Package 'dclone'

February 26, 2018

Type Package

Title Data Cloning and MCMC Tools for Maximum Likelihood Methods

Version 2.2-0

Date 2018-02-26

Author Peter Solymos

Maintainer Peter Solymos <solymos@ualberta.ca>

Description Low level functions for implementing
maximum likelihood estimating procedures for
complex models using data cloning and Bayesian
Markov chain Monte Carlo methods
as described in Solymos 2010 (R Journal 2(2):29--37).
Sequential and parallel MCMC support
for 'JAGS', 'WinBUGS' and 'OpenBUGS'.

License GPL-2

Depends R (>= 2.15.1), coda (>= 0.13), parallel, Matrix

Suggests MASS, lattice, R2WinBUGS, R2OpenBUGS, BRugs, rlecuyer

Imports methods, stats, rjags (>= 4-4)

SystemRequirements none, one or more of JAGS (>= 3.0.0), WinBUGS (>= 1.4), OpenBUGS (>= 3.2.2)

URL https://groups.google.com/forum/#!forum/dclone-users, http://datacloning.org

BugReports https://github.com/datacloning/dclone/issues

LazyLoad yes

LazyData true

NeedsCompilation no

Repository CRAN

Date/Publication 2018-02-26 22:04:04 UTC

R topics documented:

dclone-package *Data Cloning*

Description

Low level functions for implementing maximum likelihood estimating procedures for complex models using data cloning and Bayesian Markov chain Monte Carlo methods. Sequential and parallel MCMC support for JAGS, WinBUGS and OpenBUGS.

Main functions include:

- `dclone`, `dcdim`, `dciid`, `dctr`: cloning R objects in various ways.
- `jags.fit`, `bugs.fit`: conveniently fit BUGS models. `jags.parfit` and `bugs.parfit` fits chains on parallel workers.
- `dc.fit`: iterative model fitting by the data cloning algorithm. `dc.parfit` is the parallelized version.
- `dctable`, `dcdiag`: helps evaluating data cloning convergence by descriptive statistics and diagnostic tools. (These are based on e.g. `chisq.diag` and `lambdamax.diag`.)
- `coef.mcmc.list`, `confint.mcmc.list.dc`, `dcsd.mcmc.list`, `quantile.mcmc.list`, `vcov.mcmc.list.dc`, `mcmcapply`, `stack.mcmc.list`: methods for `mcmc.list` objects.
- `write.jags.model`, `clean.jags.model`, `custommodel`: convenient functions for handling BUGS models.
- `jagsModel`, `codaSamples`: basic functions from **rjags** package rewrote to recognize data cloning attributes from data (`parJagsModel`, `parUpdate`, `parCodaSamples` are the parallel versions).

Author(s)

Author: Peter Solymos

Maintainer: Peter Solymos, <solymos@ualberta.ca>

References

Forum: https://groups.google.com/forum/#!forum/dclone-users

Issues: https://github.com/datacloning/dcmle/issues

Data cloning website: http://datacloning.org

Solymos, P., 2010. dclone: Data Cloning in R. *The R Journal* **2(2)**, 29–37. URL: https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Solymos.pdf

Lele, S.R., B. Dennis and F. Lutscher, 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods. *Ecology Letters* **10**, 551–563.

Lele, S. R., K. Nadeem and B. Schmuland, 2010. Estimability and likelihood inference for generalized linear mixed models using data cloning. *Journal of the American Statistical Association* **105**, 1617–1625.

.dcFit *Internal function for iterative model fitting with data cloning*

Description

This is the workhorse for dc.fit and dc.parfit.

Usage

```
.dcFit(data, params, model, inits, n.clones,
    multiply = NULL, unchanged = NULL,
    update = NULL, updatefun = NULL, initsfun = NULL,
    flavour = c("jags", "bugs"),
    n.chains=3, cl = NULL, parchains = FALSE,
    return.all=FALSE, ...)
```

Arguments

data	A named list (or environment) containing the data.
params	Character vector of parameters to be sampled. It can be a list of 2 vectors, 1st element is used as parameters to monitor, the 2nd is used as parameters to use in calculating the data cloning diagnostics.
model	Character string (name of the model file), a function containing the model, or a custommodel object (see Examples).
inits	Optional specification of initial values in the form of a list or a function (see Initialization at jags.model). If missing, will be treated as NULL and initial values will be generated automatically.
n.clones	An integer vector containing the numbers of clones to use iteratively.
multiply	Numeric or character index for list element(s) in the data argument to be multiplied by the number of clones instead of repetitions.
unchanged	Numeric or character index for list element(s) in the data argument to be left unchanged.
update	Numeric or character index for list element(s) in the data argument that has to be updated by updatefun in each iterations. This usually is for making priors more informative, and enhancing convergence. See Details and Examples.
updatefun	A function to use for updating data[[update]]. It should take an 'mcmc.list' object as 1st argument, 2nd argument can be the number of clones. See Details and Examples.
initsfun	A function to use for generating initial values, inits are updated by the object returned by this function from the second iteration. If initial values are not dependent on the previous iteration, this should be NULL, otherwise, it should take an 'mcmc.list' object as 1st argument, 2nd argument can be the number of clones. This feature is useful if latent nodes are provided in inits so it also requires to be cloned for subsequent iterations. See Details and Examples.

flavour	If "jags", the function jags.fit is called. If "bugs", the function bugs.fit is called.
n.chains	Number of chains to generate.
cl	A cluster object created by makeCluster, or an integer, see parDosa and evalParallelArgument.
parchains	Logical, whether parallel chains should be run.
return.all	Logical. If TRUE, all the MCMC list objects corresponding to the sequence n.clones are returned for further inspection (this only works with partype = "parchains"). Otherwise only the MCMC list corresponding to highest number of clones is returned with summary statistics for the rest.
...	Other values supplied to jags.fit, or bugs.fit, depending on the flavour argument.

Value

An object inheriting from the class 'mcmc.list'.

Author(s)

Peter Solymos, <solymos@ualberta.ca>, implementation is based on many discussions with Khuram Nadeem and Subhash Lele.

See Also

dc.fit, dc.parfit

---

bugs.fit *Fit BUGS models with cloned data*

Description

Convenient functions designed to work well with cloned data arguments and WinBUGS and OpenBUGS.

Usage

```
bugs.fit(data, params, model, inits = NULL, n.chains = 3,
    format = c("mcmc.list", "bugs"),
    program = c("winbugs", "openbugs", "brugs"),
    seed, ...)
## S3 method for class 'bugs'
as.mcmc.list(x, ...)
```

Arguments

data	A list (or environment) containing the data.
params	Character vector of parameters to be sampled.
model	Character string (name of the model file), a function containing the model, or a custommodel object (see Examples).
inits	Optional specification of initial values in the form of a list or a function. If NULL, initial values will be generated automatically.
n.chains	number of Markov chains.
format	Required output format.
program	The program to use, not case sensitive. winbugs calls the function bugs from package **R2WinBUGS**, openbugs calls the function bugs from package **R2OpenBUGS** (this has changed since **dclone** version 1.8-1, this is now the preferred OpenBUGS interface). brugs calls the function openbugs from package **R2WinBUGS** and requires the CRAN package **BRugs** (this is provided for back compatibility purposes, but gives a warning because it is not the preferred interface to R2OpenBUGS).
seed	Random seed (bugs.seed argument for bugs in package **R2WinBUGS** or bugs in package **R2OpenBUGS**, seed argument for openbugs). It takes the corresponding default values (NULL or 1) when missing.
x	A fitted 'bugs' object.
...	Further arguments of the bugs function, except for codaPkg are passed also, most notably the ones to set up burn-in, thin, etc. (see Details).

Value

By default, an mcmc.list object. If data cloning is used via the data argument, summary returns a modified summary containing scaled data cloning standard errors (scaled by sqrt(n.clones)), and R_{hat} values (as returned by gelman.diag).

bugs.fit can return a bugs object if format = "bugs". In this case, summary is not changed, but the number of clones used is attached as attribute and can be retrieved by the function nclones.

The function as.mcmc.list.bugs converts a 'bugs' object into 'mcmc.list' and retrieves data cloning information as well.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Underlying functions: bugs in package **R2WinBUGS**, openbugs in package **R2WinBUGS**, bugs in package **R2OpenBUGS**

Methods: dcsd, confint.mcmc.list.dc, coef.mcmc.list, quantile.mcmc.list, vcov.mcmc.list.dc

Examples

```
## Not run:
## fitting with WinBUGS, bugs example
if (require(R2WinBUGS)) {
data(schools)
dat <- list(J = nrow(schools),
    y = schools$estimate,
    sigma.y = schools$sd)
bugs.model <- function(){
       for (j in 1:J){
         y[j] ~ dnorm (theta[j], tau.y[j])
         theta[j] ~ dnorm (mu.theta, tau.theta)
         tau.y[j] <- pow(sigma.y[j], -2)
       }
       mu.theta ~ dnorm (0.0, 1.0E-6)
       tau.theta <- pow(sigma.theta, -2)
       sigma.theta ~ dunif (0, 1000)
     }
inits <- function(){
    list(theta=rnorm(nrow(schools), 0, 100), mu.theta=rnorm(1, 0, 100),
         sigma.theta=runif(1, 0, 100))
}
param <- c("mu.theta", "sigma.theta")
if (.Platform$OS.type == "windows") {
sim <- bugs.fit(dat, param, bugs.model, inits)
summary(sim)
}
dat2 <- dclone(dat, 2, multiply="J")
if (.Platform$OS.type == "windows") {
sim2 <- bugs.fit(dat2, param, bugs.model,
    program="winbugs", n.iter=2000, n.thin=1)
summary(sim2)
}
}
if (require(BRugs)) {
## fitting the model with OpenBUGS
## using the less preferred BRugs interface
sim3 <- bugs.fit(dat2, param, bugs.model,
    program="brugs", n.iter=2000, n.thin=1)
summary(sim3)
}
if (require(R2OpenBUGS)) {
## fitting the model with OpenBUGS
## using the preferred R2OpenBUGS interface
sim4 <- bugs.fit(dat2, param, bugs.model,
    program="openbugs", n.iter=2000, n.thin=1)
summary(sim4)
}
if (require(rjags)) {
## fitting the model with JAGS
sim5 <- jags.fit(dat2, param, bugs.model)
summary(sim5)
```

```
}

## End(Not run)
```

`bugs.parfit` *Parallel computing with WinBUGS/OpenBUGS*

Description

Does the same job as `bugs.fit`, but parallel chains are run on parallel workers, thus computations can be faster (up to 1/`n.chains`) for long MCMC runs.

Usage

```
bugs.parfit(cl, data, params, model, inits=NULL, n.chains = 3,
    seed, program=c("winbugs", "openbugs", "brugs"), ...)
```

Arguments

`cl`	A cluster object created by `makeCluster`, or an integer, see `parDosa` and `evalParallelArgument`.
`data`	A named list or environment containing the data. If an environment, `data` is coerced into a list.
`params`	Character vector of parameters to be sampled.
`model`	Character string (name of the model file), a function containing the model, or a or `custommodel` object (see Examples).
`inits`	Specification of initial values in the form of a list or a function, can be missing. If this is a function and using 'snow' type cluster as `cl`, the function must be self containing, i.e. not having references to R objects outside of the function, or the objects should be exported with `clusterExport` before calling `bugs.parfit`. Forking type parallelism does not require such attention.
`n.chains`	Number of chains to generate, must be higher than 1. Ideally, this is equal to the number of parallel workers in the cluster.
`seed`	Vector of random seeds, must have `n.chains` unique values. See Details.
`program`	The program to use, not case sensitive. See `bugs.fit`.
`...`	Other arguments passed to `bugs.fit`.

Details

Chains are run on parallel workers, and the results are combined in the end.

The seed must be supplied, as it is the user's responsibility to make sure that pseudo random sequences do not seriously overlap.

The WinBUGS implementation is quite unsafe from this regard, because the pseudo-random number generator used by WinBUGS generates a finite (albeit very long) sequence of distinct numbers, which would eventually be repeated if the sampler were run for a sufficiently long time.

Thus it's usage must be discouraged. That is the reason for the warning that is issued when `program = "winbugs"`.

OpenBUGS (starting from version 3.2.2) implemented a system where internal state of the pseudo random number generator can be set to one of 14 predefined states (seed values in `1:14`). Each predefined state is 10^12 draws apart to avoid overlap in pseudo random number sequences.

Note: the default setting `working.directory = NULL` cannot be changed when running parallel chains with `bugs.parfit` because the multiple instances would try to read/write the same directory.

Value

An `mcmc.list` object.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Sequential version: `bugs.fit`

Examples

```
## Not run:
## fitting with WinBUGS, bugs example
if (require(R2WinBUGS)) {
data(schools)
dat <- list(J = nrow(schools),
    y = schools$estimate,
    sigma.y = schools$sd)
bugs.model <- function(){
       for (j in 1:J){
         y[j] ~ dnorm (theta[j], tau.y[j])
         theta[j] ~ dnorm (mu.theta, tau.theta)
         tau.y[j] <- pow(sigma.y[j], -2)
       }
       mu.theta ~ dnorm (0.0, 1.0E-6)
       tau.theta <- pow(sigma.theta, -2)
       sigma.theta ~ dunif (0, 1000)
     }
param <- c("mu.theta", "sigma.theta")
SEED <- floor(runif(3, 100000, 999999))
cl <- makePSOCKcluster(3)
if (.Platform$OS.type == "windows") {
sim <- bugs.parfit(cl, dat, param, bugs.model, seed=SEED)
summary(sim)
}
dat2 <- dclone(dat, 2, multiply="J")
if (.Platform$OS.type == "windows") {
sim2 <- bugs.parfit(cl, dat2, param, bugs.model,
    program="winbugs", n.iter=2000, n.thin=1, seed=SEED)
summary(sim2)
```

```
}
}
if (require(BRugs)) {
## fitting the model with OpenBUGS
## using the less preferred BRugs interface
sim3 <- bugs.parfit(cl, dat2, param, bugs.model,
    program="brugs", n.iter=2000, n.thin=1, seed=1:3)
summary(sim3)
}
if (require(R2OpenBUGS)) {
## fitting the model with OpenBUGS
## using the preferred R2OpenBUGS interface
sim4 <- bugs.parfit(cl, dat2, param, bugs.model,
    program="openbugs", n.iter=2000, n.thin=1, seed=1:3)
summary(sim4)
}
stopCluster(cl)
## multicore type forking
if (require(R2OpenBUGS) && .Platform$OS.type != "windows") {
sim7 <- bugs.parfit(3, dat2, param, bugs.model,
    program="openbugs", n.iter=2000, n.thin=1, seed=1:3)
summary(sim7)
}

## End(Not run)
```

`clusterSize` *Optimizing the number of workers*

Description

These functions help in optimizing workload for the workers if problems are of different size.

Usage

```
clusterSize(size)
plotClusterSize(n, size,
    balancing = c("none", "load", "size", "both"),
    plot = TRUE, col = NA, xlim = NULL, ylim = NULL,
    main, ...)
```

Arguments

`n`	Number of workers.
`size`	Vector of problem sizes (recycled if needed). The default `1` indicates equality of problem sizes.
`balancing`	Character, type of balancing to perform, one of `c("none", "load", "size", "both")`.
`plot`	Logical, if a plot should be drawn.

col	Color of the polygons for work load pieces.
xlim, ylim	Limits for the x and the y axis, respectively (optional).
main	Title of the plot, can be missing.
...	Other arguments passed to polygon.

Details

These functions help determine the optimal number of workers needed for different sized problems ('size' indicates approximate processing time here). The number of workers needed depends on the type of balancing.

For the description of the balancing types, see parDosa.

Value

clusterSize returns a data frame with approximate processing time as the function of the number of workers (rows, in 1:length(size)) and the type of balancing (c("none", "load", "size", "both")). Approximate processing time is calculated from values in size without taking into account any communication overhead.

plotClusterSize invisibly returns the total processing time needed for a setting given its arguments. As a side effect, a plot is produced (if plot = TRUE).

Author(s)

Peter Solymos, <solymos@ualberta.ca>

Examples

```
## determine the number of workers needed
clusterSize(1:5)
## visually compare balancing options
opar <- par(mfrow=c(2, 2))
plotClusterSize(2,1:5, "none")
plotClusterSize(2,1:5, "load")
plotClusterSize(2,1:5, "size")
plotClusterSize(2,1:5, "both")
par(opar)
```

---

clusterSplitSB *Size balancing*

Description

Functions for size balancing.

Usage

```
clusterSplitSB(cl = NULL, seq, size = 1)
parLapplySB(cl = NULL, x, size = 1, fun, ...)
parLapplySLB(cl = NULL, x, size = 1, fun, ...)
```

Arguments

cl	A cluster object created by makeCluster the the package **parallel**.
x, seq	A vector to split.
fun	A function or character string naming a function.
size	Vector of problem sizes (approximate processing times) corresponding to elements of seq (recycled if needed). The default 1 indicates equality of problem sizes.
...	Other arguments of fun.

Details

clusterSplitSB splits seq into subsets, with respect to size. In size balancing, the problem is re-ordered from largest to smallest, and then subsets are determined by minimizing the total approximate processing time. This splitting is deterministic (reproducible).

parLapplySB and parLapplySLB evaluates fun on elements of x in parallel, similarly to parLapply. parLapplySB uses size balancing (via clusterSplitSB). parLapplySLB uses size and load balancing. This means that the problem is re-ordered from largest to smallest, and then undeterministic load balancing is used (see clusterApplyLB). If size is correct, this is identical to size balancing. This splitting is non-deterministic (might not be reproducible).

Value

clusterSplitSB returns a list of subsets split with respect to size.

parLapplySB and parLapplySLB evaluates fun on elements of x, and return a result corresponding to x. Usually a list with results returned by the cluster.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Related functions without size balancing: clusterSplit, parLapply.

Underlying functions: clusterApply, clusterApplyLB.

Optimizing the number of workers: clusterSize, plotClusterSize.

Examples

```
## Not run:
cl <- makePSOCKcluster(2)
## equal sizes, same as clusterSplit(cl, 1:5)
clusterSplitSB(cl, 1:5)
## different sizes
clusterSplitSB(cl, 1:5, 5:1)
x <- list(1, 2, 3, 4)
parLapplySB(cl, x, function(z) z^2, size=1:4)
stopCluster(cl)

## End(Not run)
```

`codaSamples` *Generate posterior samples in mcmc.list format*

Description

This function sets a trace monitor for all requested nodes, updates the model and coerces the output to a single `mcmc.list` object. This function uses `coda.samples` but keeps track of data cloning information supplied via the `model` argument.

Usage

```
codaSamples(model, variable.names, n.iter, thin = 1, na.rm = TRUE, ...)
```

Arguments

`model`	a jags model object
`variable.names`	a character vector giving the names of variables to be monitored
`n.iter`	number of iterations to monitor
`thin`	thinning interval for monitors
`na.rm`	logical flag that indicates whether variables containing missing values should be omitted. See details in help page of `coda.samples`.
`...`	optional arguments that are passed to the update method for jags model objects

Value

An `mcmc.list` object. An `n.clones` attribute is attached to the object, but unlike in `jags.fit` there is no `updated.model` attribute as it is equivalent to the input jags model object.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

coda.samples, update.jags, jags.model

Parallel version: parCodaSamples

Examples

```
## Not run:
model <- function() {
    for (i in 1:N) {
        Y[i] ~ dnorm(mu[i], tau)
        mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar <- mean(x[])
    alpha ~ dnorm(0.0, 1.0E-4)
    beta ~ dnorm(0.0, 1.0E-4)
    sigma <- 1.0/sqrt(tau)
    tau ~ dgamma(1.0E-3, 1.0E-3)
}
## data generation
set.seed(1234)
N <- 100
alpha <- 1
beta <- -1
sigma <- 0.5
x <- runif(N)
linpred <- crossprod(t(model.matrix(~x)), c(alpha, beta))
Y <- rnorm(N, mean = linpred, sd = sigma)
jdata <- dclone(list(N = N, Y = Y, x = x), 2, multiply="N")
jpara <- c("alpha", "beta", "sigma")
## jags model
res <- jagsModel(file=model, data=jdata, n.chains = 3, n.adapt=1000)
nclones(res)
update(res, n.iter=1000)
nclones(res)
m <- codaSamples(res, jpara, n.iter=2000)
summary(m)
nclones(m)

## End(Not run)
```

---

dc.fit *Iterative model fitting with data cloning*

Description

jags.fit or bugs.fit is iteratively used to fit a model with increasing the number of clones.

Usage

```
dc.fit(data, params, model, inits, n.clones,
    multiply = NULL, unchanged = NULL,
    update = NULL, updatefun = NULL, initsfun = NULL,
    flavour = c("jags", "bugs"), n.chains = 3,
    return.all=FALSE, ...)
```

Arguments

data	A named list (or environment) containing the data.
params	Character vector of parameters to be sampled. It can be a list of 2 vectors, 1st element is used as parameters to monitor, the 2nd is used as parameters to use in calculating the data cloning diagnostics.
model	Character string (name of the model file), a function containing the model, or a custommodel object (see Examples).
inits	Optional specification of initial values in the form of a list or a function (see Initialization at jags.model). If missing, will be treated as NULL and initial values will be generated automatically.
n.clones	An integer vector containing the numbers of clones to use iteratively.
multiply	Numeric or character index for list element(s) in the data argument to be multiplied by the number of clones instead of repetitions.
unchanged	Numeric or character index for list element(s) in the data argument to be left unchanged.
update	Numeric or character index for list element(s) in the data argument that has to be updated by updatefun in each iterations. This usually is for making priors more informative, and enhancing convergence. See Details and Examples.
updatefun	A function to use for updating data[[update]]. It should take an 'mcmc.list' object as 1st argument, 2nd argument can be the number of clones. See Details and Examples.
initsfun	A function to use for generating initial values, inits are updated by the object returned by this function from the second iteration. If initial values are not dependent on the previous iteration, this should be NULL, otherwise, it should take an 'mcmc.list' object as 1st argument, 2nd argument can be the number of clones. This feature is useful if latent nodes are provided in inits so it also requires to be cloned for subsequent iterations. See Details and Examples.
flavour	If "jags", the function jags.fit is called. If "bugs", the function bugs.fit is called.
n.chains	Number of chains to generate.
return.all	Logical. If TRUE, all the MCMC list objects corresponding to the sequence n.clones are returned for further inspection. Otherwise only the MCMC list corresponding to highest number of clones is returned with summary statistics for the rest.
...	Other values supplied to jags.fit, or bugs.fit, depending on the flavour argument.

Details

The function fits a JAGS/BUGS model with increasing numbers of clones, as supplied by the argument `n.clones`. Data cloning is done by the function `dclone` using the arguments `multiply` and `unchanged`. An updating function can be provided, see Examples.

Value

An object inheriting from the class 'mcmc.list'.

Author(s)

Peter Solymos, <solymos@ualberta.ca>, implementation is based on many discussions with Khuram Nadeem and Subhash Lele.

References

Lele, S.R., B. Dennis and F. Lutscher, 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods. *Ecology Letters* **10**, 551–563.

Lele, S. R., K. Nadeem and B. Schmuland, 2010. Estimability and likelihood inference for generalized linear mixed models using data cloning. *Journal of the American Statistical Association* **105**, 1617–1625.

Solymos, P., 2010. dclone: Data Cloning in R. *The R Journal* **2(2)**, 29–37. URL: https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Solymos.pdf

See Also

Data cloning: `dclone`.

Parallel computations: `dc.parfit`

Model fitting: `jags.fit`, `bugs.fit`

Convergence diagnostics: `dctable`, `dcdiag`

Examples

```
## Not run:
## simulation for Poisson GLMM
set.seed(1234)
n <- 20
beta <- c(2, -1)
sigma <- 0.1
alpha <- rnorm(n, 0, sigma)
x <- runif(n)
X <- model.matrix(~x)
linpred <- crossprod(t(X), beta) + alpha
Y <- rpois(n, exp(linpred))
## JAGS model as a function
jfun1 <- function() {
    for (i in 1:n) {
        Y[i] ~ dpois(lambda[i])
```

```
        log(lambda[i]) <- alpha[i] + inprod(X[i,], beta)
        alpha[i] ~ dnorm(0, 1/sigma^2)
    }
    for (j in 1:np) {
        beta[j] ~ dnorm(0, 0.001)
    }
    sigma ~ dlnorm(0, 0.001)
}
## data
jdata <- list(n = n, Y = Y, X = X, np = NCOL(X))
## inits with latent variable and parameters
ini <- list(alpha=rep(0,n), beta=rep(0, NCOL(X)))
## function to update inits
ifun <- function(model, n.clones) {
    list(alpha=dclone(rep(0,n), n.clones),
        beta=coef(model)[-length(coef(model))])
}
## iteartive fitting
jmod <- dc.fit(jdata, c("beta", "sigma"), jfun1, ini,
    n.clones = 1:5, multiply = "n", unchanged = "np",
    initsfun=ifun)
## summary with DC SE and R hat
summary(jmod)
dct <- dctable(jmod)
plot(dct)
## How to use estimates to make priors more informative?
glmm.model.up <- function() {
    for (i in 1:n) {
        Y[i] ~ dpois(lambda[i])
        log(lambda[i]) <- alpha[i] + inprod(X[i,], beta[1,])
        alpha[i] ~ dnorm(0, 1/sigma^2)
    }
    for (j in 1:p) {
        beta[1,j] ~ dnorm(priors[j,1], priors[j,2])
    }
    sigma ~ dgamma(priors[(p+1),2], priors[(p+1),1])
}
## function for updating, x is an MCMC object
upfun <- function(x) {
    if (missing(x)) {
        p <- ncol(X)
        return(cbind(c(rep(0, p), 0.001), rep(0.001, p+1)))
    } else {
        par <- coef(x)
        return(cbind(par, rep(0.01, length(par))))
    }
}
updat <- list(n = n, Y = Y, X = X, p = ncol(X), priors = upfun())
dcmod <- dc.fit(updat, c("beta", "sigma"), glmm.model.up,
    n.clones = 1:5, multiply = "n", unchanged = "p",
    update = "priors", updatefun = upfun)
summary(dcmod)
## time series example
```

```
## data and model taken from Ponciano et al. 2009
## Ecology 90, 356-362.
paurelia <- c(17,29,39,63,185,258,267,392,510,
    570,650,560,575,650,550,480,520,500)
dat <- list(ncl=1, n=length(paurelia), Y=dcdim(data.matrix(paurelia)))
beverton.holt <- function() {
    for (k in 1:ncl) {
        for(i in 2:(n+1)){
            ## observations
            Y[(i-1), k] ~ dpois(exp(X[i, k]))
            ## state
            X[i, k] ~ dnorm(mu[i, k], 1 / sigma^2)
            mu[i, k] <- X[(i-1), k] + log(lambda) - log(1 + beta * exp(X[(i-1), k]))
        }
        ## state at t0
        X[1, k] ~ dnorm(mu0, 1 / sigma^2)
    }
    # Priors on model parameters
    beta ~ dlnorm(-1, 1)
    sigma ~ dlnorm(0, 1)
    tmp ~ dlnorm(0, 1)
    lambda <- tmp + 1
    mu0 <- log(2)  + log(lambda) - log(1 + beta * 2)
}
mod <- dc.fit(dat, c("lambda","beta","sigma"), beverton.holt,
    n.clones=c(1, 2, 5, 10), multiply="ncl", unchanged="n")
## compare with results from the paper:
##   beta   = 0.00235
##   lambda = 2.274
##   sigma  = 0.1274
summary(mod)

## Using WinBUGS/OpenBUGS
library(R2WinBUGS)
data(schools)
dat <- list(J = nrow(schools), y = schools$estimate,
    sigma.y = schools$sd)
bugs.model <- function(){
       for (j in 1:J){
         y[j] ~ dnorm (theta[j], tau.y[j])
         theta[j] ~ dnorm (mu.theta, tau.theta)
         tau.y[j] <- pow(sigma.y[j], -2)
       }
       mu.theta ~ dnorm (0.0, 1.0E-6)
       tau.theta <- pow(sigma.theta, -2)
       sigma.theta ~ dunif (0, 1000)
     }
inits <- function(){
    list(theta=rnorm(nrow(schools), 0, 100), mu.theta=rnorm(1, 0, 100),
         sigma.theta=runif(1, 0, 100))
}
param <- c("mu.theta", "sigma.theta")
if (.Platform$OS.type == "windows") {
```

```
sim2 <- dc.fit(dat, param, bugs.model, n.clones=1:2,
    flavour="bugs", program="WinBUGS", multiply="J",
    n.iter=2000, n.thin=1)
summary(sim2)
}
sim3 <- dc.fit(dat, param, bugs.model, n.clones=1:2,
    flavour="bugs", program="brugs", multiply="J",
    n.iter=2000, n.thin=1)
summary(sim3)
library(R2OpenBUGS)
sim4 <- dc.fit(dat, param, bugs.model, n.clones=1:2,
    flavour="bugs", program="openbugs", multiply="J",
    n.iter=2000, n.thin=1)
summary(sim4)

## End(Not run)
```

`dc.parfit` *Parallel model fitting with data cloning*

Description

Iterative model fitting on parallel workers with different numbers of clones.

Usage

```
dc.parfit(cl, data, params, model, inits, n.clones,
    multiply=NULL, unchanged=NULL,
    update = NULL, updatefun = NULL, initsfun = NULL,
    flavour = c("jags", "bugs"), n.chains = 3,
    partype=c("balancing", "parchains", "both"),
    return.all=FALSE, ...)
```

Arguments

`cl`	A cluster object created by `makeCluster`, or an integer, see `parDosa` and `evalParallelArgument`.
`data`	A named list (or environment) containing the data.
`params`	Character vector of parameters to be sampled. It can be a list of 2 vectors, 1st element is used as parameters to monitor, the 2nd is used as parameters to use in calculating the data cloning diagnostics. (`partype = "both"` currently cannot handle `params` as list.)
`model`	Character string (name of the model file), a function containing the model, or a or `custommodel` object (see Examples).
`inits`	Optional specification of initial values in the form of a list or a function (see Initialization at `jags.model`). If missing, will be treated as `NULL` and initial values will be generated automatically. If this is a function, it must be self containing, i.e. not having references to R objects outside of the function, or the objects should be exported with `clusterExport` before calling `dc.parfit`.

n.clones	An integer vector containing the numbers of clones to use iteratively.
multiply	Numeric or character index for list element(s) in the data argument to be multiplied by the number of clones instead of repetitions.
unchanged	Numeric or character index for list element(s) in the data argument to be left unchanged.
update	Numeric or character index for list element(s) in the data argument that has to be updated by updatefun in each iterations. This usually is for making priors more informative, and enhancing convergence. This argument is ignored if size balancing is used (default), and not ignored when multiple parallel chains are used.
updatefun	A function to use for updating data[[update]]. It should take an 'mcmc.list' object as 1st argument, 2nd argument can be the number of clones. This argument is ignored if size balancing is used (default), and not ignored when multiple parallel chains are used.
initsfun	A function to use for generating initial values, inits are updated by the object returned by this function from the second iteration. If initial values are not dependent on the previous iteration, this should be NULL, otherwise, it should take an 'mcmc.list' object as 1st argument, 2nd argument can be the number of clones. This feature is useful if latent nodes are provided in inits so it also requires to be cloned for subsequent iterations. The 1st argument of the initsfun function is ignored if partype != "parchains" but the function must have a first argument regardless, see Examples.
flavour	If "jags", the function jags.fit is called. If "bugs", the function bugs.fit is called (available with partype = "balancing" only). See Details.
partype	Type of parallel workload distribution, see Details.
n.chains	Number of chains to generate.
return.all	Logical. If TRUE, all the MCMC list objects corresponding to the sequence n.clones are returned for further inspection (this only works with partype = "parchains"). Otherwise only the MCMC list corresponding to highest number of clones is returned with summary statistics for the rest.
...	Other values supplied to jags.fit, or bugs.fit, depending on the flavour argument.

Details

The dc.parfit is a parallel computing version of dc.fit. After parallel computations, temporary objects passed to workers and the **dclone** package is cleaned up. It is not guaranteed that objects already on the workers and independently loaded packages are not affected. Best to start new instances beforehand.

partype="balancing" distributes each model corresponding to values in n.clones as jobs to workers according to size balancing (see parDosa). partype="parchains" makes repeated calls to jags.parfit for each value in n.clones. partype="both" also calls jags.parfit but each chain of each cloned model is distributed as separate job to the workers.

The vector n.clones is used to determine size balancing. If load balancing is also desired besides of size balancing (e.g. due to unequal performance of the workers, the option "dclone.LB" should

be set to TRUE (by using options("dclone.LB" = TRUE)). By default, the "dclone.LB" option is FALSE for reproducibility reasons.

Some arguments from dc.fit are not available in parallel version (update, updatefun, initsfun) when size balancing is used (partype is "balancing" or "both"). These arguments are evaluated only when partype="parchains".

Size balancing is recommended if n.clones is a relatively long vector, while parallel chains might be more efficient when n.clones has few elements. For efficiency reasons, a combination of the two (partype="both") is preferred if cluster size allows it.

Only partype="balancing" is available for flavour="bugs".

Additionally loaded JAGS modules (e.g. "glm") need to be loaded to the workers.

Value

An object inheriting from the class 'mcmc.list'.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

References

Lele, S.R., B. Dennis and F. Lutscher, 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods. *Ecology Letters* **10**, 551–563.

Lele, S. R., K. Nadeem and B. Schmuland, 2010. Estimability and likelihood inference for generalized linear mixed models using data cloning. *Journal of the American Statistical Association* **105**, 1617–1625.

Solymos, P., 2010. dclone: Data Cloning in R. *The R Journal* **2(2)**, 29–37. URL: https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Solymos.pdf

See Also

Sequential version: dc.fit.

Optimizing the number of workers: clusterSize, plotClusterSize.

Underlying functions: jags.fit, bugs.fit.

Examples

```
## Not run:
set.seed(1234)
n <- 20
x <- runif(n, -1, 1)
X <- model.matrix(~x)
beta <- c(2, -1)
mu <- crossprod(t(X), beta)
Y <- rpois(n, exp(mu))
glm.model <- function() {
    for (i in 1:n) {
```

```
        Y[i] ~ dpois(lambda[i])
        log(lambda[i]) <- inprod(X[i,], beta[1,])
    }
    for (j in 1:np) {
        beta[1,j] ~ dnorm(0, 0.001)
    }
}
dat <- list(Y=Y, X=X, n=n, np=ncol(X))
k <- 1:3
## sequential version
dcm <- dc.fit(dat, "beta", glm.model, n.clones=k, multiply="n",
    unchanged="np")
## parallel version
cl <- makePSOCKcluster(3)
pdcm1 <- dc.parfit(cl, dat, "beta", glm.model, n.clones=k,
    multiply="n", unchanged="np",
    partype="balancing")
pdcm2 <- dc.parfit(cl, dat, "beta", glm.model, n.clones=k,
    multiply="n", unchanged="np",
    partype="parchains")
pdcm3 <- dc.parfit(cl, dat, "beta", glm.model, n.clones=k,
    multiply="n", unchanged="np",
    partype="both")
summary(dcm)
summary(pdcm1)
summary(pdcm2)
summary(pdcm3)
stopCluster(cl)
## multicore type forking
if (.Platform$OS.type != "windows") {
mcdcm1 <- dc.parfit(3, dat, "beta", glm.model, n.clones=k,
    multiply="n", unchanged="np",
    partype="balancing")
mcdcm2 <- dc.parfit(3, dat, "beta", glm.model, n.clones=k,
    multiply="n", unchanged="np",
    partype="parchains")
mcdcm3 <- dc.parfit(3, dat, "beta", glm.model, n.clones=k,
    multiply="n", unchanged="np",
    partype="both")
}

## Using WinBUGS/OpenBUGS
library(R2WinBUGS)
data(schools)
dat <- list(J = nrow(schools), y = schools$estimate,
    sigma.y = schools$sd)
bugs.model <- function(){
       for (j in 1:J){
         y[j] ~ dnorm (theta[j], tau.y[j])
         theta[j] ~ dnorm (mu.theta, tau.theta)
         tau.y[j] <- pow(sigma.y[j], -2)
       }
       mu.theta ~ dnorm (0.0, 1.0E-6)
```

```
       tau.theta <- pow(sigma.theta, -2)
       sigma.theta ~ dunif (0, 1000)
     }
inits <- function(){
    list(theta=rnorm(nrow(schools), 0, 100), mu.theta=rnorm(1, 0, 100),
         sigma.theta=runif(1, 0, 100))
}
param <- c("mu.theta", "sigma.theta")
cl <- makePSOCKcluster(2)
if (.Platform$OS.type == "windows") {
sim2 <- dc.parfit(cl, dat, param, bugs.model, n.clones=1:2,
    flavour="bugs", program="WinBUGS", multiply="J",
    n.iter=2000, n.thin=1)
summary(sim2)
}
sim3 <- dc.parfit(cl, dat, param, bugs.model, n.clones=1:2,
    flavour="bugs", program="brugs", multiply="J",
    n.iter=2000, n.thin=1)
summary(sim3)
library(R2OpenBUGS)
sim4 <- dc.parfit(cl, dat, param, bugs.model, n.clones=1:2,
    flavour="bugs", program="openbugs", multiply="J",
    n.iter=2000, n.thin=1)
summary(sim4)
stopCluster(cl)

## simulation for Poisson GLMM with inits
set.seed(1234)
n <- 5
beta <- c(2, -1)
sigma <- 0.1
alpha <- rnorm(n, 0, sigma)
x <- runif(n)
X <- model.matrix(~x)
linpred <- crossprod(t(X), beta) + alpha
Y <- rpois(n, exp(linpred))
## JAGS model as a function
jfun1 <- function() {
    for (i in 1:n) {
        Y[i] ~ dpois(lambda[i])
        log(lambda[i]) <- alpha[i] + inprod(X[i,], beta)
        alpha[i] ~ dnorm(0, 1/sigma^2)
    }
    for (j in 1:np) {
        beta[j] ~ dnorm(0, 0.001)
    }
    sigma ~ dlnorm(0, 0.001)
}
## data
jdata <- list(n = n, Y = Y, X = X, np = NCOL(X))
## inits with latent variable and parameters
ini <- list(alpha=rep(0,n), beta=rep(0, NCOL(X)))
## model arg is necessary as 1st arg,
```

```
## but not used when partype!=parchains
ifun <-
function(model, n.clones) {
    list(alpha=dclone(rep(0,n), n.clones),
        beta=c(0,0))
}
## make cluster
cl <- makePSOCKcluster(2)
## pass global n variable used in ifun to workers
tmp <- clusterExport(cl, "n")
## fit the model
jmod2 <- dc.parfit(cl, jdata, c("beta", "sigma"), jfun1, ini,
    n.clones = 1:2, multiply = "n", unchanged = "np",
    initsfun=ifun, partype="balancing")
stopCluster(cl)

## End(Not run)
```

---

dclone *Cloning R objects*

Description

Makes clones of R objects, that is values in the object are repeated n times, leaving the original structure of the object intact (in most of the cases).

Usage

```
dclone(x, n.clones=1, ...)
## Default S3 method:
dclone(x, n.clones = 1, attrib=TRUE, ...)
## S3 method for class 'dcdim'
dclone(x, n.clones = 1, attrib=TRUE, ...)
## S3 method for class 'dciid'
dclone(x, n.clones = 1, attrib=TRUE, ...)
## S3 method for class 'dctr'
dclone(x, n.clones = 1, attrib=TRUE, ...)
## S3 method for class 'list'
dclone(x, n.clones = 1,
    multiply = NULL, unchanged = NULL, attrib=TRUE, ...)
## S3 method for class 'environment'
dclone(x, n.clones = 1,
    multiply = NULL, unchanged = NULL, attrib=TRUE, ...)
dcdim(x, drop = TRUE, perm = NULL)
dciid(x, iid=character(0))
dctr(x)
```

Arguments

x	An R object to be cloned, or a cloned object to print.
n.clones	Number of clones.
multiply	Numeric or character index for list element(s) to be multiplied by n.clones instead of repetitions (as done by dclone.default).
unchanged	Numeric or character index for list element(s) to be left unchanged.
attrib	Logical, TRUE if attributes are to be attached.
drop	Logical, if TRUE, deletes the last dimension of an array if that have only one level.
perm	The subscript permutation value, if the cloning dimension is not the last.
iid	Character (or optionally numeric or logical). Column(s) to be treated as i.i.d. observations. Ignored when x is a vector.
...	Other arguments passed to function.

Details

dclone is a generic function for cloning objects. It is separate from rep, because there are different ways of cloning, depending on the BUGS code implementation:

(1) Unchanged: no cloning at all (for e.g. constants).

(2) Repeat: this is the most often used cloning method, repeating the observations row-wise as if there were more samples. The dctr option allows repeating the data column-wise.

(3) Multiply: sometimes it is enough to multiply the numbers (e.g. for Binomial distribution).

(4) Add dimension: under specific circumstances, it is easier to add another dimension for clones, but otherwise repeat the observations (e.g. in case of time series, or for addressing special indexing conventions in the BUGS code, see examples dcdim and dclone.dcdim).

(5) Repeat pattern (i.i.d.): this is useful for example when a grouping variable is considered, and more i.i.d. groups are to be added to the data set. E.g. c(1, 1, 2, 2) is to be cloned as c(1, 1, 2, 2, 3, 3, 4, 4) instead of c(1, 1, 2, 2, 1, 1, 2, 2).

Value

An object with class attributes "dclone" plus the original one(s). Dimensions of the original object might change according to n.clones. The function tries to take care of names, sometimes replacing those with the combination of the original names and an integer for number of clones.

dcdim sets the class attribute of an object to "dcdim", thus dclone will clone the object by adding an extra dimension for the clones.

dciid sets the class attribute of an object to "dciid", thus dclone will clone the object by treating columns defined by the iid argument as i.i.d. observations. These columns must be numeric. This aims to facilitates working with the **INLA** package to generate approximate marginals based on DC. Columns specified by iid will be replaced by an increasing sequence of values respecting possible grouping structure (see Examples).

Lists (i.e. BUGS data objects) are handled differently to enable element specific determination of the mode of cloning. This can be done via the unchanged and multiply arguments, or by setting the behaviour by the dcdim function.

Environments are coerced into a list, and return value is identical to dclone(as.list(x), ...).

Author(s)

Peter Solymos, <solymos@ualberta.ca>, implementation is based on many discussions with Khurram Nadeem and Subhash Lele.

References

Lele, S.R., B. Dennis and F. Lutscher, 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods. *Ecology Letters* **10**, 551–563.

Lele, S. R., K. Nadeem and B. Schmuland, 2010. Estimability and likelihood inference for generalized linear mixed models using data cloning. *Journal of the American Statistical Association* **105**, 1617–1625.

Solymos, P., 2010. dclone: Data Cloning in R. *The R Journal* **2(2)**, 29–37. URL: https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Solymos.pdf

Examples

```
## scalar
dclone(4, 2)
## vector
(x <- 1:6)
dclone(x, 2)
## matrix
(m <- matrix(x, 2, 3))
dclone(m, 2)
## data frame
(dfr <- as.data.frame(t(m)))
dclone(dfr, 2)
## list
(l <- list(n = 10, y = 1:10, x = 1:10, p = 1))
dclone(l, 2)
dclone(as.environment(l), 2)
dclone(l, 2, attrib = FALSE)
dclone(l, 2, multiply = "n", unchanged = "p")
## effect of dcdim
l$y <- dcdim(l$y)
dclone(l, 2, multiply = "n", unchanged = "p")
## time series like usage of dcdim
z <- data.matrix(rnorm(10))
dclone(dcdim(z), 2)
## usage if dciid
ll <- dciid(data.frame(x=1:10, y=1:10), iid="y")
dclone(ll, 2)
## respecting grouping structure in iid
ll$y <- rep(1:5, each=2)
(dci <- dclone(ll, 2))
nclones(dci)
## repeating the data column-wise
dclone(dctr(m), 2)
```

DcloneEnv *Manipulating dclone environments*

Description

Manipulating dclone environments.

Usage

```
pullDcloneEnv(x, type = c("model", "results"))
pushDcloneEnv(x, value, type = c("model", "results"))
clearDcloneEnv(..., list = character(),
    type = c("model", "results"))
listDcloneEnv(type = c("model", "results"))
existsDcloneEnv(x, type = c("model", "results"),
    mode = "any", inherits = TRUE)
```

Arguments

x	a variable name, given as a character string. No coercion is done, and the first element of a character vector of length greater than one will be used, with a warning.
value	a value to be assigned to x.
type	character, the type of environment to be accessed, see Details.
...	the objects to be removed, as names (unquoted) or character strings (quoted).
list	a character vector naming objects to be removed.
mode	the mode or type of object sought: see the exists.
inherits	logical, should the enclosing frames of the environment be searched?

Details

type = "model" manipulates the .DcloneEnvModel environment, which is meant to store temporary objects for model fitting with 'snow' type parallelism (see parDosa for the implementation). This is swiped clean after use.

Thetype = "results" manipulates the .DcloneEnvResults environment, which is meant to store result objects on the workers. This is *not* swiped clean after use.

pullDcloneEnv pulls an object from these environments, similar to get in effect.

pushDcloneEnv pushes an object to these environments, similar to assign in effect.

clearDcloneEnv removes object(s) from these environments, similar to rm in effect.

listDcloneEnv lists name(s) of object(s) in these environments, similar to ls in effect.

existsDcloneEnv tests if an object exists in these environments, similar to exists in effect.

Value

For pullDcloneEnv, the object found. If no object is found an error results.

pushDcloneEnv is invoked for its side effect, which is assigning value to the variable x.

For clearDcloneEnv its is the side effect of an object removed. No value returned.

listDcloneEnv returns a character vector.

existsDcloneEnv returns logical, TRUE if and only if an object of the correct name and mode is found.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

parDosa

---

dcoptions *Setting Options*

Description

Setting options.

Usage

```
dcoptions(...)
```

Arguments

...	Arguments in tag = value form, or a list of tagged values. The tags must come from the parameters described below.

Details

dcoptions is a convenient way of handling options related to the package.

Value

When parameters are set by dcoptions, their former values are returned in an invisible named list. Such a list can be passed as an argument to dcoptions to restore the parameter values. Tags are the following:

autoburnin	logical, to use in gelman.diag (default is TRUE).
diag	critical value to use for data cloning convergence diagnostics, default is 0.05.
LB	logical, should load balancing be used, default is FALSE.

overwrite	logical, should existing model file be overwritten, default is TRUE.
rhat	critical value for testing chain convergence, default is 1.1.
RNG	parallel RNG type, either "none" (default), or "RNGstream", see clusterSetRNGStream.
verbose	integer, should output be verbose (>0) or not (0), default is 1.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

Examples

```
## set LB option, but store old value
ov <- dcoptions("LB"=TRUE)
## this is old value
ov
## this is new value
getOption("dcoptions")
## reset to old value
dcoptions(ov)
## check reset
getOption("dcoptions")
```

---

dctable *Retrieve descriptive statistics from fitted objects to evaluate convergence*

Description

The function is used to retrieve descriptive statistics from fitted objects on order to evaluate convergence of the data cloning algorithm. This is best done via visual display of the results, separately for each parameters of interest.

Usage

```
dctable(x, ...)
## Default S3 method:
dctable(x, ...)
## S3 method for class 'dctable'
plot(x, which = 1:length(x),
    type = c("all", "var", "log.var"),
    position = "topright", box.cex = 0.75, box.bg, ...)
extractdctable(x, ...)
## Default S3 method:
extractdctable(x, ...)

dcdiag(x, ...)
```

```
## Default S3 method:
dcdiag(x, ...)
## S3 method for class 'dcdiag'
plot(x, which = c("all", "lambda.max",
    "ms.error", "r.squared", "log.lambda.max"),
    position = "topright", ...)
extractdcdiag(x, ...)
## Default S3 method:
extractdcdiag(x, ...)
```

Arguments

x	An MCMC or a 'dctable' object.
...	Optionally more fitted model objects for function dctable.
which	What to plot. For dctable, character names or integer indices of the estimated parameters are accepted. for dcdiag it should be one of c("all", "lambda.max", "ms.error", "r.squ
type	Type of plot to be drawn. See Details.
position	Position for the legend, as for legend.
box.cex	Scaling factor for the interquartile boxes.
box.bg	Background color for the interquartile boxes.

Details

dctable returns the "dctable" attribute of the MCMC object, or if it is NULL, calculates the dctable summaries. If more than one fitted objects are provided, summaries are calculated for all objects, and results are ordered by the number of clones.

The plot method for dctable helps in graphical representation of the descriptive statistics. type = "all" results in plotting means, standard deviations and quantiles against the number of clones as boxplot. type = "var" results in plotting the scaled variances against the number of clones. In this case variances are divided by the variance of the model with smallest number of clones, min(n.clones). type = "log.var" is the same as "var", but on the log scale. Along with the values, the min(n.clones) / n.clones line is plotted for reference.

Lele et al. (2010) introduced diagnostic measures for checking the convergence of the data cloning algorithm which are based on the joint posterior distribution and not only on single parameters. These include to calculate the largest eigenvalue of the posterior variance covariance matrix (lambda.max as returned by lambdamax.diag), or to calculate mean squared error (ms.error) and another correlation-like fit statistic (r.squared) based on a Chi-squared approximation (as returned by chisq.diag). The maximum eigenvalue reflects the degenerateness of the posterior distribution, while the two fit measures reflect if the Normal approximation is adequate. All three statistics should converge to zero as the number of clones increases. If this happens, different prior specifications are no longer influencing the results (Lele et al., 2007, 2010). These are conveniently collected by the dcdiag function.

IMPORTANT! Have you checked if different prior specifications lead to the same results?

Value

An object of class 'dctable'. It is a list, and contains as many data frames as the number of parameters in the fitted object. Each data frame contains descriptives as the function of the number of clones.

`dcdiag` returns a data frame with convergence diagnostics.

The `plot` methods produce graphs as side effect.

Author(s)

Peter Solymos, <solymos@ualberta.ca>, implementation is based on many discussions with Khurram Nadeem and Subhash Lele.

References

Lele, S.R., B. Dennis and F. Lutscher, 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods. *Ecology Letters* **10**, 551–563.

Lele, S. R., K. Nadeem and B. Schmuland, 2010. Estimability and likelihood inference for generalized linear mixed models using data cloning. *Journal of the American Statistical Association* **105**, 1617–1625.

Solymos, P., 2010. dclone: Data Cloning in R. *The R Journal* **2(2)**, 29–37. URL: https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Solymos.pdf

See Also

Data cloning: `dclone`

Model fitting: `jags.fit`, `bugs.fit`, `dc.fit`

Examples

```
## Not run:
## simulation for Poisson GLMM
set.seed(1234)
n <- 20
beta <- c(2, -1)
sigma <- 0.1
alpha <- rnorm(n, 0, sigma)
x <- runif(n)
X <- model.matrix(~x)
linpred <- crossprod(t(X), beta) + alpha
Y <- rpois(n, exp(linpred))
## JAGS model as a function
jfun1 <- function() {
    for (i in 1:n) {
        Y[i] ~ dpois(lambda[i])
        log(lambda[i]) <- alpha[i] + inprod(X[i,], beta[1,])
        alpha[i] ~ dnorm(0, 1/sigma^2)
    }
    for (j in 1:np) {
```

```
        beta[1,j] ~ dnorm(0, 0.001)
    }
    sigma ~ dlnorm(0, 0.001)
}
## data
jdata <- list(n = n, Y = Y, X = X, np = NCOL(X))
## number of clones to be used, etc.
## iteartive fitting
jmod <- dc.fit(jdata, c("beta", "sigma"), jfun1,
    n.clones = 1:5, multiply = "n", unchanged = "np")
## summary with DC SE and R hat
summary(jmod)
dct <- dctable(jmod)
plot(dct)
## How to use estimates to make priors more informative?
glmm.model.up <- function() {
    for (i in 1:n) {
        Y[i] ~ dpois(lambda[i])
        log(lambda[i]) <- alpha[i] + inprod(X[i,], beta[1,])
        alpha[i] ~ dnorm(0, 1/sigma^2)
    }
    for (j in 1:p) {
        beta[1,j] ~ dnorm(priors[j,1], priors[j,2])
    }
    sigma ~ dgamma(priors[(p+1),2], priors[(p+1),1])
}
## function for updating, x is an MCMC object
upfun <- function(x) {
    if (missing(x)) {
        p <- ncol(X)
        return(cbind(c(rep(0, p), 0.001), rep(0.001, p+1)))
    } else {
        par <- coef(x)
        return(cbind(par, rep(0.01, length(par))))
    }
}
updat <- list(n = n, Y = Y, X = X, p = ncol(X), priors = upfun())
dcmod <- dc.fit(updat, c("beta", "sigma"), glmm.model.up,
    n.clones = 1:5, multiply = "n", unchanged = "p",
    update = "priors", updatefun = upfun)
summary(dcmod)
dct <- dctable(dcmod)
plot(dct)
plot(dct, type = "var")

## End(Not run)
```

`errlines` *Plot error bars*

Description

The function plots error bars to existing plot.

Usage

```
errlines(x, ...)
## Default S3 method:
errlines(x, y, type = "l", code = 0,
    width = 0, vertical = TRUE, col = 1, bg = NA, ...)
```

Arguments

x	Numeric vector with coordinates along the horizontal axis (if `vertical = FALSE`, this sets the vertical axis).
y	A matrix-like object with 2 columns for lower and upper values on the vertical axis (if `vertical = FALSE`, this sets the horizontal axis).
type	Character, `"l"` for lines, `"b"` for boxes to be drawn.
code	Integer code, determining the kind of ticks to be drawn. See Details.
width	Numeric, width of the ticks (if `type = "l"`) or width of the boxes (if `type = "b"`).
vertical	Logical, if errorbars should be plotted vertically or horizontally.
col	Color of the error lines to be drawn, recycled if needed.
bg	If `type = "b"` the background color of the boxes. By default, no background color used.
...	Other arguments passed to the function `lines`.

Details

The `errlines` function uses `lines` to draw error bars to existing plot when `type = "l"`. `polygon` is used for boxes when `type = "b"`.

If `code = 0` no ticks are drawn, if `code = 1`, only lower ticks are drawn, if `code = 2` only lower ticks are drawn, if `code = 3` both lower and upper ticks are drawn.

Value

Adds error bars to an existing plot as a side effect. Returns `NULL` invisibly.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

`lines`, `polygon`

Examples

```
x <- 1:10
a <- rnorm(10,10)
a <- a[order(a)]
b <- runif(10)
y <- cbind(a-b, a+b+rev(b))
opar <- par(mfrow=c(2, 3))
plot(x, a, ylim = range(y))
errlines(x, y)
plot(x, a, ylim = range(y))
errlines(x, y, width = 0.5, code = 1)
plot(x, a, ylim = range(y), col = 1:10)
errlines(x, y, width = 0.5, code = 3, col = 1:10)
plot(x, a, ylim = range(y))
errlines(x, y, width = 0.5, code = 2, type = "b")
plot(x, a, ylim = range(y))
errlines(x, y, width = 0.5, code = 3, type = "b")
plot(x, a, ylim = range(y), type = "n")
errlines(x, y, width = 0.5, code = 3, type = "b", bg = 1:10)
errlines(x, cbind(a-b/2, a+b/2+rev(b)/2))
points(x, a)
par(opar)
```

---

evalParallelArgument *Evaluates parallel argument*

Description

Evaluates parallel argument.

Usage

```
evalParallelArgument(cl, quit = FALSE)
```

Arguments

cl	NULL, a cluster object or an integer. Can be missing.
quit	Logical, whether it should stop with error when ambiguous parallel definition is found (conflicting default environmental variable settings).

Value

NULL for sequential evaluation or the original value of cl if parallel evaluation is meaningful.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

Examples

```
evalParallelArgument()
evalParallelArgument(NULL)
evalParallelArgument(1)
evalParallelArgument(2)
cl <- makePSOCKcluster(2)
evalParallelArgument(cl)
stopCluster(cl)
oop <- options("mc.cores"=2)
evalParallelArgument()
options(oop)
```

---

jags.fit *Fit JAGS models with cloned data*

Description

Convenient functions designed to work well with cloned data arguments and JAGS.

Usage

```
jags.fit(data, params, model, inits = NULL, n.chains = 3,
    n.adapt = 1000, n.update = 1000, thin = 1, n.iter = 5000,
    updated.model = TRUE, ...)
```

Arguments

data	A named list or environment containing the data. If an environment, data is coerced into a list.
params	Character vector of parameters to be sampled.
model	Character string (name of the model file), a function containing the model, or a or custommodel object (see Examples).
inits	Optional specification of initial values in the form of a list or a function (see Initialization at jags.model). If NULL, initial values will be generated automatically. It is an error to supply an initial value for an observed node.
n.chains	Number of chains to generate.
n.adapt	Number of steps for adaptation.
n.update	Number of updates before iterations. It is usually a bad idea to use n.update=0 if n.adapt>0, so a warning is issued in such cases.
thin	Thinning value.
n.iter	Number of iterations.
updated.model	Logical, if the updated model should be attached as attribute (this can be used to further update if convergence was not satisfactory, see updated.model and update.mcmc.list).
...	Further arguments passed to coda.samples, and update.jags (e.g. the progress.bar argument).

Value

An mcmc.list object. If data cloning is used via the data argument, summary returns a modified summary containing scaled data cloning standard errors (scaled by sqrt(n.clones), see dcsd), and R_{hat} values (as returned by gelman.diag).

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Underlying functions: jags.model, update.jags, coda.samples

Parallel chain computations: jags.parfit

Methods: dcsd, confint.mcmc.list.dc, coef.mcmc.list, quantile.mcmc.list, vcov.mcmc.list.dc

Examples

```
## Not run:
if (require(rjags)) {
## simple regression example from the JAGS manual
jfun <- function() {
    for (i in 1:N) {
        Y[i] ~ dnorm(mu[i], tau)
        mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar <- mean(x[])
    alpha ~ dnorm(0.0, 1.0E-4)
    beta ~ dnorm(0.0, 1.0E-4)
    sigma <- 1.0/sqrt(tau)
    tau ~ dgamma(1.0E-3, 1.0E-3)
}
## data generation
set.seed(1234)
N <- 100
alpha <- 1
beta <- -1
sigma <- 0.5
x <- runif(N)
linpred <- crossprod(t(model.matrix(~x)), c(alpha, beta))
Y <- rnorm(N, mean = linpred, sd = sigma)
## list of data for the model
jdata <- list(N = N, Y = Y, x = x)
## what to monitor
jpara <- c("alpha", "beta", "sigma")
## fit the model with JAGS
regmod <- jags.fit(jdata, jpara, jfun, n.chains = 3)
## model summary
summary(regmod)
## data cloning
dcdata <- dclone(jdata, 5, multiply = "N")
dcmod <- jags.fit(dcdata, jpara, jfun, n.chains = 3)
```

```
summary(dcmod)
}

## End(Not run)
```

`jags.parfit` *Parallel computing with JAGS*

Description

Does the same job as `jags.fit`, but parallel chains are run on parallel workers, thus computations can be faster (up to 1/`n.chains`) for long MCMC runs.

Usage

```
jags.parfit(cl, data, params, model, inits = NULL, n.chains = 3, ...)
```

Arguments

`cl`	A cluster object created by `makeCluster`, or an integer, see `parDosa` and `evalParallelArgument`.
`data`	A named list or environment containing the data. If an environment, `data` is coerced into a list.
`params`	Character vector of parameters to be sampled.
`model`	Character string (name of the model file), a function containing the model, or a or `custommodel` object (see Examples).
`inits`	Specification of initial values in the form of a list or a function, can be missing. Missing value setting can include RNG seed information, see Initialization at `jags.model`. If this is a function and using 'snow' type cluster as `cl`, the function must be self containing, i.e. not having references to R objects outside of the function, or the objects should be exported with `clusterExport` before calling `jags.parfit`. Forking type parallelism does not require such attention.
`n.chains`	Number of chains to generate, must be higher than 1. Ideally, this is equal to the number of parallel workers in the cluster.
`...`	Other arguments passed to `jags.fit`.

Details

Chains are run on parallel workers, and the results are combined in the end.

No update method is available for parallel `mcmc.list` objects. See `parUpdate` and related parallel functions (`parJagsModel`, `parCodaSamples`) for such purpose.

Additionally loaded JAGS modules (e.g. `"glm"`, `"lecuyer"`) need to be loaded to the workers when using 'snow' type cluster as `cl` argument. See Examples.

The use of the `"lecuyer"` module is recommended when running more than 4 chains. See Examples and `parallel.inits`.

Value

An `mcmc.list` object.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Sequential version: `jags.fit`

Function for stepwise modeling with JAGS: `parJagsModel`, `parUpdate`, `parCodaSamples`

Examples

```
## Not run:
if (require(rjags)) {
set.seed(1234)
n <- 20
x <- runif(n, -1, 1)
X <- model.matrix(~x)
beta <- c(2, -1)
mu <- crossprod(t(X), beta)
Y <- rpois(n, exp(mu))
glm.model <- function() {
    for (i in 1:n) {
        Y[i] ~ dpois(lambda[i])
        log(lambda[i]) <- inprod(X[i,], beta[1,])
    }
    for (j in 1:np) {
        beta[1,j] ~ dnorm(0, 0.001)
    }
}
dat <- list(Y=Y, X=X, n=n, np=ncol(X))
load.module("glm")
m <- jags.fit(dat, "beta", glm.model)
cl <- makePSOCKcluster(3)
## load glm module
tmp <- clusterEvalQ(cl, library(dclone))
parLoadModule(cl, "glm")
pm <- jags.parfit(cl, dat, "beta", glm.model)
## chains are not identical -- this is good
pm[1:2,]
summary(pm)
## examples on how to use initial values
## fixed initial values
inits <- list(list(beta=matrix(c(0,1),1,2)),
    list(beta=matrix(c(1,0),1,2)),
    list(beta=matrix(c(0,0),1,2)))
pm2 <- jags.parfit(cl, dat, "beta", glm.model, inits)
## random numbers generated prior to jags.parfit
inits <- list(list(beta=matrix(rnorm(2),1,2)),
```

```
    list(beta=matrix(rnorm(2),1,2)),
    list(beta=matrix(rnorm(2),1,2)))
pm3 <- jags.parfit(cl, dat, "beta", glm.model, inits)
## self contained function
inits <- function() list(beta=matrix(rnorm(2),1,2))
pm4 <- jags.parfit(cl, dat, "beta", glm.model, inits)
## function pointing to the global environment
fun <- function() list(beta=matrix(rnorm(2),1,2))
inits <- function() fun()
clusterExport(cl, "fun")
## using the L'Ecuyer module with 6 chains
load.module("lecuyer")
parLoadModule(cl,"lecuyer")
pm5 <- jags.parfit(cl, dat, "beta", glm.model, inits,
    n.chains=6)
nchain(pm5)
unload.module("lecuyer")
parUnloadModule(cl,"lecuyer")
stopCluster(cl)
## multicore type forking
if (.Platform$OS.type != "windows") {
pm6 <- jags.parfit(3, dat, "beta", glm.model)
}
}

## End(Not run)
```

---

jagsModel *Create a JAGS model object*

Description

jagsModel is used to create an object representing a Bayesian graphical model, specified with a BUGS-language description of the prior distribution, and a set of data. This function uses jags.model but keeps track of data cloning information supplied via the data argument. The model argument can also accept functions or 'custommodel' objects.

Usage

```
jagsModel(file, data=sys.frame(sys.parent()), inits, n.chains = 1,
    n.adapt=1000, quiet=FALSE)
```

Arguments

file	the name of the file containing a description of the model in the JAGS dialect of the BUGS language. Alternatively, file can be a readable text-mode connection, or a complete URL. It can be also a function or a custommodel object.
data	a list or environment containing the data. Any numeric objects in data corresponding to node arrays used in file are taken to represent the values of observed nodes in the model

inits	optional specification of initial values in the form of a list or a function. If omitted, initial values will be generated automatically. It is an error to supply an initial value for an observed node.
n.chains	the number of chains for the model
n.adapt	the number of iterations for adaptation. See adapt for details. If n.adapt = 0 then no adaptation takes place.
quiet	if TRUE then messages generated during compilation will be suppressed.

Value

parJagsModel returns an object inheriting from class jags which can be used to generate dependent samples from the posterior distribution of the parameters.

An object of class jags is a list of functions that share a common environment, see jags.model for details.

An n.clones attribute is attached to the object when applicable.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Underlying functions: jags.model, update.jags

See example on help page of codaSamples.

Parallel version: parJagsModel

---

lambdamax.diag *Data Cloning Diagnostics*

Description

These functions calculates diagnostics for evaluating data cloning convergence.

Usage

```
lambdamax.diag(x, ...)
## S3 method for class 'mcmc.list'
lambdamax.diag(x, ...)

chisq.diag(x, ...)
## S3 method for class 'mcmc.list'
chisq.diag(x, ...)
```

Arguments

x	An object of class mcmc or mcmc.list.
...	Other arguments to be passed.

Details

These diagnostics can be used to test for the data cloning convergence (Lele et al. 2007, 2010). Asymptotically the posterior distribution of the parameters approaches a degenerate multivariate normal distribution. As the distribution is getting more degenerate, the maximal eigenvalue (λ_{max}) of the unscaled covariance matrix is decreasing. There is no critical value under which λ_{max} is good enough. By default, 0.05 is used (see `getOption("dclone")$diag`).

Another diagnostic tool is to check if the joint posterior distribution is multivariate normal. It is done by `chisq.diag` as described by Lele et al. (2010).

Value

`lambdamax.diag` returns a single value, the maximum of the eigenvalues of the unscaled variance covariance matrix of the estimated parameters.

`chisq.diag` returns two test statistic values (mean squared error and r-squared) with empirical and theoretical quantiles.

Author(s)

Khurram Nadeem, <knadeem@math.ualberta.ca>

Peter Solymos, <solymos@ualberta.ca>

References

Lele, S.R., B. Dennis and F. Lutscher, 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods. *Ecology Letters* **10**, 551–563.

Lele, S. R., K. Nadeem and B. Schmuland, 2010. Estimability and likelihood inference for generalized linear mixed models using data cloning. *Journal of the American Statistical Association* **105**, 1617–1625.

Solymos, P., 2010. dclone: Data Cloning in R. *The R Journal* **2(2)**, 29–37. URL: https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Solymos.pdf

See Also

Eigen decomposition: `eigen`

Examples

```
data(regmod)
lambdamax.diag(regmod)
chisq.diag(regmod)
```

`make.symmetric` *Make a square matrix symmetric by averaging.*

Description

Matrix symmetry might depend on numerical precision issues. The older version of JAGS had a bug related to this issue for multivariate normal nodes. This simple function can fix the issue, but new JAGS versions do not require such intervention.

Usage

```
make.symmetric(x)
```

Arguments

`x`	A square matrix.

Details

The function takes the average as `(x[i, j] + x[j, i]) / 2` for each off diagonal cells.

Value

A symmetric square matrix.

Note

The function works for any matrix, even for those not intended to be symmetric.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

Examples

```
x <- as.matrix(as.dist(matrix(1:25, 5, 5)))
diag(x) <- 100
x[lower.tri(x)] <- x[lower.tri(x)] - 0.1
x[upper.tri(x)] <- x[upper.tri(x)] + 0.1
x
make.symmetric(x)
```

`mclapplySB` *Size balancing version of mclapply*

Description

`mclapplySB` is a size balancing version of `mclapply`.

Usage

```
mclapplySB(X, FUN, ...,
    mc.preschedule = TRUE, mc.set.seed = TRUE,
    mc.silent = FALSE, mc.cores = 1L,
    mc.cleanup = TRUE, mc.allow.recursive = TRUE,
    size = 1)
```

Arguments

`X`	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by `as.list`.
`FUN`	the function to be applied to each element of `X`
`...`	optional arguments to `FUN`
`mc.preschedule`	see `mclapply`
`mc.set.seed`	see `mclapply`
`mc.silent`	see `mclapply`
`mc.cores`	The number of cores to use, i.e. how many processes will be spawned (at most)
`mc.cleanup`	see `mclapply`
`mc.allow.recursive`	see `mclapply`
`size`	Vector of problem sizes (or relative performance information) corresponding to elements of `X` (recycled if needed). The default `1` indicates equality of problem sizes.

Details

`mclapply` gives details of the forking mechanism.

`mclapply` is used unmodified if sizes of the jobs are equal (`length(unique(size)) == 1`). Size balancing (as described in `parDosa`) is used to balance workload on the child processes otherwise.

Value

A list.

Author(s)

Peter Solymos

See Also

mclapply, parDosa

---

mcmc.list-methods *Methods for the 'mcmc.list' class*

Description

Methods for 'mcmc.list' objects.

Usage

```
dcsd(object, ...)
## S3 method for class 'mcmc.list'
dcsd(object, ...)
## S3 method for class 'mcmc.list'
coef(object, ...)
## S3 method for class 'mcmc.list.dc'
confint(object, parm, level = 0.95, ...)
## S3 method for class 'mcmc.list'
vcov(object, ...)
## S3 method for class 'mcmc.list.dc'
vcov(object, invfisher = TRUE, ...)
## S3 method for class 'mcmc.list'
quantile(x, ...)
```

Arguments

x, object	MCMC object to be processed.
parm	A specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
level	The confidence level required.
...	Further arguments passed to functions.
invfisher	Logical, if the inverse of the Fisher information matrix (TRUE) should be returned instead of the variance-covariance matrix of the joint posterior distribution (FALSE).

Value

dcsd returns the data cloning standard errors of a posterior MCMC chain calculated as standard deviation times the square root of the number of clones.

The coef method returns mean of the posterior MCMC chains for the monitored parameters.

The confint method returns Wald-type confidence intervals for the parameters assuming asymptotic normality.

The vcov method returns the inverse of the Fisher information matrix (`invfisher = TRUE`) or the covariance matrix of the joint posterior distribution (`invfisher = FALSE`). The `invfisher` is valid only for `mcmc.list.dc` (data cloned) objects.

The `quantile` method returns quantiles for each variable.

Note

Some functions only available for the 'mcmc.list.dc' class which inherits from class 'mcmc.list'.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

jags.fit, bugs.fit

Examples

```
## Not run:
## simple regression example from the JAGS manual
jfun <- function() {
    for (i in 1:N) {
        Y[i] ~ dnorm(mu[i], tau)
        mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar <- mean(x)
    alpha ~ dnorm(0.0, 1.0E-4)
    beta ~ dnorm(0.0, 1.0E-4)
    sigma <- 1.0/sqrt(tau)
    tau ~ dgamma(1.0E-3, 1.0E-3)
}
## data generation
set.seed(1234)
N <- 100
alpha <- 1
beta <- -1
sigma <- 0.5
x <- runif(N)
linpred <- crossprod(t(model.matrix(~x)), c(alpha, beta))
Y <- rnorm(N, mean = linpred, sd = sigma)
## data for the model
dcdata <- dclone(list(N = N, Y = Y, x = x), 5, multiply = "N")
## data cloning
dcmod <- jags.fit(dcdata, c("alpha", "beta", "sigma"), jfun,
    n.chains = 3)
summary(dcmod)
coef(dcmod)
dcsd(dcmod)
confint(dcmod)
vcov(dcmod)
vcov(dcmod, invfisher = FALSE)
```

```
quantile(dcmod)

## End(Not run)
```

`mcmcapply` *Calculations on 'mcmc.list' objects*

Description

Conveniently calculates statistics for mcmc.list objects.

Usage

```
mcmcapply(x, FUN, ...)
## S3 method for class 'mcmc.list'
stack(x, ...)
```

Arguments

`x`	Objects of class `mcmc.list`.
`FUN`	A function to be used in the calculations, returning a single value.
`...`	Other arguments passed to `FUN`.

Details

`mcmcapply` returns a certain statistics based on `FUN` after coercing into a matrix. `FUN` can be missing, in this case `mcmcapply` is equivalent to calling `as.matrix` on an 'mcmc.list' object.

`stack` can be used to concatenates 'mcmc.list' objects into a single vector along with index variables indicating where each observation originated from (e.g. iteration, variable, chain).

Value

`mcmcapply` returns statistic value for each variable based on `FUN`, using all values in all chains of the MCMC object.

`stack` returns a data frame with columns: iter, variable, chain, value.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

Examples

```
data(regmod)
mcmcapply(regmod, mean)
mcmcapply(regmod, sd)

x <- stack(regmod)
head(x)
summary(x)
library(lattice)
xyplot(value ~ iter | variable, data=x,
    type="l", scales = "free", groups=chain)
```

`nclones` *Number of Clones*

Description

Retrieves the number of clones from an object.

Usage

```
nclones(x, ...)
## Default S3 method:
nclones(x, ...)
## S3 method for class 'list'
nclones(x, ...)
```

Arguments

`x`	An object.
`...`	Other arguments to be passed.

Value

Returns the number of of clones, or `NULL`.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

dclone

Examples

```
x <- dclone(1:10, 10)
nclones(x)
nclones(1:10) # this is NULL
```

ovenbird *Abundances of ovenbird in Alberta*

Description

The data set contains observations (point counts) of 198 sites of the Alberta Biodiversity Monitoring Institute.

`count`: integer, ovenbird counts per site.

`site, year`: numeric, site number and year of data collection.

`ecosite`: factor with 5 levels, ecological categorization of the sites.

`uplow`: factor with 2 levels, ecological categorization of the sites (same es ecosite but levels are grouped into `upland` and `lowland`).

`dsucc, dalien, thd`: numeric, percentage of successional, alienating and total human disturbance based on interpreted 3 x 7 km photoplots centered on each site.

`long, lat`: numeric, public longitude/latitude coordinates of the sites.

Usage

```
data(ovenbird)
```

Source

Alberta Biodiversity Monitoring Institute, http://www.abmi.ca

Examples

```
data(ovenbird)
summary(ovenbird)
str(ovenbird)
```

pairs.mcmc.list *Scatterplot Matrices for 'mcmc.list' Objects*

Description

A matrix of scatterplots is produced.

Usage

```
## S3 method for class 'mcmc.list'
pairs(x, n = 25, col = 1:length(x),
    col.hist = "gold", col.image = terrain.colors(50),
    density = TRUE, contour = TRUE, mean = TRUE, ...)
```

Arguments

x	an 'mcmc.list' object.
n	number of of grid points in each direction for two-dimensional kernel density estimation. Can be scalar or a length-2 integer vector.
col	color for chains in upper panel scatterplots.
col.hist	color for histogram fill in diagonal panels.
col.image	color palette for image plot in lower panel scatterplots.
density	logical, if image plot based on the two-dimensional kernel density estimation should be plotted in lower panel.
contour	logical, if contour plot based on the two-dimensional kernel density estimation should be plotted in lower panel.
mean	logical, if lines should indicate means of the posterior densities in the panels.
...	additional graphical parameters/arguments.

Details

The function produces a scatterplot matrix for 'mcmc.list' objects. Diagonal panels are posterior densities with labels and rug on the top. Upper panels are pairwise bivariate scatterplots with coloring corresponding to chains, thus highlighting mixing properties although not as clearly as trace plots. Lower panels are two-dimensional kernel density estimates based on kde2d function of **MASS** package using image and contour.

Value

The function returns NULL invisibly and produces a plot as a side effect.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

pairs, plot.mcmc.list

Two-dimensional kernel density estimation: kde2d in **MASS** package

Examples

```
data(regmod)
pairs(regmod)
```

`parallel.inits` *Parallel RNGs for initial values*

Description

This function takes care of initial values with safe RNGs based on `parallel.seeds` of the **rjags** package.

Usage

```
parallel.inits(inits, n.chains)
```

Arguments

`inits`	Initial values (see Initialization at `jags.model`). If `NULL`, an empty list of length `n.chains` will be generated and seeded (RNG type and seed).
`n.chains`	Number of chains to generate.

Details

Initial values are handled similar to as it is done in `jags.model`.

RNGs are based on values returned by `parallel.seeds`.

If the `"lecuyer"` JAGS module is active, RNGs are based on the `"lecuyer::RngStream"` factory, otherwise those are based on the `"base::BaseRNG"` factory.

Value

Returns a list of initial values with RNGs.

Author(s)

Peter Solymos, <solymos@ualberta.ca>. Based on Martyn Plummer's `parallel.seeds` function and code in `jags.model` for initial value handling in the **rjags** package.

See Also

`parallel.seeds`, `jags.model`

This seeding function is used in all of **dclone**'s parallel functions that do initialization: `parJagsModel`, `jags.parfit`, `dc.parfit`

Examples

```
if (require(rjags)) {
## "base::BaseRNG" factory.
parallel.inits(NULL, 2)
## "lecuyer::RngStream" factory
load.module("lecuyer")
parallel.inits(NULL, 2)
unload.module("lecuyer")
## some non NULL inits specifications
parallel.inits(list(a=0), 2)
parallel.inits(list(list(a=0), list(a=0)), 2)
parallel.inits(function() list(a=0), 2)
parallel.inits(function(chain) list(a=chain), 2)
}
```

`parCodaSamples` *Generate posterior samples in 'mcmc.list' format on parallel workers*

Description

This function sets a trace monitor for all requested nodes, updates the model on each workers. Finally, it return the chains to the master and coerces the output to a single `mcmc.list` object.

Usage

```
parCodaSamples(cl, model, variable.names, n.iter, thin = 1, na.rm = TRUE, ...)
```

Arguments

`cl`	A cluster object created by `makeCluster`, or an integer. It can also be `NULL`, see `parDosa`.
`model`	character, name of a jags model object
`variable.names`	a character vector giving the names of variables to be monitored
`n.iter`	number of iterations to monitor
`thin`	thinning interval for monitors
`na.rm`	logical flag that indicates whether variables containing missing values should be omitted. See details in help page of `coda.samples`.
`...`	optional arguments that are passed to the update method for jags model objects

Value

An `mcmc.list` object with possibly an `n.clones` attribute.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Original sequential function in **rjags**: coda.samples

Sequential **dclone**-ified version: codaSamples

Examples

```
## Not run:
if (require(rjags)) {
model <- function() {
    for (i in 1:N) {
        Y[i] ~ dnorm(mu[i], tau)
        mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar <- mean(x[])
    alpha ~ dnorm(0.0, 1.0E-4)
    beta ~ dnorm(0.0, 1.0E-4)
    sigma <- 1.0/sqrt(tau)
    tau ~ dgamma(1.0E-3, 1.0E-3)
}
## data generation
set.seed(1234)
N <- 100
alpha <- 1
beta <- -1
sigma <- 0.5
x <- runif(N)
linpred <- crossprod(t(model.matrix(~x)), c(alpha, beta))
Y <- rnorm(N, mean = linpred, sd = sigma)
jdata <- list(N = N, Y = Y, x = x)
jpara <- c("alpha", "beta", "sigma")
## jags model on parallel workers
## n.chains must be <= no. of workers
cl <- makePSOCKcluster(4)
parJagsModel(cl, name="res", file=model, data=jdata,
    n.chains = 2, n.adapt=1000)
parUpdate(cl, "res", n.iter=1000)
m <- parCodaSamples(cl, "res", jpara, n.iter=2000)
stopifnot(2==nchain(m))
## with data cloning
dcdata <- dclone(list(N = N, Y = Y, x = x), 2, multiply="N")
parJagsModel(cl, name="res2", file=model, data=dcdata,
    n.chains = 2, n.adapt=1000)
parUpdate(cl, "res2", n.iter=1000)
m2 <- parCodaSamples(cl, "res2", jpara, n.iter=2000)
stopifnot(2==nchain(m2))
nclones(m2)
stopCluster(cl)
}

## End(Not run)
```

parDosa *Parallel wrapper function to call from within a function*

Description

parDosa is a wrapper function around many functionalities of the **parallel** package. It is designed to work closely with MCMC fitting functions, e.g. can easily be called from inside of a function.

Usage

```
parDosa(cl, seq, fun, cldata,
    lib = NULL, dir = NULL, evalq=NULL,
    size = 1, balancing = c("none", "load", "size", "both"),
    rng.type = c("none", "RNGstream"),
    cleanup = TRUE, unload = FALSE, iseed=NULL, ...)
```

Arguments

cl	A cluster object created by makeCluster, or an integer. It can also be NULL, see Details.
seq	A vector to split.
fun	A function or character string naming a function.
cldata	A list containing data. This list is then exported to the cluster by clusterExport. It is stored in a hidden environment. Data in cldata can be used by fun.
lib	Character, name of package(s). Optionally packages can be loaded onto the cluster. More than one package can be specified as character vector. Packages already loaded are skipped.
dir	Working directory to use, if NULL working directory is not set on workers (default). Can be a vector to set different directories on workers.
evalq	Character, expressions to evaluate, e.g. for changing global options (passed to clusterEvalQ). More than one expressions can be specified as character vector.
balancing	Character, type of balancing to perform (see Details).
size	Vector of problem sizes (or relative performance information) corresponding to elements of seq (recycled if needed). The default 1 indicates equality of problem sizes.
rng.type	Character, "none" will not set any seeds on the workers, "RNGstream" selects the "L'Ecuyer-CMRG" RNG and then distributes streams to the members of a cluster, optionally setting the seed of the streams by set.seed(iseed) (otherwise they are set from the current seed of the master process: after selecting the L'Ecuyer generator). See clusterSetRNGStream. The logical value !(rng.type == "none") is used for forking (e.g. when cl is integer).
cleanup	logical, if cldata should be removed from the workers after applying fun. If TRUE, effects of dir argument is also cleaned up.
unload	logical, if pkg should be unloaded after applying fun.

iseed	integer or NULL, passed to clusterSetRNGStream to be supplied to set.seed on the workers, or NULL not to set reproducible seeds.
...	Other arguments of fun, that are simple values and not objects. (Arguments passed as objects should be specified in cldata, otherwise those are not exported to the cluster by this function.)

Details

The function uses 'snow' type clusters when cl is a cluster object. The function uses 'multicore' type forking (shared memory) when cl is an integer. The value from getOption("mc.cores") is used if the argument is NULL.

The function sets the random seeds, loads packages lib onto the cluster, sets the working directory as dir, exports cldata and evaluates fun on seq.

No balancing (balancing = "none") means, that the problem is split into roughly equal subsets, without respect to size (see clusterSplit). This splitting is deterministic (reproducible).

Load balancing (balancing = "load") means, that the problem is not splitted into subsets *a priori*, but subsequent items are placed on the worker which is empty (see clusterApplyLB for load balancing). This splitting is non-deterministic (might not be reproducible).

Size balancing (balancing = "size") means, that the problem is splitted into subsets, with respect to size (see clusterSplitSB and parLapplySB). In size balancing, the problem is re-ordered from largest to smallest, and then subsets are determined by minimizing the total approximate processing time. This splitting is deterministic (reproducible).

Size and load balancing (balancing = "both") means, that the problem is re-ordered from largest to smallest, and then undeterministic load balancing is used (see parLapplySLB). If size is correct, this is identical to size balancing. This splitting is non-deterministic (might not be reproducible).

Value

Usually a list with results returned by the cluster.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Size balancing: parLapplySB, parLapplySLB, mclapplySB

Optimizing the number of workers: clusterSize, plotClusterSize.

parDosa is used internally by parallel **dclone** functions: jags.parfit, dc.parfit, parJagsModel, parUpdate, parCodaSamples.

parDosa manipulates specific environments described on the help page DcloneEnv.

parJagsModel *Create a JAGS model object on parallel workers*

Description

parJagsModel is used to create an object representing a Bayesian graphical model, specified with a BUGS-language description of the prior distribution, and a set of data.

Usage

```
parJagsModel(cl, name, file, data=sys.frame(sys.parent()),
    inits, n.chains = 1, n.adapt=1000, quiet=FALSE)
```

Arguments

cl	A cluster object created by makeCluster, or an integer. It can also be NULL, see parDosa. Size of the cluster must be equal to or larger than n.chains.
name	character, name for the model to be assigned on the workers.
file	the name of the file containing a description of the model in the JAGS dialect of the BUGS language. Alternatively, file can be a readable text-mode connection, or a complete URL. It can be also a function or a custommodel object.
data	a list or environment containing the data. Any numeric objects in data corresponding to node arrays used in file are taken to represent the values of observed nodes in the model
inits	optional specification of initial values in the form of a list or a function (see Initialization on help page of jags.model). If omitted, initial values will be generated automatically. It is an error to supply an initial value for an observed node.
n.chains	the number of parallel chains for the model
n.adapt	the number of iterations for adaptation. See adapt for details. If n.adapt = 0 then no adaptation takes place.
quiet	if TRUE then messages generated during compilation will be suppressed. Effect of this argument is not visible on the master process.

Value

parJagsModel returns an object inheriting from class jags which can be used to generate dependent samples from the posterior distribution of the parameters. These jags models are residing on the workers, thus updating/sampling is possible.

Length of cl must be equal to or greater than n.chains. RNG seed generation takes place first on the master, and chains then initialized on each worker by distributing inits and single chained models.

An object of class jags is a list of functions that share a common environment, see jags.model for details. Data cloning information is attached to the returned object if data argument has n.clones attribute.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

Original sequential function in **rjags**: `jags.model`

Sequential **dclone**-ified version: `jagsModel`

See example on help page of `parCodaSamples`.

---

parLoadModule *Dynamically load JAGS modules on parallel workers*

Description

A JAGS module is a dynamically loaded library that extends the functionality of JAGS. These functions load and unload JAGS modules and show the names of the currently loaded modules on parallel workers.

Usage

```
parLoadModule(cl, name, path, quiet=FALSE)
parUnloadModule(cl, name, quiet=FALSE)
parListModules(cl)
```

Arguments

`cl`	a cluster object created by the **parallel** package.
`name`	character, name of the module to be loaded
`path`	file path to the location of the DLL. If omitted, the option `jags.moddir` is used to locate the modules. it can be a vector of length `length(cl)` to set different DLL locations on each worker
`quiet`	a logical. If `TRUE`, no message will be printed about loading the module

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

`list.modules`, `load.module`, `unload.module`

Examples

```
## Not run:
if (require(rjags)) {
cl <- makePSOCKcluster(3)
parListModules(cl)
parLoadModule(cl, "glm")
parListModules(cl)
parUnloadModule(cl, "glm")
parListModules(cl)
stopCluster(cl)
}

## End(Not run)
```

---

parSetFactory *Advanced control over JAGS on parallel workers*

Description

JAGS modules contain factory objects for samplers, monitors, and random number generators for a JAGS model. These functions allow fine-grained control over which factories are active on parallel workers.

Usage

```
parListFactories(cl, type)
parSetFactory(cl, name, type, state)
```

Arguments

cl	a cluster object created by the **parallel** package.
name	name of the factory to set
type	type of factory to query or set. Possible values are "sampler", "monitor", or "rng"
state	a logical. If TRUE then the factory will be active, otherwise the factory will become inactive.

Value

parListFactories returns a a list of data frame with two columns per each worker, the first column shows the names of the factory objects in the currently loaded modules, and the second column is a logical vector indicating whether the corresponding factory is active or not.

sparStFactory is called to change the future behaviour of factory objects. If a factory is set to inactive then it will be skipped.

Note

When a module is loaded, all of its factory objects are active. This is also true if a module is unloaded and then reloaded.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

list.modules, set.factory

Examples

```
## Not run:
if (require(rjags)) {
cl <- makePSOCKcluster(3)
parListFactories(cl, "sampler")
parListFactories(cl, "monitor")
parListFactories(cl, "rng")
parSetFactory(cl, "base::Slice", "sampler", FALSE)
parListFactories(cl, "sampler")
parSetFactory(cl, "base::Slice", "sampler", TRUE)
stopCluster(cl)
}

## End(Not run)
```

---

parUpdate *Update jags models on parallel workers*

Description

Update the Markov chain associated with the model on parallel workers. (This represents the 'burn-in' phase when nodes are not monitored.)

Usage

```
parUpdate(cl, object, n.iter=1, ...)
```

Arguments

cl	A cluster object created by makeCluster, or an integer. It can also be NULL, see parDosa.
object	character, name of a jags model object
n.iter	number of iterations of the Markov chain to run
...	additional arguments to the update method, see update.jags

Value

The parUpdate function modifies the original object on parallel workers and returns NULL.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

update.jags

See example on help page of parCodaSamples.

---

regmod *Exemplary MCMC list object*

Description

This data set was made via the jags.fit function.

Usage

```
data(regmod)
```

Source

See Example.

Examples

```
data(regmod)
summary(regmod)
plot(regmod)
## Not run:
## DATA GENERATION
## simple regression example from the JAGS manual
jfun <- function() {
    for (i in 1:N) {
        Y[i] ~ dnorm(mu[i], tau)
        mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar <- mean(x[])
    alpha ~ dnorm(0.0, 1.0E-4)
    beta ~ dnorm(0.0, 1.0E-4)
    sigma <- 1.0/sqrt(tau)
    tau ~ dgamma(1.0E-3, 1.0E-3)
}
## data generation
set.seed(1234)
N <- 100
```

```
alpha <- 1
beta <- -1
sigma <- 0.5
x <- runif(N)
linpred <- crossprod(t(model.matrix(~x)), c(alpha, beta))
Y <- rnorm(N, mean = linpred, sd = sigma)
## list of data for the model
jdata <- list(N = N, Y = Y, x = x)
## what to monitor
jpara <- c("alpha", "beta", "sigma")
## fit the model with JAGS
regmod <- jags.fit(jdata, jpara, jfun, n.chains = 3,
    updated.model = FALSE)

## End(Not run)
```

`update.mcmc.list` *Automatic updating of an MCMC object*

Description

Automatic updating of an MCMC object until a desired statistic value reached.

Usage

```
updated.model(object, ...)
## S3 method for class 'mcmc.list'
update(object, fun,
    times = 1, n.update = 0, n.iter, thin, ...)
```

Arguments

`object`	A fitted MCMC object ('mcmc.list' class for example), with `"updated.model"` attribute.
`fun`	A function that evaluates convergence of the MCMC chains, must return logical result. See Examples. The iterative updating quits when return value is `TRUE`. Can be missing, in which case there is no stopping rule.
`times`	Number of times the updating should be repeated. If `fun` returns `TRUE`, updating is finished and MCMC object is returned.
`n.update`	Number of updating iterations. The default 0 indicates, that only `n.iter` iterations are used.
`n.iter`	Number of iterations for sampling and evaluating `fun`. If missing, value is taken from `object`.
`thin`	Thinning value. If missing, value is taken from `object`.
`...`	Other arguments passed to `coda.samples`.

Details

updated.model can be used to retrieve the updated model from an MCMC object fitted via the function jags.fit and dc.fit (with flavour = "jags"). The update method is a wrapper for this purpose, specifically designed for the case when MCMC convergence is problematic. A function is evaluated on the updated model in each iteration of the updating process, and an MCMC object is returned when iteration ends, or when the evaluated function returns TRUE value.

n.update and n.iter can be vectors, if lengths are shorter then times, values are recycled.

Data cloning information is preserved.

Value

updated.model returns the state of the JAGS model after updating and sampling. This can be further updated by the function update.jags and sampled by coda.samples if convergence diagnostics were not satisfactory.

update returns an MCMC object with "updated.model" attribute.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

jags.fit, coda.samples, update.jags

Examples

```
## Not run:
## simple regression example from the JAGS manual
jfun <- function() {
    for (i in 1:N) {
        Y[i] ~ dnorm(mu[i], tau)
        mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar <- mean(x[])
    alpha ~ dnorm(0.0, 1.0E-4)
    beta ~ dnorm(0.0, 1.0E-4)
    sigma <- 1.0/sqrt(tau)
    tau ~ dgamma(1.0E-3, 1.0E-3)
}
## data generation
set.seed(1234)
N <- 100
alpha <- 1
beta <- -1
sigma <- 0.5
x <- runif(N)
linpred <- crossprod(t(model.matrix(~x)), c(alpha, beta))
Y <- rnorm(N, mean = linpred, sd = sigma)
## list of data for the model
jdata <- list(N = N, Y = Y, x = x)
```

```
## what to monitor
jpara <- c("alpha", "beta", "sigma")
## fit the model with JAGS
regmod <- jags.fit(jdata, jpara, jfun, n.chains = 3)
## get the updated model
upmod <- updated.model(regmod)
upmod
## automatic updating
## using R_hat < 1.1 as criteria
critfun <- function(x)
    all(gelman.diag(x)$psrf[,1] < 1.1)
mod <- update(regmod, critfun, 5)
## update just once
mod2 <- update(regmod)
summary(mod)

## End(Not run)
```

`write.jags.model` *Write and remove model file*

Description

Writes or removes a BUGS model file to or from the hard drive.

Usage

```
write.jags.model(model, filename = "model.txt", digits = 5,
    dir = tempdir(), overwrite = getOption("dcoptions")$overwrite)
clean.jags.model(filename = "model.txt")
custommodel(model, exclude = NULL, digits = 5)
```

Arguments

`model`	JAGS model to write onto the hard drive (see Example). For `write.jags.model`, it can be name of a file or a function, or it can be an 'custommodel' object returned by `custommodel`. `custommodel` can take its `model` argument as function. If `model` is not function, its is coerced as character.
`digits`	Number of significant digits used in the output.
`filename`	Character, the name of the file to write/remove. It can be a `link{connection}`.
`dir`	Optional argument for directory where to write the file. The default is to use a temporary directory and use `file.path(dir, filename)`. When `NULL`, it uses the current working directory (`getwd()`).
`overwrite`	Logical, if `TRUE` the `filename` will be forced and existing file with same name will be overwritten.
`exclude`	Numeric, lines of the model to exclude (see Details).

Details

`write.jags.model` is built upon the function `write.model` of the **R2WinBUGS** package.

`clean.jags.model` is built upon the function `file.remove`, and intended to be used internally to clean up the JAGS model file after estimating sessions, ideally via the `on.exit` function. It requires the full path as returned by `write.jags.model`.

The function `custommodel` can be used to exclude some lines of the model. This is handy when there are variations of the same model. `write.jags.model` accepts results returned by `custommodel`. This is also the preferred way of including BUGS models into R packages, because the function form often includes undefined functions.

Use the `%_%` operator if the model is a function and the model contains truncation (`I()` in WinBUGS, `T()` in JAGS). See explanation on help page of `write.model`.

Value

`write.jags.model` invisibly returns the name of the file that was written eventually (possibly including random string). The return value includes the full path.

`clean.jags.model` invisibly returns the result of `file.remove` (logical).

`custommodel` returns an object of class 'custommodel', which is a character vector.

Author(s)

Peter Solymos, <solymos@ualberta.ca>

See Also

`write.model`, `file.remove`

Examples

```
## Not run:
## simple regression example from the JAGS manual
jfun <- function() {
    for (i in 1:N) {
        Y[i] ~ dnorm(mu[i], tau)
        mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar <- mean(x)
    alpha ~ dnorm(0.0, 1.0E-4)
    beta ~ dnorm(0.0, 1.0E-4)
    sigma <- 1.0/sqrt(tau)
    tau ~ dgamma(1.0E-3, 1.0E-3)
}
## data generation
set.seed(1234)
N <- 100
alpha <- 1
beta <- -1
sigma <- 0.5
x <- runif(N)
```

```
linpred <- crossprod(t(model.matrix(~x)), c(alpha, beta))
Y <- rnorm(N, mean = linpred, sd = sigma)
## list of data for the model
jdata <- list(N = N, Y = Y, x = x)
## what to monitor
jpara <- c("alpha", "beta", "sigma")
## write model onto hard drive
jmodnam <- write.jags.model(jfun)
## fit the model
regmod <- jags.fit(jdata, jpara, jmodnam, n.chains = 3)
## cleanup
clean.jags.model(jmodnam)
## model summary
summary(regmod)

## End(Not run)
## let's customize this model
jfun2 <- structure(
    c(" model { ",
    "     for (i in 1:n) { ",
    "         Y[i] ~ dpois(lambda[i]) ",
    "         Y[i] <- alpha[i] + inprod(X[i,], beta[1,]) ",
    "         log(lambda[i]) <- alpha[i] + inprod(X[i,], beta[1,]) ",
    "         alpha[i] ~ dnorm(0, 1/sigma^2) ",
    "     } ",
    "     for (j in 1:np) { ",
    "         beta[1,j] ~ dnorm(0, 0.001) ",
    "     } ",
    "     sigma ~ dlnorm(0, 0.001) ",
    " } "),
    class = "custommodel")
custommodel(jfun2)
## GLMM
custommodel(jfun2, 4)
## LM
custommodel(jfun2, c(3,5))
## deparse when print
print(custommodel(jfun2), deparse=TRUE)
```

Index