

```
mixnormerr.optim(X, K = 2, param = NULL)
dmixnormerr(x, param)
```

Arguments

X	a gene expression data matrix of dimension N * R which has N genes and R replicates.
K	number of components to fit.
x	vector of quantiles.
param	parameters of mixnormerr, typically the element param of the mixnormerr.optim() returning object.

Details

The function `mixnormerr.optim()` maximizes likelihood using `constrOptim()` based on the gene expression data `X` (usually in log scale) for `N` genes and `R` replicates (`NA` is allowed). The likelihood of each gene expression is a `K = 2` component mixed normal distribution ($\sum_k p_k N(mu_k, \sigma_k^2 + \sigma_e^2)$) with measurement errors of the replicates ($N(0, \sigma_e^2)$).

The `sigma_k^2` is as the error of random component and the `sigma_e^2` is as the error of fixed component. Both are within a mixture model of two normal distributions.

The function `dmixnormerr()` computes the density of the mixed normal distribution.

`param` is a parameter list and contains five elements: `K` for number of components, `prop` for proportions, `mu` for centers of components, `sigma2` for variance of components, and `sigma2.e` for variance of measurement errors.

Value

`mixnormerr.optim()` returns a list containing three main elements `param` is the final results (MLEs), `param.start` is the starting parameters, and `optim.ret` is the original returns of `constrOptim()`.

Note

This function is limited for small `K`. An equivalent EM algorithm should be done in a more stable way for large `K`.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

`print.mixnormerr()`, `simu.mixnormerr()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

### Get individual of phi.Obs.
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))))
phi.Obs.all <- yassour[, -1] / sum(GM) * 15000
phi.Obs.all[phi.Obs.all == 0] <- NA

### Run optimization.
X <- log(as.matrix(phi.Obs.all))
param.init <- list(K = 2, prop = c(0.95, 0.05), mu = c(-0.59, 3.11),
                   sigma2 = c(1.40, 0.59), sigma2.e = 0.03)
ret <- mixnormerr.optim(X, K = 2, param = param.init)
```

```
print(ret)

## End(Not run)
```

Plotbin *Plot Binning Results*

Description

Plot binning results to visualize the effects of mutation and selection along with expression levels empirically.

Usage

```
prop.bin.roc(reu13.df, phi.Obs = NULL, nclass = 20, bin.class = NULL,
             weightedCenters = TRUE, logBins = FALSE)

plotbin(ret.bin, ret.model = NULL, main = NULL,
        xlab = "Production Rate (log10)", ylab = "Proportion",
        xlim = NULL, lty = 1, x.log10 = TRUE, stderr = FALSE, ...)
```

Arguments

reu13.df	a reu13.df object.
phi.Obs	a phi.Obs object.
nclass	number of binning classes across the range of phi.Obs.
bin.class	binning proportion, e.g. c(0, seq(0.05, 0.95, length = nclass), 1).
ret.bin	binning results from prop.bin.roc().
weightedCenters	if centers are weighted.
logBins	if use log scale for bin.
ret.model	model results from prop.model.roc().
main	an option passed to plot().
xlab	an option passed to plot().
ylab	an option passed to plot().
xlim	range of X-axis.
lty	line type if ret.model is provided.
x.log10	log10() transformation of X-axis.
stderr	plot stand error instead of stand deviation.
...	options passed to plot().

Details

The function `plotbin()` plots the binning results `ret.bin` returned from `prop.bin.roc()`. Fitted curves may be added if `ret.model` is provided which can be obtained from `prop.model.roc()`.

`plotaddmodel()` can append model later if `ret.model` is not provided to `plotbin()`.

Currently, only ROC model is supported. Colors are controlled by `.CF.PT`.

Value

A binning plot is drawn.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

`plotmodel()` and `prop.model.roc()`.

Examples

```
## Not run:
demo(plotbin, 'cubfits', ask = F, echo = F)

## End(Not run)
```

Plotmodel *Plot Fitted Models*

Description

Plot model results to visualize the effects of mutation and selection along with expression levels. The model can be fitted by MCMC or multinomial logistic regression.

Usage

```
prop.model.roc(b.Init, phi.Obs.lim = c(0.01, 10), phi.Obs.scale = 1,
               nclass = 40, x.log10 = TRUE)

plotmodel(ret.model, main = NULL,
          xlab = "Production Rate (log10)", ylab = "Proportion",
          xlim = NULL, lty = 1, x.log10 = TRUE, ...)

plotaddmodel(ret.model, lty, u.codon = NULL, color = NULL,
             x.log10 = TRUE)
```

Arguments

b.Init	a b object.
phi.Obs.lim	range of phi.Obs.
phi.Obs.scale	optional scaling factor.
nclass	number of binning classes across the range of phi.Obs.
x.log10	log10() transformation of X-axis.
ret.model	model results from prop.model.roc().
main	an option passed to plot().
xlab	an option passed to plot().
ylab	an option passed to plot().
xlim	range of X-axis.
lty	line type.
u.codon	unique synonymous codon names.
color	a color vector for unique codon, typically returns of the internal function get.color().
...	options passed to plot().

Details

The function plotmodel() plots the fitted curves obtained from prop.model.roc().

The function plotaddmodel() can append model curves to a binning plot provided unique synonymous codons and colors are given. This function is nearly for an internal call within plotmodel(), but is exported and useful for workflow.

Currently, only ROC model is supported. Colors are controlled by .CF.PT.

Value

A fitted curve plot is drawn.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

plotbin(), prop.bin.roc(), and prop.model.roc().

Examples

```
## Not run:
demo(plotbin, 'cubfits', ask = F, echo = F)

## End(Not run)
```

Plotprxy *Predictive X-Y Plot*

Description

This utility function provides a basic plot of production rates.

Usage

```
plotprxy(x, y, x.ci = NULL, y.ci = NULL,
         log10.x = TRUE, log10.y = TRUE,
         add.lm = TRUE, add.one.to.one = TRUE, weights = NULL,
         add.legend = TRUE,
         xlim = NULL, ylim = NULL,
         xlab = "Predicted Production Rate (log10)",
         ylab = "Observed Production Rate (log10)",
         main = NULL)
```

Arguments

x	expression values.
y	expression values, of the same length of x.
x.ci	confidence interval of x, of dimension length{x} * 2, for outliers labeling.
y.ci	confidence interval of y, of dimension length{y} * 2, for outliers labeling.
log10.x	log10() and mean transformation of x axis.
log10.y	log10() and mean transformation of y axis.
add.lm	if add lm() fit.
add.one.to.one	if add one-to-one line.
weights	weights to lm().
add.legend	if add default legend.
xlim	limits of x-axis.
ylim	limits of y-axis.
xlab	an option passed to plot().
ylab	an option passed to plot().
main	an option passed to plot().

Details

As the usual X-Y plot where x and y are expression values.

If add.lm = TRUE and weights are given, then both ordinary and weighted least squares results will be plotted.

Value

A scatter plot with a fitted `lm()` line and R squared value.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

`plotbin()` and `plotmodel()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

y.scuo <- convert.y.to.scuo(ex.train$y)
SCUO <- calc_scuo_values(y.scuo)$SCUO
plotprxy(ex.train$phi.Obs, SCUO)

## End(Not run)
```

Posterior Results of Yassour2009

Posterior Results of Yassour 2009 Yeast Experiment Dataset

Description

Output summarized from MCMC posterior results analyzing Yassour 2009 data.

Usage

```
yassour.PM.fits
yassour.PM.appr
yassour.info
```

Format

These are `list`'s containing several posterior means: `E.Phi` for expected expression, `b.InitList.roc` for parameters, `AA.prob` for proportion of amino acids, `sigmaW` for standard error of measure errors, and `gene.length` for gene length.

Details

yassour.PM.fits and yassour.PM.appr are the MCMC output of with/without observed expression, respectively. Both contain posterior means of expected expressions and coefficient parameters: E.Phi and b.InitList.roc are scaled results such that each MCMC iteration has mean 1 at E.Phi.

yassour.info contains sequences information (Yeast): AA.prob and gene.length are summarized from corresponding genes in the analysis.

Note that some of genes may not have good quality of expression or sequence information, so those genes are dropped from yassour dataset.

References

https://github.com/snoweye/cubfits/

See Also

yassour

Examples

```
## Not run:
str(yassour.PM.fits)
str(yassour.PM.appr)
str(yassour.PM.info)

## End(Not run)
```

Print *Functions for Printing Objects According to Classes*

Description

A Class mixnormerr is declared in **cubfits**, and this is the function to print and summary objects.

Usage

```
## S3 method for class 'mixnormerr'
print(x, digits = max(4, getOption("digits") - 3), ...)
```

Arguments

x	an object with the class attributes.
digits	for printing out numbers.
...	other possible options.

Details

This is an useful function for summarizing and debugging.

Value

The results will cat or print on the STDOUT by default.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

mixnormerr.optim().

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

### Get individual of phi.Obs.
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))))
phi.Obs.all <- yassour[, -1] / sum(GM) * 15000
phi.Obs.all[phi.Obs.all == 0] <- NA

### Run optimization.
X <- log(as.matrix(phi.Obs.all))
param.init <- list(K = 2, prop = c(0.95, 0.05), mu = c(-0.59, 3.11),
                   sigma2 = c(1.40, 0.59), sigma2.e = 0.03)
ret <- mixnormerr.optim(X, K = 2, param = param.init)
print(ret)

## End(Not run)
```

Randomize SCUO Index *Generate Randomized SCUO Index*

Description

Generate randomized SCUO indices in log normal distribution, but provided original unchanged SCUO order.

Usage

```
scuo.random(SCUO, phi.Obs = NULL, meanlog = .CF.PARAM$phi.meanlog,
            sdlog = .CF.PARAM$phi.sdlog)
```

Arguments

SCUO	SCUO index returned from calc_scuo_values().
phi.Obs	optional object of format phi.Obs.
meanlog	mean of log normal distribution.
sdlog	std of log normal distribution.

Details

This function takes SCUO indices (outputs of calc_scuo_values()) computes the rank of them, generates log normal random variables, and replaces SCUO indices by those variables in the same rank orders. Typically, these random variables are used to replace expression values when either no expression is observed or for the purpose of model validation.

If phi.Obs is provided, the mean and std of log(phi.Obs) are used for log normal random variables. Otherwise, menalog and sdlog are used.

The default meanlog and sdlog was estimated from yassour dataset.

Value

A vector of log normal random variables is returned.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

calc_scuo_values(), yassour.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

### example dataset.
y.scuo <- convert.y.to.scuo(ex.train$y)
SCUO <- calc_scuo_values(y.scuo)$SCUO
plotprxy(ex.train$phi.Obs, SCUO)

### yassour dataset.
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))))
phi.Obs <- GM / sum(GM) * 15000
mean(log(phi.Obs))
sd(log(phi.Obs))
ret <- scuo.random(SCUO, meanlog = -0.441473, sdlog = 1.393285)
plotprxy(ret, SCUO)
```

```
## End(Not run)
```

`Rearrangment Utility` *Rearrange Data Structure by ORF Names*

Description

These utility functions rearrange data in the order of ORF names.

Usage

```
rearrange.reu13.df(reu13.df)
rearrange.y(y)
rearrange.n(n)
rearrange.phi.Obs(phi.Obs)
```

Arguments

`reu13.df`	a list of `reu13.df` data frames divided by amino acids.
`y`	a list of `y` data frames divided by amino acids.
`n`	a list of `n` vectors divided by amino acids.
`phi.Obs`	a vector of `phi.Obs` format.

Details

These utility functions take inputs and return ordered outputs. It is necessary to rearrange data in a right order of ORF names which avoids subsetting data frame within MCMC and improve performance.

Value

The outputs are in the same format of inputs except the order of data is sorted by ORF names.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

AllDataFormats, `convert.n.to.list()`, `convert.reu13.df.to.list()`, and `convert.y.to.list()`.

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

reu13.df <- rearrange.reu13.df(ex.train$reu13.df)
y <- rearrange.y(ex.train$y)
n <- rearrange.n(ex.train$n)
phi.Obs <- rearrange.phi.Obs(ex.train$phi.Obs)

## End(Not run)
```

SCUO Index *Function for Synonymous Codon Usage Order (SCUO) Index*

Description

Calculate the Synonymous Codon Usage Order (SCUO) index for each gene. Used as a substitute for expression in cases of without expression measurements.

Usage

```
calc_scuo_values(codon.counts)
```

Arguments

codon.counts	an object of format scuo.

Details

This function computes SCUO index for each gene. Typically, this method is completely based on entropy and information theory to estimate expression values of sequences according to their codon information.

Value

SCUO indices are returned.

Author(s)

Drew Schmidt.

References

https://www.tandfonline.com/doi/abs/10.1080/03081070500502967

Wan X.-F., Zhou J., Xu D. "CodonO: a new informatics method for measuring synonymous codon usage bias within and across genomes" International Journal of General Systems Vol. 35, Iss. 1, 2006.

See Also

scuo.random(), calc_cai_values(), calc_scu_values().

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

y.scuo <- convert.y.to.scuo(ex.train$y)
SCUO <- calc_scuo_values(y.scuo)$SCUO
plotprxy(ex.train$phi.Obs, SCUO, ylab = "SCUO (log10)")

## End(Not run)
```

Selection on Codon Usage

Function for Selection on Codon Usage (SCU)

Description

Calculate the average translational selection per transcript include mSCU and SCU (if gene expression is provided) for each gene.

Usage

```
calc_scu_values(b, y.list, phi.Obs = NULL)
```

Arguments

b	an object of format b.
y.list	an object of format y.list.
phi.Obs	an object of format phi.Obs, for SCU only.

Details

This function computes SCU and mSCU for each gene. Typically, this method is completely based on estimated parameters of mutation and selection such as outputs of MCMC or fitMultinom().

Value

A list with two named elements SCU and mSCU are returned.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

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

See Also

`calc_scuo_values()`, `calc_cai_values()`.

Examples

```
## Not run:
library(cubfits, quietly = TRUE)

b <- b.Init$roc
phi.Obs <- ex.train$phi.Obs
y <- ex.train$y
y.list <- convert.y.to.list(y)
mSCU <- calc_scu_values(b, y.list, phi.Obs)$mSCU
plot(mSCU, log10(phi.Obs), main = "Expression vs mSCU",
     xlab = "mSCU", ylab = "Expression (log10)")

### Compare with CAI with weights seqinr::cubtab$sc.
library(seqinr, quietly = TRUE)
w <- caitab$sc
names(w) <- codon.low2up(rownames(caitab))
CAI <- calc_cai_values(y, y.list, w = w)$CAI

plot(mSCU, CAI, main = "CAI vs mSCU",
     xlab = "mSCU", ylab = "CAI")

## End(Not run)
```

Simulation Tool *Simulate ORFs and Expression Data*

Description

These utility functions generate data for simulation studies including fake ORFs and expression values.

Usage

```
simu.orf(n, b.Init, phi.Obs = NULL, AA.prob = NULL, orf.length = NULL,
         orf.names = NULL, model = .CF.CT$model)
simu.phi.Obs(Phi, sigmaW.lim = 1, bias.Phi = 0)
simu.mixnormerr(n, param)
```

Arguments

n	number of ORFs or sequences.
b.Init	parameters of mutation and selection of format b.
phi.Obs	an object of format phi.Obs.
AA.prob	proportion of amino acids.
orf.length	lengths of ORFs.
orf.names	names of ORFs.
model	model to be simulated.
Phi	expression values (potentially true expression).
sigmaW.lim	std of measurement errors (between Phi and phi.Obs).
bias.Phi	bias (in log scale) for observed phi.
param	as in dmixnormerr()

Details

simu.orf() generates ORFs or sequences based on the b.Init and phi.Obs.

If phi.Obs is omitted, then standard log normal random variables are instead).

If AA.prob is omitted, then uniform proportion is assigned.

If orf.length is omitted, then 10 to 20 codons are randomly assigned.

If orf.names is omitted, then "ORF1" to "ORFn" are assigned.

simu.phi.Obs() generates phi.Obs by adding normal random errors to Phi, and errors have mean 0 and standard deviation sigmaW.lim.

simu.mixnormerr() generates Phi according to the param, and adds normal random errors to Phi.

Value

simu.orf() returns a list of format seq.data.

simu.phi.Obs() returns a vector of format phi.Obs.

simu.mixnormerr() returns a list contains three vectors of length n: one for expected gene expression Phi, one for observed gene expression phi.Obs, and one for the component id id.K.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

read.seq(), read.phi.df(), write.seq(), write.phi.df(), and mixnormerr.optim().

Examples

```
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

# Generate sequences.
da.roc <- simu.orf(length(ex.train$phi.Obs), b.Init$roc,
                   phi.Obs = ex.train$phi.Obs, model = "roc")
names(da.roc) <- names(ex.train$phi.Obs)
write.fasta(da.roc, names(da.roc), "toy_roc.fasta")

## End(Not run)
```

`Yassour2009` *Yassour 2009 Yeast Experiment Dataset*

Description

Experiments and data are obtained from Yassour et. al. (2009).

Usage

```
yassour
```

Format

A `data.frame` contains 6303 rows and 5 columns: `ORF` is for gene names in character, and `YPD0.1`, `YPD0.2`, `YPD15.1`, and `YPD15.2` are gene expressions in positive double corresponding to 4 controlled Yeast experiments.

Details

The original data are available as the URL of the section of Source next. As the section of Examples next, data are selected from `SD3.xls` and reordered by `ORF`.

For further analysis, the Examples section also provides how to convert them to `phi.Obs` values either in geometric means or individually.

Source

https://www.pnas.org/content/early/2009/02/10/0812841106

https://www.pnas.org/highwire/filestream/598612/field_highwire_adjunct_files/3/SD3.xls

Yassour M, Kaplan T, Fraser HB, Levin JZ, Pfiffner J, Adiconis X, Schroth G, Luo S, Khrebtukova I, Gnirke A, Nusbaum C, Thompson DA, Friedman N, Regev A. (2009) "Ab initio construction of a eukaryotic transcriptome by massively parallel mRNA sequencing." Proc Natl Acad Sci USA 106(9):3264-9. [PMID:19208812]

References

Wallace E.W.J., Airoldi E.M., and Drummond D.A. “Estimating Selection on Synonymous Codon Usage from Noisy Experimental Data” Mol Biol Evol (2013) 30(6):1438–1453.

Examples

```
## Not run:
### SD3.xls is available from the URL provided in the References.
da <- read.table("SD3.xls", header = TRUE, sep = "\t", quote = "",
                 stringsAsFactors = FALSE)

### Select ORF, YPD0.1, YPD0.2, YPD15.1, YPD15.2.
da <- da[, c(1, 8, 9, 10, 11)]
colnames(da) <- c("ORF", "YPD0.1", "YPD0.2", "YPD15.1", "YPD15.2")

### Drop inappropriate values (NaN, NA, Inf, -Inf, and 0).
tmp <- da[, 2:5]
id.tmp <- rowSums(is.finite(as.matrix(tmp)) & tmp != 0) >= 3
tmp <- da[id.tmp, 1:5]
yassour <- tmp[order(tmp$ORF),]    # cubfits::yassour

### Get geometric mean of phi.Obs and scaling similar to Wallace (2013).
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))))
phi.Obs <- GM / sum(GM) * 15000

### Get individual of phi.Obs.
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))))
phi.Obs.all <- yassour[, -1] / sum(GM) * 15000
phi.Obs.all[phi.Obs.all == 0] <- NA

## End(Not run)
```

Index

∗ **cedric**
Cedric Convergence Utilities, 6
Cedric IO Utilities, 7
Cedric Plot Utilities, 8
∗ **dataformats**
Data Formats, 23
∗ **datasets**
Controls, 11
Datasets, 24
Posterior Results of Yassour2009, 40
Yassour2009, 49
∗ **main function**
CUB Model Approximation, 16
CUB Model Fits, 18
CUB Model Prediction, 20
∗ **package**
cubfits-package, 2
∗ **plotting**
Plotbin, 36
Plotmodel, 37
Plotprxy, 39
∗ **summary**
Print, 41
∗ **tool**
Codon Adaptation Index, 10
Estimate Phi, 26
Fit Multinomial, 27
Initial Generic Functions, 30
Randomize SCUO Index, 42
SCUO Index, 45
Selection on Codon Usage, 46
Simulation Tool, 47
∗ **utility**
Asymmetric Laplace Distribution, 4
Coverting Utility, 14
Generating Utility, 29
Input and Output Utility, 33
Mixed Normal Optimization, 34
Rearrangment Utility, 44
.CF.AC, *31*
.CF.AC *(*Controls*)*, 11
.CF.CONF *(*Controls*)*, 11
.CF.CT, *31*, *32*
.CF.CT *(*Controls*)*, 11
.CF.DP, *17*, *19*, *22*
.CF.DP *(*Controls*)*, 11
.CF.GV *(*Controls*)*, 11
.CF.OP *(*Controls*)*, 11
.CF.PARAM *(*Controls*)*, 11
.CF.PT, *37*, *38*
.CF.PT *(*Controls*)*, 11
.CO.CT *(*Controls*)*, 11
.cubfitsEnv, *31*, *32*
.cubfitsEnv *(*Controls*)*, 11

AllDataFormats, *16*, *29*, *30*, *44*
AllDataFormats *(*Data Formats*)*, 23
asl.optim *(*Asymmetric Laplace Distribution*)*, 4
Asymmetric Laplace Distribution, 4

b, *15*, *17*, *19*, *21*, *22*, *25–28*, *31*, *38*, *46*, *48*
b *(*Data Formats*)*, 23
b.Init *(*Datasets*)*, 24
bVec, *15*
bVec *(*Data Formats*)*, 23

calc_cai_values, *46*, *47*
calc_cai_values *(*Codon Adaptation Index*)*, 10
calc_scu_values, *10*, *46*
calc_scu_values *(*Selection on Codon Usage*)*, 46
calc_scuo_values, *10*, *43*, *47*
calc_scuo_values *(*SCUO Index*)*, 45
Cedric Convergence Utilities, 6
Cedric IO Utilities, 7
Cedric Plot Utilities, 8

Codon Adaptation Index, 10
codon.low2up (Coverting Utility), 14
codon.up2low (Coverting Utility), 14
Controls, 11
convert.b.to.bVec (Coverting Utility), 14
convert.bVec.to.b (Coverting Utility), 14
convert.n.to.list, *44*
convert.n.to.list (Coverting Utility), 14
convert.reu13.df.to.list, *44*
convert.reu13.df.to.list (Coverting Utility), 14
convert.seq.data.to.string, *30*, *33*, *34*
convert.seq.data.to.string (Coverting Utility), 14
convert.y.to.list, *44*
convert.y.to.list (Coverting Utility), 14
convert.y.to.scuo (Coverting Utility), 14
Coverting Utility, 14
CUB Model Approximation, 16
CUB Model Fits, 18
CUB Model Prediction, 20
cubappr, *3*, *14*, *20*, *22*, *25*, *31*, *32*
cubappr (CUB Model Approximation), 16
cubfits, *3*, *14*, *18*, *22*, *25*, *31*, *32*
cubfits (CUB Model Fits), 18
cubfits-package, 2
cubmultichain (Cedric Convergence Utilities), 6
cubpred, *3*, *14*, *18*, *20*, *25*, *31*, *32*
cubpred (CUB Model Prediction), 20
cubsinglechain (Cedric Convergence Utilities), 6

dasl (Asymmetric Laplace Distribution), 4
dasla (Asymmetric Laplace Distribution), 4
Data Formats, 23
DataConverting, *18*, *20*, *22*
DataConverting (Coverting Utility), 14
DataGenerating (Generating Utility), 29
DataIO, *18*, *20*, *22*
DataIO (Input and Output Utility), 33
Datasets, 24
dmixnormerr, *48*
dmixnormerr (Mixed Normal Optimization), 34
dna.low2up (Coverting Utility), 14
dna.up2low (Coverting Utility), 14

Estimate Phi, 26
estimatePhi, *28*, *31*
estimatePhi (Estimate Phi), 26
ex.test (Datasets), 24
ex.train (Datasets), 24

Fit Multinomial, 27
fitMultinom, *27*, *31*, *46*
fitMultinom (Fit Multinomial), 27

gen.n (Generating Utility), 29
gen.phi.Obs (Generating Utility), 29
gen.reu13.df (Generating Utility), 29
gen.reu13.list (Generating Utility), 29
gen.scuo (Generating Utility), 29
gen.y (Generating Utility), 29
Generating Utility, 29
get.expath (Input and Output Utility), 33

init.function, *3*, *12*, *14*, *25–28*
init.function (Initial Generic Functions), 30
Initial Generic Functions, 30
Input and Output Utility, 33
isConverged (Cedric Convergence Utilities), 6

Mixed Normal Optimization, 34
mixnormerr.optim, *14*, *42*, *48*
mixnormerr.optim (Mixed Normal Optimization), 34

n, *15*, *17*, *19*, *21*, *25*, *27*, *29*, *44*
n (Data Formats), 23
n.list, *26*
normalizeDataSet (Cedric IO Utilities), 7

pasl (Asymmetric Laplace Distribution), 4
pasla (Asymmetric Laplace Distribution), 4
phi.df, *29*, *33*

phi.df (Data Formats), 23
phi.Obs, *17*, *19*, *21*, *25*, *27*, *36*, *43*, *44*, *46*, *48*
phi.Obs (Data Formats), 23
plotaddmodel, *37*
plotaddmodel (Plotmodel), 37
Plotbin, 36
plotbin, *38*, *40*
plotbin (Plotbin), 36
plotCUB (Cedric Plot Utilities), 8
plotExpectedPhiTrace (Cedric Plot Utilities), 8
Plotmodel, 37
plotmodel, *37*, *40*
plotmodel (Plotmodel), 37
Plotprxy, 39
plotprxy (Plotprxy), 39
plotPTraces (Cedric Plot Utilities), 8
plotTraces (Cedric Plot Utilities), 8
Posterior Results of Yassour2009, 40
Print, 41
print.mixnormerr, *35*
print.mixnormerr (Print), 41
prop.bin.roc, *36*, *38*
prop.bin.roc (Plotbin), 36
prop.model.roc, *36–38*
prop.model.roc (Plotmodel), 37

qasl (Asymmetric Laplace Distribution), 4
qasla (Asymmetric Laplace Distribution), 4

Randomize SCUO Index, 42
rasl (Asymmetric Laplace Distribution), 4
rasla (Asymmetric Laplace Distribution), 4
read.phi.df, *29*, *30*, *48*
read.phi.df (Input and Output Utility), 33
read.seq, *16*, *30*, *48*
read.seq (Input and Output Utility), 33
readGenome (Cedric IO Utilities), 7
rearrange.n, *16*
rearrange.n (Rearrangment Utility), 44
rearrange.phi.Obs (Rearrangment Utility), 44
rearrange.reu13.df, *16*
rearrange.reu13.df (Rearrangment Utility), 44
rearrange.y, *16*
rearrange.y (Rearrangment Utility), 44
Rearrangment Utility, 44
reu13.df, *15*, *17*, *19*, *21*, *25*, *27*, *29*, *36*, *44*
reu13.df (Data Formats), 23
reu13.list, *26*
reu13.list (Data Formats), 23

scuo, *15*, *45*
scuo (Data Formats), 23
SCUO Index, 45
scuo.random, *46*
scuo.random (Randomize SCUO Index), 42
Selection on Codon Usage, 46
seq.data, *15*, *33*, *48*
seq.data (Data Formats), 23
seq.string, *15*, *29*, *33*
seq.string (Data Formats), 23
simu.mixnormerr, *35*
simu.mixnormerr (Simulation Tool), 47
simu.orf (Simulation Tool), 47
simu.phi.Obs (Simulation Tool), 47
Simulation Tool, 47

write.phi.df, *48*
write.phi.df (Input and Output Utility), 33
write.seq, *48*
write.seq (Input and Output Utility), 33

y, *10*, *15*, *17*, *19*, *21*, *25*, *27*, *29*, *44*
y (Data Formats), 23
y.list, *10*, *26*, *46*
yassour, *41*
yassour (Yassour2009), 49
yassour.info (Posterior Results of Yassour2009), 40
yassour.PM.appr (Posterior Results of Yassour2009), 40
yassour.PM.fits (Posterior Results of Yassour2009), 40
Yassour2009, 49