John Miyamoto (jmiyamot@uw.edu), Department of Psychology, University of Washington
Seattle, WA 98195 June 6, 2012

** This document is under revision. A finished copy should be completed in a few days. **

JM Functions for Working with OpenBUGS, WinBUGS & JAGS Output

TERMINOLOGY: In this document, BUGS refers to either OpenBUGS or WinBUGS or JAGS. This terminology is a bit inaccurate because there was an original BUGS program from which all of these are descendents. So far as I know, all of the functions that are described in this document work with all versions of BUGS.

Still to do: Hidden bk = todo

- Redo the sample output to include WinBUGS and JAGS examples for the same data.
- Add the 'wb2ob.mod' function (converts WinBUGS model file to an OpenBUGS model file. This function is based heavily on the 'Win2OpenBUGS' function that can be downloaded from 'http://www.openbugs.info/w/UserContributedCode'.

Add to Documentation: to.mcmc.chains,

- Add 'n.thin' to the output of show.bugs.
- draw.beta & draw.gamma have a strange NA in the stats output. fix.
- Standardize the argument names in all of the distribution plotting functions. Make plot.dist be the argument that controls plotting. Functions that use plot.dist argument:
  - draw.beta, draw.normal, res.param,

#----------h.bk = todo ---##

This document explains the use of my R functions that were designed for looking at BUGS output. The **bugs** function in the **R2OpenBUGS** package transfers data from R to OpenBUGS, tells OpenBUGS where to find the model file that defines the probability model, and returns the results of the OpenBUGS analysis to R in the form of a list. If **Bugs.Out** is the list that is returned by a call to **bugs**, then the functions described in this document access the parameter estimates and other information in **Bugs.Out** for analysis in R. Similar remarks apply to the WinBUGS package. For the JAGS program, the **R2jags** provides an interface with R; the **jags** function plays the analogous role as the **bugs** function. Again, if **Bugs.Out** is the list that is returned by a call to **jagsfs**, then the functions described in this document access the parameter estimates and other information in **Bugs.Out** for analysis in R.

Contents

1. Software requirements ** Add OpenBUGs & JAGS to this table **

This document assumes that you have installed the following software:

R	Download R from CRAN (http://cran.us.r-project.org/). Installation instructions are available at the website.
WinBUGS	Download WinBUGS from http://www.mrc-bsu.cam.ac.uk/bugs/. Installation instructions are available at the website.
R2WinBUGS	This is a R package. * The easiest method for downloading this package is to run R on your computer, then run the R command: **install.packages("R2WinBUGS")**. The package will automatically be downloaded and installed. * Alternatively, go to the website that can be downloaded from http://cran.us.r-project.org/; then, click on the link to "Packages" in the left column. Search for **R2WinBUGS** in the extensive list of packages.
polspline	This is a R package. * The easiest method for downloading this package is to run R on your computer, then run the R command: **install.packages("polspline")**. The package will automatically be downloaded and installed. * Alternatively, go to the website that can be downloaded from http://cran.us.r-project.org/; then, click on the link to "Packages" in the left column. Search for **polspline** in the extensive list of packages.
jmbugs.rda	This is a R file containing the functions that are described in this document. This file can be downloaded from http://faculty.washington.edu/jmiyamot/downloads.htm. After attaching **jmbugs.rda** to the search path, you can view documentation for a function in **jmbugs.rda** by using the **doc** function that is in **jmbugs.rda**. For example, **doc(plot.chains)** will result in a display of documentation for the plot.chains function.

2. Files that are distributed with the present document

bugs.fns.pdf	This is the present document, which describes John Miyamoto's R functions for working with **bugs** output.
bugs.fns.example.txt	WinBUGS model file to create the **bugs** objects that are discussed in this document.
bugs.fns.rcode.txt	A text file containing the R code that is also shown in this document.
jmbugs.rda	A R file that contains the data and functions described in the present document. Put this file on the search path by issuing the R command: **attach("PATH/jmbugs.rda", pos=2)**, where **PATH** denotes the directory path to the location of **jmbugs.rda**.

To run the examples shown below, first run the following R command:
attach("PATH/jmbugs.rda", pos=2). In this command, replace **PATH** with a specification of the Windows directory path to the **jmbugs.rda** file.

3. Creating the bugs objects that are used to illustrate the functions in this document TOC

The functions that are explained in this document use two **bugs** objects (objects of class **bugs** that are created by a call to the **bugs** function) to illustrate the function. This section creates these two objects. The following R-code is a modification of R-code that was distributed with Lee and Wagenmakers (2010) chapter 5.

```
# Set up the attached libraries.
preferred.bugs.program = "OpenBUGS"
#set to "WinBUGS" if it is preferred.

if (preferred.bugs.program == "OpenBUGS") {
   detach("package:R2WinBUGS")
   library(R2OpenBUGS)
   } #end 'if (preferred.bugs.program == "OpenBUGS)'

if (preferred.bugs.program == "WinBUGS") {
   detach("package:R2OpenBUGS")
   library(R2WinBUGS)
   } #end 'if (preferred.bugs.program == "OpenBUGS)'

# Note that the detach commands will produce an error message if the named package is
# not on the search path, but this will not cause a problem.

# Attach the file, jmfuns.rda, to the search path.
attach("c:/mydata/jmfuns.rda")

# You may have to modify this attach command to indicate the directory where the jmfuns.rda file
# is located on your computer.

# Create the data and variables which will be modeled in the subsequent call to bugs.
# Note that this R-code is 98% identical to code in Lee & Wagenmakers (2010).

#  Create a matrix of data for the examples in this document
x <- matrix(c(10,8.04, 8,6.95, 13,7.58, 9,8.81, 11,8.33,
              14,9.96, 6,7.24, 4,4.26, 12,10.84, 7,4.82,
              5,5.68), nrow=11, ncol=2, byrow=T)
#  Sample size
n <- nrow(x) # number of people/units measured

data.names  <- list("x", "n") # to be passed on to WinBUGS

initial.vals <- list(
   chain1 = list(rho = 0, mu = c(0,0), lambda = c(1,1)),
   chain2 = list(rho = 0, mu = c(0,0), lambda = c(1,1)),
   chain3 = list(rho = 0, mu = c(0,0), lambda = c(1,1))
   ) #end list

# parameters to be monitored:
params <- c("rho", "mu", "sigma")

# The following model file was distributed with Lee & Wagenmakers (2010).
# The name for the correlation has been changed from "r" to "rho".

model.file = "
# This model file was distributed with Lee & Wagenmakers (2010).
#
# Pearson Correlation
model {
   # Likelihood
   for (i in 1:n){
```

```
      x[i,1:2] ~ dmnorm(mu[],TI[,])
   }

   # Priors
   mu[1] ~ dnorm(0,.001)
   mu[2] ~ dnorm(0,.001)
   lambda[1] ~ dgamma(.001,.001)
   lambda[2] ~ dgamma(.001,.001)
   rho ~ dunif(-1,1)

   # Reparameterization
   sigma[1] <- 1/sqrt(lambda[1])
   sigma[2] <- 1/sqrt(lambda[2])
   T[1,1] <- 1/lambda[1]
   T[1,2] <- rho*sigma[1]*sigma[2]
   T[2,1] <- rho*sigma[1]*sigma[2]
   T[2,2] <- 1/lambda[2]
   TI[1:2,1:2] <- inverse(T[1:2,1:2])

} #end of model file
" #end of the string that represents the model file.

writeLines(model.file, con = "func.expl.txt")
```

Now run the **bugs** function (the **bugs** function is slightly different in R2OpenBUGs
and R2WinBUGS, but not in ways that will affect this document.

```
N.iter   <- 1000
N.chains <- 3
N.burnin <- 0     #set N.burnin to either 0 or 500

samples <-
   bugs(
      data = data.names,
      inits = initial.vals,
      parameters.to.save = params,
      model.file =  "func.expl.txt",
      n.chains = N.chains,
      n.iter = N.iter,
      n.burnin = N.burnin,
      n.thin = 1,
      DIC = T, codaPkg = F, debug = T)

if (N.burnin == 0) bugs.0 <- samples
if (N.burnin == 500) bugs.500 <- samples
```

4. `bugs` outputs that are used as examples in the documentation of these functions

The objects, **bugs.0** and **bugs.500** can be created by running the R commands in Section 3. Of course, R and OpenBUGS or WinBUGS installed on the computer to make these commands work.

bugs.0	**bugs.0** contains output from the **bugs** function in the R2WinBUGS package. The data are 11 pairs of observations. The output contains chains of samples for the correlation **rho**, the means of the populations, **mu[1]** and **mu[2]**, and the standard deviations of the populations, **sigma[1]** and **sigma[2]**. The **bugs** analysis computed 3 chains of 1000 iterations. No burnin samples were dropped. The only difference between **bugs.0** and **bugs.500** is that **n.burnin = 0** for **bugs.0** and **n.burnin = 500** for **bugs.500**.
bugs.500	**bugs.500** contains output from the **bugs** function in the R2WinBUGS package. The data are 11 pairs of observations. The output contains chains of samples for the correlation **rho**, the means of the populations, **mu[1]** and **mu[2]**, and the standard deviations of the populations, **sigma[1]** and **sigma[2]**. The **bugs** analysis computed 3 chains of 1000 iterations. 500 burnin samples were dropped, so 500 samples were saved in each chain. The only difference between **bugs.0** and **bugs.500** is that **n.burnin = 0** for **bugs.0** and **n.burnin = 500** for **bugs.500**.

5. Documentation for JM BUGS-Related R Functions

6. brugs.chains: A function that loads MCMC chains into R that were created by using OpenBUGS and function in the BRugs package. TOC

7. ci.bugs: A function for computing the limits of an α-level credible interval from a sample of parameter values from OpenBUGS or JAGS output. TOC

8. doc: A function for displaying "on the fly" documentation for home-grown R functions

*EXAMPLES OF **doc**: The following examples assume that the **jmbugs.rda** file is on the search path in position 2. If it is in a different position, modify the code to indicate the correct position.*

```
search()
ls(2)
```

Note that for most of the objects, I have created corresponding doc objects, e.g., the
function **extract.vars** is accompanied by an object called **extract.vars.doc**.
To see the nature of the latter object, simply type:

```
extract.vars.doc
```

The purpose of the **doc** function is to display this character vector in
a more user-friendly format:

```
doc(extract.vars)
```

In general, the **doc** function expects to receive as input an R object that has a
corresponding **.doc** object, e.g., **doc(xxx)** causes **doc** to look for a
character vector named **xxx.doc**. **doc** prints **xxx.doc** to the screen in
an attractive format. Of course, the user must create **xxx.doc** before attempting
to use **doc(xxx)**.

Other examples:
```
doc(extract.chains)
doc(bugs.500)
doc(doc)
doc(jmbugs)
```

Suppose **xxx** is an object, and we want to create documentation for **xxx** (to be
saved within R).
```
xxx = c(3.14159, 2.718282)
```
There are a few simple rules for creating a **doc** object. First, **doc** takes an R
objects as argument, and looks for a **doc** object with a corresponding name.
For example:
```
doc(xxx)
```
The preceding command produces an error message. Now create **xxx.doc**.
```
xxx.doc = c("This is preliminary documentation for a dummy object xxx")
doc(xxx)
```
doc(xxx) expects **xxx.doc** to be a character vector. It automatically
combines the components of **xxx.doc** into a paragraph of output.
```
xxx.doc = c(
    "This is component 1 of the input.  Blah, blah, blah, blah.  ",
    "This is component 2 of the input.  Blah, blah, blah, blah.  ",
    "This is component 3 of the input.  Blah, blah, blah, blah.  ")
doc(xxx)
```
Therefore when creating documentation for an object **xxx**, simply enter the
documentation as a series of lines - **doc** will reformat the lines into a continuous
paragraph. Sometimes it is preferable to force the documentation to start on a
new line. In this case, remember that **\n** is the character for a new line of text
in R. For example:
```
xxx.doc = c(
    "This is component 1 of the input.  Blah, blah, blah, blah.  ",
    "This is component 2 of the input. \n",
    "This line is forced to start on a new line.  Blah, blah, blah, blah. ",
    "Blah, blah, blah, blah.  Blah, blah, blah, blah.",
    "Blah, blah, blah, blah.  \n\nThis line is forced to be preceded",
    "by a blank line.  Blah, blah, blah, blah.  Blah, blah, blah, blah.",
    "Blah, blah, blah, blah.  Blah, blah, blah, blah.")
doc(xxx)
```

9. `extract.chains`: A function for extracting multiple chains of samples for each model parameter that is monitored (saved). This function is useful when examining issues of dependence on starting value and convergence. TOC

`extract.chains` takes **`bugs`** function output as input and returns a list. Each component of the output list is a vector or a matrix for a parameter whose samples were monitored (saved) in a call to **`bugs`**. The component is a vector if the inputted **`bugs`** output was computed with only 1 chain or if the call to **`extract.chains`** specified that multiple chains are to be combined into a single vector (see **`combine.chains`** below). If the multiple chains in the **`bugs`** run are preserved (**`combine.chains = FALSE`**), then each component is a matrix whose columns are the chains for that parameter. The samples of parameters are enumerated in the order in which they were computed unless randomization is requested. Each matrix has as many columns as there were chains in the **`bugs`** run.

extract.chains (bugs.out, burnin = 0, n.thin = 1, parameters = NA, improve.names = TRUE, combine.chains = FALSE, randomize = FALSE, Warn = TRUE)

`bugs.out`	Output from a call to **`bugs`** in the R2WinBUGS package.
`burnin`	**`burnin = 0`** (default) is used to set the number of burn-in samples to be discarded. If the **`bugs`** call was computed with a non-zero burnin specification, then these samples will have been discarded before **`extract.chains`** can access these samples. Therefore if **`extract.chains`** is called with **`burnin`** > 0, then these samples are discarded in addition to any that were discarded as burnin samples in the call to **`bugs`**. In other words, if the call to **`bugs`** specified a burnin of 10 samples, and the call to **`extract.chains`** specified a burnin of 5 samples, then the output of **`extract.chains`** will contain chains that have had an initial 15 sample removed. Typically, **`extract.chains`** should be used with a **`bugs`** call that specifies a burnin of 0 because **`extract.chains`** can drop any burnin samples after examining all samples for poor mixing in the initial samples.
`n.thin`	Indicates the rate at which samples are saved. If **`n.thin = 1`**, then all samples are saved (not counting the discarded burnin samples. Note that the discarding of samples by **`n.thin`** occurs after discarding the initial **`burnin`** samples and after thinning has occurred in the call to **`bugs`**. In other words, if the call to **`bugs`** specified a thinning rate of 3, and if the call to **`extract.chains`** specified a thinning rate of 2, then the output of **`extract.chains`** will contain chains that retain every 6-th sample (6 = 3*2). Typically, the user will specify a thinning rate of 1 for either the call to **`bugs`** or the call to **`extract.chains`**, but not both.
`parameters`	If **`parameters = NA`** (default), then all saved parameters are extracted. Otherwise, set **`parameters`** to a character vector that names the parameters whose samples are to be extracted. Use **`show.bugs(Bugs.Out)`** or **`dimnames(Bugs.Out$sims.array)[[3]]`** to see the names of the parameters whose samples were saved in the call to **`bugs`** (substitute the name of the **`bugs`** output for **`Bugs.Out`**).

improve.names	**improve.names = TRUE** (default) uses **make.names.jm** to create names for list components that are syntactically valid in R. If **improve.names = FALSE**, then the parameter names in the **bugs** output are retained without changes. E.g., parameter name like **mu[1]**, is a legal name in BUGS but not in R; if **improve.names = TRUE**, then the list component that contains the chains for **mu[1]** is named **mu.1**.
combine.chains	**combine.chains = FALSE** is the default. If **$n.chains > 1 & combine.chains = FALSE**, then each component of the output list is a matrix with **$n.chains** columns and **$n.iter - burnin** rows. If **$n.chains = 1** or if **combine.chains = TRUE**, then each component of the output is a vector. In the case where **$n.chains > 1** and **combine.chains = TRUE**, the separate chains of samples are combined into a single vector of samples (after discarding burnin samples from each chain).
randomize	**randomize = FALSE** is the default. If the output vectors or matrices are in the order of **$sims.array** (after discarding burnin samples. If **randomize = TRUE**, then the rows of the output matrix (multiple chains), or the elements of the output vector (1 chain) are randomly reordered.
Warn	**Warn = TRUE** (default) gives the user feedback whenever any burnin samples were dropped either in the original **bugs** call or in the call to **extract.chains**. If **Warn = FALSE**, the feedback is suppressed.

EXAMPLES OF **extract.chains***: The following examples assume that* **bugs.0** *and* **bugs.500** *have been created and are in a file on the search path. If they are not, see Section 4 for the code that is used to create these objects.*

The main use of **extract.chains** is to extract multiple chains of samples for the parameters
that were saved from a WinBUGS model.

```
b0.chains <- extract.chains(bugs.0)
is.list(b0.chains)
names(b0.chains)
is.matrix(b0.chains$rho)
head(b0.chains$rho) #see 3 chains for rho
head(b0.chains$mu.1)#see 3 chains for mu.1
```

This output shows that each component of **b0.chains** is a matrix. Each column of a matrix
is a chain of samples for the designated parameter. There are three columns in each matrix
because there were three chains in the **bugs** run that create **bugs.0**.

10. extract.vars: A function for creating a dataframe of samples from parameters of a model. Each variable in the dataframe combines the chains of samples for a parameter. The function allows the user to control the manner of combination, i.e., the user specifies the dropping of burnin samples, thinning of the samples, or randomization of the samples. TOC

extract.vars extracts one or more posterior distributions from **bugs** output. If **Bugs.Out** is the output of a call to **bugs** within R, then **Bugs.Out$sims.array** is an array that contains the WinBUGS results for all of the saved parameters. **extract.vars** extracts the posteriors for one or more saved parameters from output like **Bugs.Out**. If the samples were computed as 2 or more chains,

extract.vars combines these chains into a single vector after (optionally) removing an initial set of burnin samples from each chain. **extract.vars** differs from the **Bugs.Out$sims.list** component of **bugs** output insofar as **Bugs.Out$sims.list** randomizes the order of the samples so that it is not possible to identify and remove burnin samples from **Bugs.Out$sims.list** (the burnin samples have to be eliminated within WinBUGS before transferring the samples to R). **extract.vars** gives the user a choice whether to randomize (default) or not randomize the order of samples within a vector of parameter samples. By default, the **extract.vars** output is a vector if the samples for only 1 parameter are extracted, and the output is a dataframe with k variables if the samples for k > 1 parameters are extracted.

Use the **extract.chains** function to extract the individual chains of samples for a parameter (usually to examine issues of convergence). **extract.vars** uses **extract.chains** to extract the chains of samples, so you should look at **extract.chains** to see exactly how the samples were extracted.

```
extract.vars(bugs.out, parameters = NA, burnin = 0, n.thin = 1,
    new.names = NA, randomize = TRUE)
```

bugs.out	Output from a call to **bugs** in the R2WinBUGS package.
parameters	**parameters = NA** (default) extracts samples for all parameters that were saved in **bugs.out**. **parameters** can be set to a vector of variable names if only a subset of the variables are to be extracted.
burnin	**burnin = 0** (default) is used to set the number of burn-in samples to be discarded. Note that these samples are discarded from the initial segment of each chain prior to combining them into a vector.
.thin	n.thin indicates the thinning rate. n.thin = 1 indicates no thinning.
new.names	**new.names = NA** (default) if the R-legal versions of the WinBUGS parameter names are to be used as column names in the output dataframe. Set **new.names** to a character vector of names if alternative names are preferred.
randomize	**randomize = TRUE** (default) if the samples are randomly reordered. If **randomize = FALSE**, then the samples in a vector are ordered **[chain 1 - burnin][chain 2 - burnin]...[chain k - burnin]**

EXAMPLES OF **extract.vars***:*

```
# The next code shows that the default operation of extract.vars creates a dataframe. Each variable in the
# dataframe is as long as the sum of the lengths of the three chains that were sampled in the creation of
# bugs.0, i.e., 3,000 samples in length.

v.frame <- extract.vars(bugs.0)
is.data.frame(v.frame)
head(v.frame)
length(v.frame[,1])

# Notice that the parameter names in v.frame are not the same as the parameter names in the bugs
# output, bugs.0.
names(v.frame)
dimnames(bugs.0$sims.array)[[3]]
```

extract.vars changes the parameter names that are given by WinBUGS to names that conform
to R rules for object names. It is possible to use the **extract.chains** function to retain the
WinBUGS parameter names, but this is rarely useful.

```
# The next example shows that you can extract the samples for just one parameter.
names(v.frame)
rho.only <- extract.vars(bugs.0, parameters = "rho")
rho.only[1:25]
is.data.frame(rho.only)
is.vector(rho.only)

# The next example shows that you can extract the samples for a subset
# of the parameters.
names(v.frame)
mu.1.2 <- extract.vars(bugs.0, parameters = c("mu.1", "mu.2"))
is.data.frame(mu.1.2)
head(mu.1.2)
```

The next example shows that you can drop a burnin sample from the beginning of each chain of
samples prior to storing the samples in a dataframe

```
drop.500 <- extract.vars(bugs.0, burnin = 500)
head(drop.500)
length(drop.500[,1])
length(v.frame[,1])
```

The **bugs.0** output was created with 1000 iterations, 3 chains and no burnin.
Therefore there are 1000 x 3 = 3000 samples for each parameter that was saved
in **bugs.0**. The **drop.500** dataframe was created by dropping 500 samples
from each chain prior to combining them into individual vectors of parameter
samples. Thus **drop.500** has variables of length 500 x 3 = 1500. Recall
that the **bugs.500** output was created by having WinBUGS drop a burnin sample
of 500 from each chain. Therefore **bugs.500** should have the same samples
as **drop.500**, except for the fact that the variables in **drop.500** were randomized
in a different order from the samples in **bugs.500**. This is illustrated by the
following code.

```
burn500.frame <- extract.vars(bugs.500)
```

The next series of commands demonstrate that **burn500.frame** (derived from
bugs.500 which was created by dropping a burnin of 500 samples within
WinBUGS, and **drop.500** which was created by using the **extract.vars** function
to drop an initial 500 samples from the chains in **bugs.0** that was produced with
a burnin of 0 are equivalent datasets.

```
names(burn500.frame)
names(drop.500)
length(burn500.frame[,1])
length(drop.500[,1])
sapply(burn500.frame, mean)
sapply(drop.500, mean)
sapply(burn500.frame, sd)
sapply(drop.500, sd)
```

The next two statements show that **burn500.frame** and **drop.500** are identical except
for the random order in which the vectors of samples have been saved.

```
all(sort(burn500.frame$rho) == sort(drop.500$rho))
all(burn500.frame$rho == drop.500$rho)
```

The final example shows that if burnin samples are dropped from both the **bugs** run and
the **extract.vars** call, then a warning is issued that states that burnin samples have
been dropped twice. (Remember that **bugs.500** was run with a burnin of 500 samples.

```
drop.twice = extract.vars(bugs.500, burnin = 100)
```

11. from.mcmc.chains: A function for converting a list of class mcmc.list to a list of chains with one component per parameter. TOC

12. make.names.jm: A function that makes reasonably attractive R-legal names from variable names in bugs outputs (some BUGS legal names are not R-legal names).

The **make.names.jm** function is called internally by the **extract.chains** function, so an understanding of this function is not critical to this document. Nevertheless, a brief explanation will be given here. R has a function, **make.names**, for creating R-legal names from a character vector, some of whose elements may not be legal names for R objects. In some cases, **make.names** will create a name from a **bugs** output name that is not attractive. **make.names.jm** is designed to create more attractive names than would be created by **make.names**. Here is an example.

EXAMPLE of **make.names.jm***:*

```
dimnames(bugs.0$sims.array)
dimnames(bugs.0$sims.array)[[3]]

make.names(dimnames(bugs.0$sims.array)[[3]])
make.names.jm(dimnames(bugs.0$sims.array)[[3]])
```

Note that the names created by **make.names.jm** are slightly more attractive.

13. o.type: A function that shows what type of object an inputted object is.

o.type tests for the mode, factor status and other classifications of an object.

o.type(x, variables = FALSE, sorted = TRUE)

x is the object to be tested. **variables = TRUE** only has an effect if **x** is a dataframe. In this case, **variables = TRUE** causes **o.type** to list information about every variable in **x**. **sorted = TRUE** causes the output to printed with the TRUE attributes first; otherwise the attributes are always printed in the same order. **sorted = TRUE** has no effect if **x** is a dataframe and **variables = TRUE**.

EXAMPLES of **o.type**:

```
o.type(bugs.0)       #bugs.0 is a list
o.type(rho.only)     #rho.only is a numeric vector
o.type(v.frame)      #v.frame is a dataframe (both a list & a dataframe)
o.type(v.frame, var = T)   #information is returned about each variable in v.frame
```

14. `plot.chains`: A function that plots the chains of samples for a parameter (useful when examining dependence on starting values and mixing of chains)

TOC

`plot.chains` plots the chains for parameters that were saved in a call to **`bugs`**.

plot.chains(param, bugs.out = NA, xlim.f = NA, ylim.f = NA, legend = FALSE,
add.labels = TRUE, cex.lab = 1.5, ...)

ARGUMENTS:

`param`	**`param`** is one of three things: If **`bugs.out`** is the output to a call to **`bugs`**, then **`param`** must be the name of a parameter that was saved in this output. This name can be specified either by means of the name given to it by WinBUGS, or by the R-legal name that was created by the **`extract.chains`** function. If **`bugs.out`** is NA, then **`param`** is either a matrix of chains or it is a list whose sole component is a matrix of chains.
`bugs.out`	**`bugs.out`** is either NA or it is the output from a call to **`bugs`**. If **`bugs.out`** is NA, then **`param`** must be either a matrix of chains or a list whose sole component is a matrix of chains (as in the output of **`extract.chains`**. If **`bugs.out`** is the output from a call to **`bugs`**, then **`param`** must be the name of a parameter that was saved in this output.
`xlim.f`	**`xlim.f = NA`** (default) causes **`plot.chains`** to plot all of the iterations for the chains. Alternatively, set **`xlim.f`** to the lower and upper bounds on the x-axis, e.g., **`xlim.f = c(200, 300)`** to plot the iterations from 200 to 300.
`ylim.f`	**`ylim.f = NA`** (default) means that **`plot.chains`** determines the limits on the y-axis. Alternatively, set **`ylim.f`** to the lower and upper limit on the y-axis, e.g., **`ylim.f = c(.2, .8).`**
`legend`	**`legend = FALSE`** (default) then no legend is plotted. If **`legend = TRUE`**, a legend is printed across the top of the graph to show the correspondence between the line colors and chains.
`add.labels`	**`add.labels = TRUE`** (default) to have **`plot.chains`** generate the axis labels. If FALSE, the axis labels are omitted.
`cex.lab`	**`cex.lab`** controls the character size for the axis labels. The setting **`cex.lab = 1.5`** (default) generally looks good.

EXAMPLES OF **`plot.chains`***:*

First, specify a plot by naming a parameter that was saved in **`bugs.0`**.

```
plot.chains(param = "mu[1]", bugs.out = bugs.0)
```

The preceding example named the parameter by its WinBUGS name.
Note that the parameter can also be named with its corresponding R-legal name.

```
plot.chains(param = "mu.1", bugs.out = bugs.0)
```

We can also input a parameter that was previously extracted with **`extract.chains`**

```
params.0 <- extract.chains(bugs.0)
plot.chains(param = params.0$sigma.1)
```

Of course, we can suppress the ugly y-label in the preceding plot, and add a better looking one.
```
plot.chains(param = params.0$sigma.1, add.labels = FALSE)
mtext("Iterations", side = 1, cex = 1.5, line = 2.5)
mtext("Samples of sigma.1", side = 2, cex = 1.5, line = 2.5)
mtext("Check Convergence of 3 Chains for sigma.1", side = 3, cex = 1.5, line = 1)
```
We can also control the range of iterations in the display.
```
plot.chains(param = params.0$sigma.1, xlim = c(1, 100))
plot.chains(param = params.0$sigma.1, xlim = c(101, 301))
```
We can add a legend to distinguish the chains.
```
plot.chains(param = params.0$sigma.1, xlim = c(1, 200), legend = TRUE)
```
We can alter the range of the y-axis.
```
plot.chains(param = params.0$sigma.1, xlim = c(1, 200),ylim = c(0, 8), legend = TRUE)
```

15. `res.param`: A function for plotting the distribution of a sample from the posterior distribution of a parameter. TOC

The function **`res.param`** computes basic statistics for the prior or posterior distribution of a parameter in a WinBUGS analysis. By default, it plots the density of the parameter. Upon request, it displays credible intervals and location statistics on the graph.

```
res.param(param.post, plot.dist = TRUE,
   method.density = c("density", "logspline")[1],
   method.ci = c("HDI", "equal.tails")[1],
   output = c("stats", "density")[1],
   stats.which = c("mean", "median", "mode", "conf"),
   level = 0.95, conf.pct = NA, show.conf = FALSE, show.stats = TRUE,
   digits.f = 3, lwd.f = 3, cex.stats = 1.15,
   xlab.f = paste("Samples of", deparse(substitute(param.post))),
   ylab.f = "Probability Density", xlim.f = NA, stats.ht = 0.9)
```

Arguments

`param.post`	A numeric vector whose distribution is to be displayed. Typically, **`param.post`** is the vector of simulated samples of a parameter that results from a WinBUGS run.
`plot.dist`	**`plot.dist = TRUE`** (default) to produce a density plot that shows the distribution of **`param.post`**. If **`plot.dist = FALSE`**, no plot is produced.
`method.density`	**`method.density`** determines the method by which the density of **`param.post`** is approximated. The default method is **`density`** which uses the **`density`** function in the **`stats`** package. If **`method.density`** is set to **`logspline`**, then the **`logspline`** function in the **`polspline`** package is used to estimate the density of **`param.post`**. See ?density and ?logspline for descriptions of these two methods.
`method.ci`	**`method.ci`** determines the type of credible interval. The default is **`HDI`** (highest density interval), but a credible interval with equal probability tails can be selected (method.ci = **`equal.tails`**).
`output`	**`output`** controls whether to return a vector of statistics (output = "stats"), or the xy coordinates of the density (output = "density").

lwd.f	**lwd.f** controls the weight of the line in a density plot (default **lwd.f = 3**).
stats.which	The statistics selected by **stats.which** are outputed as a vector by **param.post**; optionally, the values of these statistics are printed on the density plot by setting **show.stats = TRUE**. If **show.stats = FALSE**, the statistics selected by **stats.which** are outputted by **param.post** but they are not plotted on the density plot. **stats.which** can be set to any or all of the statistics, **mean**, **median**, **mode** or **conf**. If **stats.which** includes **conf**, the lower and upper bounds of a credible interval are computed and optionally printed on the density plot. The level of the credible interval is set by **level** (choose any number between 0 and 1.0 - the default value is .95). A pre-4/5/2011 version used an argument called **conf.pct** to set the level of the credible interval. The **conf.pct** argument has been retained in this function to make the function compatible with older code, but in post-4/5/2011 uses of this function, use **level** to indicate the level of the credible interval.
xlab.f, ylab.f	The axis labels to be displayed on the X and Y axes are controlled by **xlab.f** and **ylab.f**.
xlim.f	The range of values displayed on the X axis is controlled by **xlim.f** - if **xlim.f = NA** (default), the default X axis range is used.
digits.f	**digits.f** specifies the number of digits to the right of the decimal place to display in the statistics. The default is 3 digits; if desired, different numbers of digits can be specified for each statistics. WARNING: If specifying separate numbers of digits for each statistic, you need to specify the number of digits for each of the statistics, **mean**, **median**, **mode** and **conf**, even if only a subset of these statistics is selected. **stats.ht** controls the height at which the statistics are printed on a plot, expressed as a proportion of the maximum height of the density plot. The default, **stats.ht = .90**, indicates that the stats are vertically centered around 90% of the maximum height of the density plot.

16. show.bugs: A function for quickly displaying information about a bugs output.

TOC

show.bugs displays basic information about **bugs** output, specifically, it shows the WinBUGS names for the parameters, and the number of iterations, the number of burnin samples, and the number of chains of samples.

show.bugs(bugs.out)

bugs.out	Output from a call to **bugs**.

EXAMPLES OF **show.bugs***:*

```
show.bugs(bugs.0)
show.bugs(bugs.500)
```

17. to.mcmc.list: A function that converts a list of chains to a list of class mcmc.list.

TOC

18. Code for JM BUGS-Related R Functions

19. Code for doc function

```
doc <- function(x, File = , complete = FALSE, suppress.o.type = FALSE) {
#  o.name is the R name of the x object.
o.name <- deparse(substitute(x))
#  These flags are useful later in the function.  Easiest to set them to FALSE here.
has.doc.extension <- FALSE; doc.object.exists <- FALSE
#  Following internal function, named reformat, reformats a vector of strings in order
#  to take advantage of string wrapping.
reformat <- function(cc) {
#        Each component of tmz is a string of text that is to be separated from preceding
#        and following components by a carriage return.
    tmz <- strsplit(paste(cc, collapse= ), \\\n)[[1]]
#        tma is like tmz except any single leading blanks are removed.
    tma <- NULL
    for (i in 1:length(tmz)) {
        if (nchar(tmz[i]) > 1 & substr(tmz[i], 1, 1) ==  )
            tma <- c(tma, substr(tmz[i], 2, nchar(tmz[i]))) else
            tma <- c(tma, tmz[i])
        } #end for (i in 1:length(tmz))
#        tmb is like tma except that each component has been broken into separate lines that
#        fit nicely on the screen.
    tmb <- sapply(tma, function(x) {
        strwrap(x, width=0.9 * getOption("width") - 8) })
    tmb
} #end def of reformat function

#  This code takes care of the case where x is an object whose name has the form, yyy.doc.
#  name.parts0 is a list of length 1.  Its only component is a character vector of o.name
#  that has been split on ".".  Note that name.parts0 may be redundant.  If you do a revision,
#  perhaps you can replace it with name.parts1.
name.parts0 <- strsplit(o.name, split = \\.)

if (mode(x) == character && length(name.parts0) == 1) {

    name.parts1 <- unlist(name.parts0)

    if (length(name.parts1) > 1 &
        tolower(name.parts1[length(name.parts1)]) == doc) {
        has.doc.extension <- TRUE

#           Prints o.type to screen.
        obj0.name <- o.name
        obj.type <- o.type(x)
        if (exists(obj0.name) & !suppress.o.type) {
            cat(paste("", obj0.name,
                ", is the following type of object:\n", sep=), file = File)
            print(obj.type)
#           Print function arguments to screen if it is a function.
            if (obj.type[fn]) {
                cat(\n, file = File, append = TRUE);
                print(args(get(obj0.name))) } #end if
```

```
        } #end if (exists(obj0.name))
      if (!suppress.o.type) cat(\nDocumentation:\n, file = File)
#       Reformats the input to make use of screen wrap.
      tmb <- reformat(x)
#       The next for loop writes tmb to the screen.
      for (i in 1:length(tmb)) cat(tmb[[i]], sep=\n, file = File)
#       The next code cleans up objects.
      rm(tmb)
      } #end if (length(name.parts1) > 1 & name.parts1[length(name.parts1)] == doc)
  } #end if (mode(x) == character && length(name.parts0) == 1)
# This if takes care of the case where an object of the form o.name.doc exists.
if (exists(paste(o.name, .doc,sep=))) {
  doc.object.exists <- TRUE
# Print location of object to screen
  find.locs <- find(o.name)
  o.locs <- paste(find.locs, collapse = ", ")
  find.doc.locs <- find(paste(o.name, .doc,sep=))
  doc.locs <- paste(find.doc.locs, collapse = ", ")

  if (length(find.locs) > 1) {
    cat(paste("Locations of ", o.name, " on the search path:  ",
      o.locs, "\n", sep = ""))
    if (find.locs[1] != find.doc.locs[1]) {
      cat(paste("Locations of ", paste(o.name, .doc,sep=),
        " on the search path:  ", doc.locs, "\n", sep = ""))
      cat(paste("WARNING: ", o.name, " and ", paste(o.name, .doc,sep=),
        ", are not in the same location on the search path.\n",
        "Check that the object and documentation pertain to the same object.\n",
        sep = ""))
      } #end if (find.locs[1] != find.doc.locs[1])
    } else { #end if (length(find.locs) > 1)
    cat(paste("Location of ", o.name, " on the search path:  ",
      o.locs, "\n", sep = ""))
    } #end of else for if (length(find.locs) > 1)

#     Prints o.type to screen.
  obj0.name <- o.name
  obj.type <- o.type(get(obj0.name))
  if (!suppress.o.type) {
    cat(paste("", obj0.name,
      ", is the following type of object:\n", sep=), file = File)
    print(obj.type)
    } #end if (!suppress.o.type)
# Print function arguments to screen if it is a function.
  if (obj.type[fn]) {
          cat(\n, file = File); print(args(get(obj0.name))) } #end if
  if (! suppress.o.type) cat(\nDocumentation:\n, file = File)
#     reformat x to make use of screen wrap.
  tmb <- reformat(get(paste(o.name, .doc,sep=)))
#     The next for loop writes tmb to the screen.
  for (i in 1:length(tmb)) cat(tmb[[i]], sep=\n, file = File)
#     The next code cleans up objects.
  rm(tmb)
  } #end if (exists(paste(o.name, .doc,sep=)))
```

```
#  This long if creates a name for the type of object that x is.
if (is.data.frame(x)) x.type <- dataframe else
    if (is.list(x)) x.type <- list else
       if (is.factor(x)) x.type <- factor else
          if (is.numeric(x)) x.type <- numeric variable else
             if (is.logical(x)) x.type <- logical variable else
                if (is.character(x)) x.type <- string variable else
                   if (is.function(x)) x.type <- function

if (!is.null(attr(x, doc))) {
#  Prints o.type to screen.
    obj0.name <- o.name
    cat(paste("", obj0.name, ", is the following type of object:\n", sep=),
       file = File)
    obj.type <- o.type(get(obj0.name))
    print(obj.type)
    cat(\nDocumentation:\n, file = File)
#  Print function arguments to screen if it is a function.
    if (obj.type[fn]) {
              cat(\n, file = File); print(args(get(obj0.name))) } #end if
#  reformat x to make use of screen wrap.
    tmb <- reformat(get(paste(o.name, .doc,sep=)))
#  The next for loop writes tmb to the screen.
    for (i in 1:length(tmb)) cat(tmb[[i]], sep=\n, file = File)
#  The next code cleans up objects.
    rm(tmb)
#  The next if writes out factor levels for x, if they exist.
    if (is.factor(x)) {
       lv <- matrix(c(level=levels(x), 1:length(levels(x))),
          ncol=2, dimnames = list(rep(     , length(levels(x))),
          c(Level, Numeric Value)))
       cat("\nThe variable, ", o.name,
          ", is a factor with levels:\n", sep=, file = File)
       print(lv, quote=FALSE)
       } #end if.factor(x))
    } else {
#        The next clause writes a message to the user in the case where x has a null doc attribute.
    if (!exists(paste(o.name, .doc,sep=)) & !has.doc.extension
       & !doc.object.exists)
          cat("\nThe ", x.type, ", ", o.name,
             ", has a NULL doc attribute ",
             "and there is no ", o.name, ".doc object.\n", sep=, file = File)
    } #end else of if (!is.null(attr(x, doc)))
#  The next if writes out documentation for individual variables in the case where
#  x is a dataframe and complete == TRUE.
if (is.data.frame(x) & complete)
    if (any(sapply(x, function(y) !is.null(attr(y, doc)) ))) {
       var.mat <- matrix(c(VARIABLE, names(x),
          LABELS, sapply(x, function(x) attributes(x)$doc)),
          ncol=2)
       max.char <- max(nchar(var.mat[,1]))
       left.indent <- paste(rep( , max.char+2), collapse=)

       var.info <- NULL
```

```
        for (i in 1:length(var.mat[,1])) {
        var.mat[i,1] <- paste(var.mat[i,1],
           paste(rep( , max.char - nchar(var.mat[i,1])),
           collapse=),   , sep=)
        wrap.label <- strwrap(var.mat[i,2],
           width = getOption("width") - max.char - 12)
        if (is.null(wrap.label) | length(wrap.label) == 0) {
           v.tmp <- c(var.mat[i,1], (no documentation))
           } else {
           v.tmp <- c(var.mat[i,1], wrap.label[1])
           if (length(wrap.label) > 1)
              v.tmp <- rbind(v.tmp,
                 cbind(left.indent, wrap.label[2:length(wrap.label)]))
           } #end if (is.null(wrap.label)) v.tmp <- var.mat[i,]
        var.info <- rbind(var.info, v.tmp)
           } #end for (i in 1:var.mat[,1])
        cat(\nVariables in the dataframe: , o.name, \n, file = File)
        cat(paste(
           apply(var.info, 1, function(x) paste(x, sep=, collapse=)),
           sep=), sep=\n, file = File)
        cat(\n, file = File)

        } else { #start else of if (any(sapply(x, function(y) !is.null( ...
           cat("\nThe ", x.type, ", ", o.name,
              ", has no variables with non-NULL doc attributes.\n", sep=,
              file = File)
        } #end if (any(sapply(x, function(y) !is.null(attr(y, doc)) )))
cat(\n, file = File)
} #end def of doc function
```

20. Code for `extract.chains`

```
extract.chains <- function(bugs.out, burnin = 0, n.thin = 1,
    parameters = NA, improve.names = TRUE,
    combine.chains = FALSE, randomize = FALSE, Warn = TRUE) {

params.all <- dimnames(bugs.out$sims.array)[[3]]
#  if !all(is.na(parameters)) and parameters are not a subset of params.all, then
#  the procedure stops with an error message.
if ((!all(is.na(parameters))) && !all(parameters %in% params.all))
    stop(
        "At least some of the parameters names in the parameters argument\n",
        "do not correspond to parameter names in bugs.out.  \n\n",
        "Parameter names in bugs.out:\n\n",
        strwrap(paste(params.all, collapse = ", "))
        ) #end stop
#  The next if sets param.names to the desired parameters to be extracted.
if (length(parameters) == 1 && is.na(parameters))
    param.names <- params.all else
    param.names <- parameters

n.names <- length(param.names)
#  param.index contains the index numbers for the parameters in param.names
param.index <- match(param.names, params.all)
```

```
n.samples <- bugs.out$n.iter - bugs.out$n.burnin
n.chains <- bugs.out$n.chains
#  Next: Warning if the original bugs call specified a burnin > 0 and the current call
#  to extract.chains also specified a burnin > 0.  In this case, the total burnin is the sum
#  of the two separate burnins.  Usually you would only need to specify a burnin on the
#  original bugs call or the extract.chains call, but not both, so this warning lets the
#  user know that something odd is happening.
if (Warn & burnin > 0 | bugs.out$n.burnin > 0) { warning( paste(
    "The original bugs call specified a burnin of ", bugs.out$n.burnin,
    " samples.\n",
    "The current call to extract.chains specified a burnin\nof ",
    burnin, " samples.  ",
    "Consequently, a total of ", bugs.out$n.burnin + burnin, " samples were\n",
    "discarded from each chain of samples in the original\n",
    "bugs call.\n"
    , sep = "") ) #end paste and end warning
    } #end if (n.samples < bugs.out$n.iter)
#  Next: Create R-legal names from the WinBUGS parameter names.  Names like x[2] are legal
#  in WinBUGS but not in R.
if (improve.names)
    new.names <- make.names.jm(param.names) else
    new.names <- param.names
#  Create an ordering variable for the case where the simulations will NOT be saved in a random order.
if (!randomize & !combine.chains) new.order <- 1:trunc((n.samples - burnin)/n.thin)
if (!randomize & combine.chains)
    new.order <- 1:(n.chains*trunc((n.samples - burnin)/n.thin))
#  Create an ordering variable for the case where the simulations will be randomized but they will
#  not be combined into a single vector (separate chains will be retained).
if (randomize & !combine.chains) {
    iter.id <- 1:(n.samples - burnin)
    random.var <- runif(trunc((n.samples - burnin)/n.thin))
    new.order <- order(random.var)
    warning(
        "The call to extract.chains specifies that the chain elements be randomly\n",
        "reordered without combining the chains into a single vector.  This is odd\n",
        "because the main reason for retaining the separate chains of samples\n",
        "is that one wants to examine the chains in the order that they were \n",
        "computed.  Check that you really want to preserve the separate chains\n",
        "while randomizing their orders.\n" )
    } #end if (randomize & !combine.chains)
#  Create an ordering variable for the case where the simulations will be randomized and the and
#  they will be combined into a single vector (separate chains will be lost).
if (randomize & combine.chains) {
    n.chains <- bugs.out$n.chains
    n.case <- n.chains*(trunc((n.samples - burnin)/n.thin))
    iter.id <- 1:n.case
    random.var <- runif(n.case)
    new.order <- order(random.var)
    } #end if (randomize & !combine.chains)

tmL <- NULL
for (i in 1:length(param.index)) {
    tmL <- c(tmL, list(NA))
```

```
#       m0 is the matrix of chains for the i-th parameter
    m0 <- bugs.out$sims.array[,, param.index[i]]
    if (is.matrix(m0)) m1 <- m0 else m1 <- matrix(m0, ncol = 1)
    m.noburn <- m1[(burnin + 1):n.samples, ]

    if (n.thin <= 1) m.keep <- m.noburn
    if (n.thin > 1) {
        N.noburn <- length(m.noburn[,1])
        index.keep <- n.thin * seq(from = 1, to = N.noburn/n.thin, by = 1)
        m.keep <- m.noburn[index.keep ,]
    } #end if (n.thin > 1)

    if (combine.chains) {
        tm.mat <- as.vector(m.keep)[new.order]
    } else { #end if (combine.chains)
        tm.mat <- m.keep[new.order, ]
        if (is.matrix(tm.mat))
            dimnames(tm.mat) <- list(NULL, paste("chain", 1:ncol(tm.mat), sep="."))
    } #end else

    tmL[i] <- list(tm.mat)

    } #end for

names(tmL) <- new.names
return(tmL)
    } #end def of extract.chains function
```

21. Code for `extract.vars`. TOC

```
extract.vars <- function(bugs.out, parameters = NA, burnin = 0, n.thin = 1,
    new.names = NA, randomize = TRUE) {

#  extract the samples for all variables from bugs.out.
vars.ini <- extract.chains(
              bugs.out = bugs.out, burnin = burnin, n.thin = n.thin,
              combine.chains = TRUE, randomize = randomize,
              improve.names = FALSE)
#  By default, extract.chains changes the variable names from WinBUGS names to legal R names.
#  The following code allows us to specify parameters using either the WinBUGS
#  names or the R names.
if (all(is.na(parameters))) vars <- names(vars.ini) else
    vars <- make.names.jm(parameters)
#  Next: Check that parameters actually specifies variable names in vars.ini
if (!all(vars %in% names(vars.ini))) {
    bugs.out.name <- deparse(substitute(bugs.out))
    BUGS.names <- dimnames(bugs.out$sims.ar)[[3]]
    R.names <- names(vars.ini)
    cmt.parameters <- paste(
        "The function call requested the extraction of the following variables:\n",
        paste(\", paste(parameters, collapse = \", \"), ", sep=""), "\n",
"\n",
        "The WinBUGS names for the variables in ", bugs.out.name, " are:\n",
        paste(\", paste(BUGS.names, collapse = \", \"), ", sep = ""), "\n",
```

```
"\n",
       "The R names for these variables are:\n",
       paste(\", paste(R.names, collapse = \", \"), ", sep = ""), "\n",
"\n",
       sep = "")
   stop(
       "At least one of the variable names specified\n",
       "by parameters is not present in the WinBUGS output.  \n",
       cmt.parameters,
       "parameters must be specified either as NA or in terms \n",
       "of the WinBUGS names or R names.")
   } #end if (!all(vars %in% names(vars.ini)))

out.0 <- vars.ini[vars]

if (length(vars) > 1) {
   out <- data.frame(out.0)
   if (!is.na(new.names)) names(out) <- new.names
   } #end if (length(vars > 1))

if (length(vars) == 1) {
   out <- unlist(out.0)
   } #end if (length(vars == 1))

return(out)
   } #end def of extract.vars function
```

22. Code for `make.names.jm`

```
make.names.jm <- function(names.jm, unique.jm = TRUE,
   rm.double = TRUE, allow_.jm = TRUE) {

n.vec.0 <- make.names(names.jm, unique = unique.jm, allow_ = allow_.jm)
n.vec.1 <- NULL
for (i in 1:length(n.vec.0)) {
   rr <- unlist(strsplit(n.vec.0[i], ""))

   ss <- NULL
   if (rm.double) {
      for (j in 1:length(rr)) {
         if (j == 1 & rr[1] != ".") ss <- rr[1]
         if (j > 1 && rr[j] != ".") ss <- c(ss, rr[j])
         if (j > 1 && rr[j] == "." && rr[j-1] != ".") ss <- c(ss, rr[j])
         } #end for (j in 1:length(rr))
      } else { #end if (!rm.double)
      ss <- rr
      } #end else

   c.nu <- NULL   #c.nu will become the new character vector
   c.end <- TRUE  #c.end is a flag for whether any non-. have been found.
   for (k in length(ss):1) {
      if (ss[k] != "." | !c.end) {
         c.end <- FALSE
         c.nu <- c(ss[k], c.nu)
```

```
        } #end if (ss[k] == "." & c.end)
    } #end for (k in length(ss):1)

n.vec.1 <- c(n.vec.1, paste(c.nu, collapse = ""))

} #end for (i in 1:length(n.vec.0))

out <- make.names(n.vec.1, unique = unique.jm, allow_ = allow_.jm)
return(out)
    } #end def of make.names.jm function
```

23. Code for `o.type` TOC

```
o.type <- function(x, variables=FALSE, sorted=TRUE) {
if (is.data.frame(x) & variables)
    out1 <- t(sapply(x, function(y) c(
        scalar=(is.numeric(y) & !is.factor(y) & length(y) == 1),
        vector= (is.vector(y) & length(y) > 1),
        numeric=is.numeric(y), factor=is.factor(y), char=is.character(y),
        logical=is.logical(y), NULL = is.null(y), zero=(length(y) == 0))))

out2 <- c(
        scalar=(is.numeric(x) & !is.factor(x) & length(x) == 1),
        vector= (is.vector(x) & length(x) > 1), matrix=is.matrix(x), array=is.array(x),
        list=is.list(x), d.frame = is.data.frame(x),
        numeric=is.numeric(x), factor=is.factor(x), char=is.character(x),
        logical=is.logical(x), NULL = is.null(x),
        all.NA = if (is.logical(x) | is.character(x) | is.numeric(x))
            all(is.na(x)) else FALSE,

        zero=(length(x) == 0), fn=(is.function(x)))

if (sorted) out2 <- c(out2[out2], out2[!out2])

if (is.data.frame(x) & variables) {
    main.out <- list(out2, out1)
    names(main.out) <- c(deparse(substitute(x)), variables)
    } else { #end if (variables)
    main.out <- out2
    } #end of else for if (variables)

return(main.out)
} #end of function definition
```

24. Code for `plot.chains` TOC

```
plot.chains <- function(param, bugs.out = NA,
    xlim.f = NA, ylim.f = NA, legend = FALSE,
    add.labels = TRUE, cex.lab = 1.5, ...) {
#Subp.ini  ----hidden parameter settings --------end hidden----
#  First extract the name of the parameter to be plotted.  Later param will be redefined
#  in the case where param is a list, so we need to extract this parameter name before
#  the redefinition of param.
if (is.character(param)) param.name <- make.names.jm(param) else
```

```
    param.name <- deparse(substitute(param))
#  Next we break down the analysis into 2 cases.
case <- NA
if (class(bugs.out) == "bugs" && is.character(param)) case <- 1
if (is.na(bugs.out) && (is.matrix(param) | is.list(param)))
    case <- 2
if (is.na(case)) stop("\n",
    "The call to plot.chains did not specify the param and/or bugs.out\n",
    "argument correctly.  Either bugs.out should be NA and param should be\n",
    "a matrix of chains for a parameter, or bugs.out should specify the output\n",
    "of a call to the bugs function in the R2WinBUGS package and param should\n",
    "be the name of a parameter that was saved in this output.  The parameter can\n",
    "be specified by either its WinBUGS name or the R-legal name that is created\n",
    "by the extract.chains function (it uses the make.names.jm function to\n",
    "creat R-legal names."  ) #end stop
#  The next if takes care of the case where param is a list that was created by using extract.chains to extract
#  the chains of samples for one parameter.  In this case, the output of extract.chains is a list of length 1.
#  For this case, the next if converts param to a matrix.
if (case == 2 && is.list(param) && length(param) == 1 && is.matrix(param[[1]]))
    param <- param[[1]]
#  The next if gives an error message when the preceding if does not apply.
if (case == 2 && is.list(param) && (length(param) != 1 || !is.matrix(param[[1]])))
    stop("\n",
        "If bugs.out is NA, then param must be either a matrix of chains, or\n",
        "a list whose only component is a matrix of chains.  Check that param\n",
        "has been specified appropriately.\n"
        ) #end stop

if (case == 1) {  #begin case where bugs.out is a bugs output file

    list.chains <- extract.chains(bugs.out, improve.names = TRUE)

    if (!(param.name %in% names(list.chains))) stop(paste(
        "The specified parameter name is not among the parameter names in\n",
        "the bugs output.  The requested parameter is named, ", param,
        " and\nthe parameters in the bugs output are named:\n     ",
        paste(names(list.chains), collapse = ", "), "\n", sep = "") #end paste
        ) #end stop

    param  <- make.names.jm(param)
    y.vals <- list.chains[[param]]

    } #end if (case == 1) The case where bugs.out is a bugs output file

#  In case 2, param is a matrix of chains.
if (case == 2) y.vals <- param

#  Stop procedure if xlim.f is out of range.
if (!any(is.na(xlim.f)))
    if ((xlim.f[1] < 1) | (xlim.f[2] > length(y.vals[,1]))) stop(paste(
        "xlim.f = c(", xlim.f[1], ", ", xlim.f[2], ")\n",
        "Either the lower bound of xlim.f is less than 1 or the upper\n",
        "bound of xlim.f exceeds the number of iterations in the chains.\n",
        "Check that the bounds in xlim.f make sense.\n"
```

```
          , sep = "") #end paste
       ) #end stop

# Next define xx
if (is.matrix(y.vals)) {
    if (any(is.na(xlim.f))) {
       xx <- 1:length(y.vals[,1])
       xlim.f <- range(xx)
       } else { #end if (any(is.na(xlim.f)))
       xx <- xlim.f[1]:xlim.f[2]
       y.vals <- y.vals[xx, ]
       } #end else
    n.chains <- ncol(y.vals)
  } else { #end if (is.matrix(y.vals)
    if (any(is.na(xlim.f))) {
       xx <- 1:length(y.vals)
       xlim.f <- range(xx)
       } else { #end if (any(is.na(xlim.f)))
       xx <- xlim.f[1]:xlim.f[2]
       y.vals <- y.vals[xx]
       } #end else
    n.chains <- 1
    } #end of else for if (is.matrix(y.vals)

# x.dummy & y.dummy are just dummy variables that establish the lower and
# upper limits of the plot.  See the following plot command.
    x.dummy <- xlim.f
    y.dummy <- range(y.vals)
# If legend = TRUE, then the upper limit of y.dummy is extended slightly upwards.
    y.dummy[2] <- y.dummy[2] + .025*(y.dummy[2] - y.dummy[1])
# The y-limits are set to y.dummy only if they are not specified in the function call.
if (any(is.na(ylim.f))) ylim.f <- y.dummy

plot(x.dummy, y.dummy, type = "n", xlab = "", ylab = "", ylim = ylim.f,
    bty = "l", ...)
# line.col designates different colors for different chains.  See the for loop below.
# If there are more than 8 chains, the remaining chains are all plotted in black.
line.col <- c(
    "red","blue","green", "black", "azure3", "deeppink",
    "cyan2", "darkmagenta")
if (length(line.col) < ncol(y.vals))
    line.col <- c(line.col,
                  rep("black", ncol(y.vals) - length(line.col)))

if (is.matrix(y.vals)) {
    if (legend) {
       aa <- (xlim.f[2] - xlim.f[1])/n.chains
       L.ini <- xlim.f[1] + aa/4
       for (cc in 1:(n.chains - 1)) L.ini <- c(L.ini, L.ini[cc] + aa)
       inc.1 <- aa/6
       inc.2 <- (aa/6) + (aa/20)

       for (cc in 1:n.chains) {
          lines(c(L.ini[cc], L.ini[cc] + inc.1), c(ylim.f[2], ylim.f[2]),
```

```
            lwd = 2, col = line.col[cc])
          text(L.ini[cc] + inc.2, ylim.f[2], paste("Chain", cc), adj= 0)
          } #end for (cc in 1:n.chains)

       } #end if (legend)

    for (kk in 1:n.chains) {
       lines(xx, y.vals[ , kk], col = line.col[kk])
       } #end for (kk in 1:ncol(y.vals))
    } else { #end if (is.matrix(y.vals))
       lines(xx, y.vals, col = line.col[1])
    } #end of else for if (is.matrix(y.vals))

if (add.labels) {
    mtext("Iteration", side = 1, cex = cex.lab, line = 2.5)
    mtext(paste("Sample of", param.name),
       side = 2, cex = cex.lab, line = 2.5)
    mtext(paste(n.chains, "Chains of Samples for", param.name),
       side = 3, cex = cex.lab, line = 1)
    } #end if (add.labels)

    } #end def of plot.chains function
```

25. Code for `plot.param` function

```
plot.param <- function(param.post, plot.dist = TRUE,
    method = c("density", "logspline")[1],
    output = c(stats, density)[1],
    stats.which = c(mean,median,mode,conf), conf.pct = 95,
    show.conf = FALSE, show.stats = TRUE, digits.f = 3, lwd.f = 3,
    cex.stats = 1.15,
    xlab.f = paste("Samples of", deparse(substitute(param.post))),
    ylab.f = "Probability Density", xlim.f = NA, stats.ht = .90 ) {
#Subp.ini  ----hidden code ---- end hidden-----------

#  Replace conf with bnd.low & bnd.hi
if (conf %in% stats.which) {
    stats.which[match(conf, stats.which)] <- bnd.low
    stats.which <- c(stats.which, bnd.hi)
    } #end if (conf %in% stats.which)

if (method == "density") {

#  Compute statistics for of the density function
    xx <- density(param.post)$x
    yy <- density(param.post)$y
    mode.dist <- xx[yy == max(yy)]

    tm.stats <- c(mean(param.post), median(param.post), mode.dist,
                  quantile(param.post, prob = (1 - conf.pct/100)/2),
                  quantile(param.post, prob = 1 - (1 - conf.pct/100)/2) )

    if (length(digits.f) == 1) dig.nu <- rep(digits.f, length(tm.stats)) else
       dig.nu <- c(digits.f, digits.f[length(digits.f)])
```

```
    res.stats <- NULL
    for (i in 1:length(tm.stats))
       res.stats <- c(res.stats, round(tm.stats[i], dig.nu[i]))
    names(res.stats) <- c(mean,median,mode,bnd.low,bnd.hi)

    } #end if (method == "density")

if (method == "logspline") {

    tmr <- require(polspline, quietly = TRUE)
    if (!tmr) stop("\n",
       "This function requires the package polspline which is not available\n",
       "on this computer system.  If this is your personal computer, run the ",
       "function\n",
       "\n",
       "install.packages(\"polspline\")\n",
       "\n",
       "to install polspline on this computer.  If this is a network computer,\n",
       "ask the system administrator to install polspline on this network.\n"
       ) #end stop

    fit.post <- logspline(param.post)
    xx <- seq(from = min(param.post), to = max(param.post), length = 1000)
    yy <- dlogspline(q = xx, fit.post)
    mode.dist <- xx[yy == max(yy)]

    tm.stats <- c(mean(param.post), median(param.post), mode.dist,
                  quantile(param.post, prob = (1 - conf.pct/100)/2),
                  quantile(param.post, prob = 1 - (1 - conf.pct/100)/2) )

    if (length(digits.f) == 1) dig.nu <- rep(digits.f, length(tm.stats)) else
       dig.nu <- c(digits.f, digits.f[length(digits.f)])

    res.stats <- NULL
    for (i in 1:length(tm.stats))
       res.stats <- c(res.stats, round(tm.stats[i], dig.nu[i]))
    names(res.stats) <- c(mean,median,mode,bnd.low,bnd.hi)

    } #end if (method == "logspline")

if (plot.dist) {
       if (any(is.na(xlim.f))) xlim.f <- c(min(xx), max(xx))

       plot(xx, yy, xlab="", ylab="", main ="", type = "n",
          xlim=xlim.f, ylim = c(0, max(yy)*1.15), bty = "l")

       if (show.conf) {
          int <- xx >= res.stats[bnd.low] & xx <= res.stats[bnd.hi]
          xx.ci <- xx[int]
          yy.ci <- yy[int]
          polygon(area.under(x = xx.ci, y = yy.ci), col = yellow)
          } #end if (show.conf)
```

```
        lines(xx, yy, lwd = lwd.f)
        abline(h=0, lty=2, lwd = .5)

        mtext(xlab.f, side = 1, cex = 1.75, line = 3)
        mtext(ylab.f, side = 2, cex = 1.5, line = 2.5)

        if (show.stats) {
            stats.out <- res.stats[stats.which]
            stats.names <- names(stats.out)

            text.out <- NULL
            for (i in 1:length(stats.out)) {
                text.out <- c(text.out,
                    paste(stats.names[i], " = ", stats.out[i], "\n", sep=""))
                } #end for (i in 1:length(stats.out))
            if (res.stats[mode] > xlim.f[1] + (xlim.f[2] - xlim.f[1])/2)
                text(xlim.f[1] + .03*(xlim.f[2] - xlim.f[1]), stats.ht*max(yy),
                    paste(text.out, collapse = ""), adj = 0, cex = cex.stats) else
                text(res.stats[mode] + .33*(xlim.f[2] - res.stats[mode]),
                    stats.ht*max(yy), paste(text.out, collapse = ""), adj = 0,cex =
cex.stats)

            } #end if (show.stats)

        } #end if (plot.dist)

if (all(output == stats)) out <- res.stats[stats.which][stats.which]
if (all(output == density)) out <- cbind(x = xx, y = yy)
if (stats %in% output & density %in% output)
    out <- list(stats = res.stats[stats.which][stats.which],
        density = cbind(x = xx, y = yy))

return(out)

    } #end def of plot.param function
```

26. Code for `res.param` TOC

27. Code for `show.bugs` TOC

```
show.bugs <- function(bugs.out) {
    param.names <- dimnames(bugs.out$sims.array)[[3]]
    info <- c(n.iterations = bugs.out$n.iter,
              n.burnin = bugs.out$n.burnin,
              n.chains = bugs.out$n.chains)

    out <- list(
              parameters = param.names,
              about.samples = info
          ) #end list
    return(out)
    } #end def of show.bugs function
```

****