R Package rube (Really Useful WinBUGS (or JAGS) Enhancer)

Version 0.3-10, November 29, 2017

Author: Howard J. Seltman

1 Introduction

These notes are on the world wide web at:
http://www.stat.cmu.edu/~hseltman/rube/rubeManual.pdf.

The R package, `rube` is a **Really Useful WinBUGS (or JAGS) Enhancer**. It makes working with WinBUGS much easier. It is (currently) built on top of the R2WinBUGS package. It works with either the WinBUGS or the JAGS MCMC engines. The latter requires the R2jags package. R2WinBUGS or R2jags takes a good first step to allow you to

- Get your data and parameter initializations ready in R.
- Run your model in WinBUGS (or JAGS) without touching WinBUGS (or JAGS). (The WinBUGSdoodle editor is really just a toy!)
- Examine your results in R.

The `rube` package is a wrapper for R2WinBUGS or R2jags, but also *much* more. If you use it, you don't need to learn the details of R2WinBUGS or R2jags.

Although `rube` contains many features and options, most users will want to start with use of the substitution, `LC()` and `FOR()` pseudo-functions for better model code maintenance, the use of `rube`() to check and run analyses (while hiding bugs() and WinBUGS or JAGS() and jags), and the use of `p3()` and `compare()` to examine MCMC results.

2 Philosophy

WinBUGS is a powerful and flexible program for performing Bayesian analyses. Unfortunately it is **not** user-friendly in a number of ways. While the R package `R2WinBUGS` simplifies getting data and initial parameter values into and out of WinBUGS, that only slightly reduces the pain, effort, and frustration of working with WinBUGS, especially when setting up your own model.

JAGS is another (perhaps better) Bayesian MCMC engine that has major overlap in model syntax with WinBUGS. As opposed to WinBUGS, it is cross-platform and work on Macs and Linux.

The `rube` package aims to greatly reduce the pain, effort, and frustration in several ways. It is based on these ideas:

1. Analysts make mistakes writing model description code and want specific information on where they went wrong (WinBUGS doesn't even point to the code line with the error, let alone explain the error in most situations).

2. Analysts make mistakes in setting up data and initializations that match the model code, and want specific information on where they went wrong. WinBUGS often simply gives a useless "TRAP" error if bad values are used. What users need is a summary of the relationships between the model distributions, the data, and the parameter initializations as an aid to finding the problems.

3. Real world analyses often involve working with different sets of covariates, including interactions, and WinBUGS and JAGS only offers two poor choices: maintaining multiple model files or using a matrix form for input with no meaningful variable names. We need a simple, flexible, familiar method for specifying covariates without modifying model code files.

4. Analysts often want to consider different approaches for small portions of their model code, but it is best to maintain only a single model file with "conditional coding" which is not supported by WinBUGS or JAGS.

5. Analysts choose default values for hyperparameters, but often perform sensitivity testing at some point. It would be nice to keep defaults in the model file, ignore them mostly, but easily change them without altering the model file.

6. Efficient and transparent methods are needed for automated evaluation of models on repeated simulated datasets, as well as for sensitivity testing.

7. Initialization functions should have access to the data and run time options so that they can flexibly set their values.

8. If you forgot to save parameters that are just functions of other parameters it should be easy to add them, so that they are included in the plots and summaries.

9. WinBUGS or JAGS result objects should be more self-documenting so that there is never any question about the conditions under which results were generated.

10. The R2WinBUGS or JAGS overview plot of results is not that useful for complex models. A generic plotting function to interactively examine traces, ACFs, and distribution shapes is the best first step to examining an MCMC result.

11. The ability to plot priors and posteriors on the same plot is highly desirable.

12. Functions are needed to help explore choice of prior distributions.

13. When considering multiple related models the specific data values, initializations, and parameter.to.save may vary. It would be nice if superfluous values would be automatically culled to simplify coding and avoid WinBUGS or JAGS errors (especially losing a correct analysis because you specified monitoring a non-existent node). Similarly, it would be nice to simply specify "all" as the parameters.to.save.

The `rube` package solves all these problems and more. `Rube` **is designed to reduce your cognitive overload relating to technical details so that you can focus on the bigger issues of Bayesian data analysis.**

3 One-time Setup

The `rube` package has so-far only been tested on a Windows operating system (including CMU Virtual Andrew and Parallel on Macintosh), with both WinBUGS (because WinBUGS only runs on Windows) and JAGS. I expect it to work with JAGS on a Macintosh.

If WinBUGS is your MCMC engine choice, first **install WinBUGS** from http://www.mrc-bsu.cam.ac.uk/bugs/. Click on "WinBUGS 1.4.3" under "WinBUGS 1.4.3" to get to the download page. Note that Windows Vista and Windows 7 do not allow writing to the "Program Files" folder, but WinBUGS tries to do this. Therefore on these operating systems, you should install WinBUGS into a folder such as "c:\\WinBUGS14". If you do this you will need to explicitly tell `rube`() and `getBugsExample()` where to find WinBUGS as explained below. (Alternatively you can let WinBUGS be installed in "Program Files", but start R using right-click/"Run as administrator".)

As of November 2012, it is also necessary to follow the instructions on that page to also install the latest WinBUGS patch as well as the "immortal key". Verify that you can run WinBUGS directly, and note the location of the WinBUGS program file. (If WinBUGS did not install its start icon on your desktop, first look in "All Programs" on the Windows Start menu. If you can't find it there, look for the WinBUGS14.exe icon in the WinBUGS14 folder at the install location.) If the program file "WinBUGS14.exe" is not in "c:/Program Files/WinBUGS14/" then you will need to need to set the environmental variable "BUGSDIR" (see below) or specify the location of the program file using the `rube` argument 'bugs.directory' each time you run `rube`().

If JAGS is your MCMC engine choice, install JAGS from JAGS.

You should **install R** (or upgrade to at least version 3.4.1) before installing `rube`. To install R go to http://r-project.org, and then click on "CRAN" under "Download", then choose a mirror, then click "Download R for Windows", then click "base", and finally click "Download R #.##.# for Windows". Save the file to any location, then run the file to install R. I recommend a custom installation so that you can change the default "mdi" to "sdi".

Next, prepare to install `rube`() by **installing the R packages** `R2WinBUGS` and `stringr`. To do this start R with its icon, then go to the "Packages" menu and choose "Install packages...". Next choose a mirror and select the two packages to be installed, using control-click to add the second package. (See below for special procedures needed when you do not have sufficient privileges to install packages.) For JAGS use, also install package R2jags.

As of November 2012, `rube` is not yet available from public sources, so you need to follow this special procedure to **install** `rube`. Put rube_0.2-17.zip anywhere on your computer, then use the R menu option `Packages => Install package(s) from local zip files`.

Note: If you are upgrading `rube` from an earlier version, **and** you also use a .First() function to load `rube` automatically when R starts, you will need to unload the old version before installing the new version using the R commands

```
detach("package:rube")
unloadNamespace("rube")}.
```

Like any package, you install `rube` once, and then you need to use `library(rube)` each time you start R (or put the "library(rube)" command into your .First function).

To **test** `rube` start with these commands:

```
library(rube)
m = "model {
       for (i in 1:N) {
         x[i] ~ dnorm(mu, 1)
       }
       mu ~ dnorm(0, PREC.MU=0.01)
    }"
d = list(N=20, x=rnorm(20, 5, 1))
i = function() list(mu=rnorm(1,0,5))
r = rube(m, d, i)
summary(r)
```

This should show a "Rube Summary" with sections called "For variables", "No assignments", "Constants", "Data", and "Stochastics". The final line should be "No problems detected!". If anything different appears, you have not loaded `rube` correctly or have entered the test code incorrectly.

Next run

```
r = rube(m, d, i, "*", n.burn=0, n.thin=1, n.iter=100)
summary(r)
```

What should happen is that JAGS or WinBUGS will run. WinBUGS opens it's own window, runs for a while, quickly draw some nice colored graphs, and then automatically close, returning you to R. JAGS shows a gui progress bar, then returns you to R. The result of the `summary(r)` command is a "Rube summary" with information about the MCMC run, with no errors or warnings and ending with the "DIC" value for the model. If you see this, WinBUGS and `rube` are correctly installed, and you may procedure to your own data analysis.

If WinBUGS runs, produces graphs, but stops with a "BlackBox" window with a "trap #60" message, then close the trap box, and notice the R warnings about "file.create" being unable to create the file "Registry.odc". In this case your problem is with insufficient Windows file permissions. The solution is to either

1. Re-install WinBUGS outside of "C:/Program Files"

2. Run R each time you use `rube` by right-clicking the R icon and then selecting "Run as administrator" and then answering "yes" to "Do you want to allow the following program from an unknown publisher to make changes to this computer?" (recommended solution)

3. Ignore these particular "trap" and "warning" messages each time you run `rube`.

If WinBUGS does not open and you get an error with the warning message "WinBUGS failed: WinBUGS executable does not exist in ..." then the problem is that `rube` cannot find WinBUGS. The possible solutions are:

1. specify the folder that contains "WinBUGS14.exe" as the 'bugs.directory" argument of `rube`.

2. Specify the folder that contains "WinBUGS14.exe" as the Windows environmental variable "BUGSDIR" (see below under "Environmental Variables").

At this point you should try method 1 first. If WinBUGS now runs, but hangs, continue below. If it runs correctly, you should try method 2 as a nicer alternative to specifying the folder every time you run `rube`. Then you will be ready for your own data analysis.

If WinBUGS opens, but does not run the model and just hangs, then the problem is the specification of the "working directory". Close WinBUGS, and fix the problem as specified here. (You may want to run `getwd()` in R; if the result starts with "////" (or with backslashes) then the problem is definitely that WinBUGS does not know how to work with that kind of folder name.) The possible solutions are:

1. Specify a different working directory using the 'wd' argument of `rube`.

2. Use the R menu item "File / Change dir..." to change your R working directory to something that starts with "C:" (or any other single alphabetic character that is a disk drive name followed by the colon).

3. Specify the folder that will serve as your working directory as the Windows environmental variable "BUGSWD" (see below under "Environmental Variables").

3.1 Using RUBE when you don't have administrative sufficient administrative privileges to install new packages (courtesy of Fabrizio Lecci)

This problem arises when using "public" machines (including Virtual Andrew at CMU) rather than your own computer. From the WinBUGS website download the "zipped version of the whole file structure" to any location on your computer. Unzip it to a folder that has a simple single drive letter as the start of its location. (Usually this can be done by right-clicking on the zip file, and then choosing "unzip".) Remember to separately install the patch and the immortal key.

To perform the one-time install of the `rube` package, download the `rube` zip file to any location on your computer, and the run the R command `.libPaths(‘‘xxx’’)` where `xxx` is a folder location that has a simple name and which you have write access to (e.g., W:/Desktop/RLibs on CMU Virtual Andrew). Then from the R menu choose "Packages / Install package(s) from local zip files", and navigate to the zip file that you saved.

Each time you use `rube` in the future you will need to run the following the R commands once per R session:

```
.libPaths("xxx") # xxx is the same location used when installing rube
library(rube)
Sys.setenv("BUGSDIR", "yyy") # yyy is the folder holding WinBUGS14.exe
```

After first installation, and these three setup lines, perform the above check of `rube` using the simple model, data, and initialization given there.

If the test shows the WinBUGS hangs, find a good alternate working directory location and then for each R session, set that as the `rube` 'wd' argument by using:

```
Sys.setenv("BUGSWD", "zzz") # zzz is the folder used as the working directory
```

3.2 Using RUBE on Macs with WinBUGS (courtesy of Liz Lorenzi)

Three options for running RUBE and WinBUGS on Macs are

1. to run Window remotely (e.g., Virtual Andrew at CMU, see above)
2. to use dual boot with a Windows partition
3. to use Parallels to run Windows 7. This option allows for simultaneous use of Windows OS and Mac OS and the installation and use of Windows programs.

As of September 2013, running JAGS and rube directly on the Mac is preferred.

To use the Parallels option, buy the Parallels USB drive (possibly online at http://www.parallels.com/products/desktop/). You will also need to have a version of Windows 7 to install. Follow the directions prompted by Parallels and Windows 7 to install the Parallels desktop and Windows platform for your Mac.

Once Windows is installed, you will need to install R to run on the Windows operating system. Do this by opening a browser window through the Windows start menu and go to http://www.r-project.org to find the latest version of R for Windows. Then follow the instructions outlined above to install WinBUGS and `rube`.

Next, select the directory where you intend to save and use your files, i.e., the "working directory". Change to this directory by going to File ⇒ Change Dir. This step is especially

important for running `rube` using Parallels. R will automatically select a directory that is unreadable by WinBUGS. Parallels causes the default directory to be a mirror version of your Mac documents, but in a format that cannot be read by WinBUGS.

3.3 Environmental Variables and .First

When `rube` runs, if no value is specified for the 'bugs.directory' argument, the Windows environmental variable "BUGSDIR" is checked first and this is used in place of the default value if it exists. Similarly then "BUGSWD" environmental variable sets the 'wd' (working.directory) argument.

There are two ways to set the environmental variables. If you have administrative privileges and want to set one or both (typically just "BUGSDIR") permanently then follow this procedure:

1. Open the Windows Control Panel
2. Open the System control panel icon (perhaps found under Performance and Maintenance)
3. On the Advanced Tab click "Environmental Variables"
4. In the "User variables" panel, click "New", then enter "BUGSDIR" as the "Variable name" and the location where you installed WinBUGS as the "Variable value". Click "OK".
5. Click "OK" back in the "Environmental Variables" dialog box, and then "OK" in the System Properties dialog box to complete the process. The next time you start R, `rube`() and `getBugsExample()` will automatically find WinBUGS. If you have problems, try the R command `Sys.getenv("BUGSDIR")` to verify that you have correctly set the environmental variable.

Or, to set an environmental variable for one R session, use

```
Sys.setenv(BUGSDIR="c:\\WINBUGS14")
```

each time you run R. (Of course, you must enter the location of WinBUGS on your own computer inside the quotes.)

One useful R feature is the `.First()` function. If you define this function and "save workspace image" when you quit R, then this function will run automatically each time you start R with the same working directory. Typically `library(rube)` and one or two `Sys.setenv()` commands are used as the body of the .First function.

Special note on upgrading `rube`: If you put `library(rube)` in your `.First()` function, this will prevent you from loading a new version of `rube`. The solution is to enter these two commands to remove the old version before loading the new one:

```
detach("package:rube")
unloadNamespace("rube")
```

4 Features

Here are the main features of `rube`:

1. The `rube` package will perform syntax checking of your model description "code" with explicit error messages. This finds most syntax errors and several other types of errors.

2. When you supply data and initializations along with your model code, `rube` can also help you find many additional problems related to the relationships between the model code, the data, and the initializations because it summarizes these relationships. This often gives ideas about how to fix WinBUGS "Trap" errors, because these are commonly due to inappropriate initializations. JAGS tends to give clearer error messages, but the `rube` summary is still useful to help you be sure your model is what you intended it to be.

3. Easy to use "pseudo-functions" make your code much more flexible and easier to maintain.

   (a) Anywhere in your code, you can use the "substitution" syntax `VAR=value` to set default values. This feature is most useful for setting hyperparameters. The two advantages are that it makes the code easier to read (it becomes more obvious where you chose arbitrary hyperparameter values), and it makes hyperparameter sensitivity testing easy because you can add simple run-time options to override any or all defaults without modifying your code. In addition, the utility function `showDefaults()` can be used to list all of the defaults in a model.

   The use of an expression for a Normal precision, as shown below, allows the more familiar standard deviation to be visible to those reading the code.

   As an example,

   ```
   Y[i] ~ dnorm(0, Y.PREC=1/100^2))
   ```

   will be interpreted as

   ```
   Y[i] ~ dnorm(0, 0.0001)
   ```

   if nothing special is done, but as

   ```
   Y[i] ~ dnorm(0, 0.0025)
   ```

   if `subs=list(Y.PREC=1/20^2)` is used as a run-time function option.

   (Note that "^" for powers is not available in WinBUGS or JAGS. The "subs" argument to `rube` shown above uses R to convert 1/20^2 to 0.0025 before WinBUGS or JAGS see it. As shown above, the use of "^" in model code is allowed by `rube` as a special exception in arguments to stochastic functions, and

the corresponding value will appear in the model code file seen by WinBUGS or JAGS.)

(b) Coding of covariates is made simpler, easier to read, and much more flexible (while avoiding the maintenance problems of multiple model code files). This is accomplished through use of the linear combination pseudo-function which takes the form `LC(prefix, FORMULA, suffix, index)` anywhere in your code. Here prefix, suffix, and index are "hard-coded" in your model, but FORMULA is specified at run-time (with an intercept only as the default), and FORMULA is just like the right-hand-side of an R model formula (with some minor restrictions). The formula will include the covariates of interest for a particular WinBUGS or JAGS model run, and the prefix and suffix are used to automatically create corresponding parameter names. An intercept is created unless -1 is included in the formula. The index is automatically appended to the covariates values as shown below. Here are a couple of examples:

```
mu[j] ~ dnorm(LC(b,DEMOGS,,j), Y.PREC=0.001)
```

will be interpreted as

```
mu[j] ~ dnorm(beta0 + bOld*Old[j] +
         bMale*Male[j] + bOldMale*Old[j]*Male[j], 0.001)
```

in various `rube` functions when you define `varList=list(DEMOGS="Old*Male")`. Then when you re-run your analysis with, say, `varList=list(DEMOGS="Old*Male+White")` the appropriate covariate model is automatically used. Your model can have any number of formulas, and the same name can be used in multiple places, or any number of different names can be used, depending on your model complexity. You can create parameter names corresponding to each covariate using the prefix, or both the prefix and suffix.

An example of a more complex covariate formula is `A+B+C+B:C+D*E*F+(G+H+I)^2` which is shorthand for the model formula

```
A + B + C + B:C + D + E + F + D:E + D:F + E:F + G + H +
    I + G:H + G:I + H:I
```

Currently the model formulas do not allow "-" (except -1 is allowed) or "/", and inside the `(...)^n` syntax "A:B" and "A*B" are treated as "A+B". **Warning:** `rube` does not *recode* your variables, so the formula system will be inappropriate for k-level categorical variables that have not been manually recoded into k-1 indicator variables (or other suitable coding). But the system does work for 0/1 or 1/-1 indicator variables and quantitative covariates.

(c) Parallel with LC() is FOR() which generates the corresponding prior distributions for the covariate parameters. The syntax is `FOR(prefix, FORMULA,`

`suffix, codeText)`. This syntax generates one model code line for each term/parameter in the FORMULA by replacing a question mark (?) in the "codeText" with the parameters one by one. Continuing with the above example, we might have a model code line

```
FOR(b,DEMOGS,, ? ~ dnorm(0, COV.PREC=0.001))
```

to automatically generate

```
b0 ~ dnorm(0, 0.001)
bOld ~ dnorm(0, 0.001)
bMale ~ dnorm(0, 0.001)
bOldMale ~ dnorm(0, 0.001)
bWhite ~ dnorm(0, 0.001)
```

when `varList=list(DEMOGS="Old*Male+White")` is defined. If intercept `b0` has a different prior, code it separately and use `varList=list(DEMOGS="Old*Male+White-1")`.

(d) A less used pseudo-function is `BASE(FORMULA,index)` which can be useful for encoding a baseline covariate status. E.g.

```
isBase[i] <- BASE(DEMOGS,i)
```

will be coded as

```
isBase[i] <- (1-Old[i])*(1-Male[i])*(1-White[i])
```

when `varList=list(DEMOGS="Old*Male+White")` is defined.

(e) The final improvement in the form of pseudo-functions is conditional coding. This allows two (or more) options to be in the code with the final choice made at run-time. One example of where this is useful is for a choice of distributions for a parameter. There are two forms of conditional coding: in-line and multi-line.

In-line conditional coding takes the form
`code1 IFCASE(caseFormula) code2 ELSECASE code3 ENDCASE code4`
which results in
`code1 code2 code4`
if the caseFormula evaluates to TRUE and
`code1 code3 code4`
if the caseFormula evaluates to FALSE. The `code1`, `ELSECASE code3`, and `code4` elements are all optional. The "caseFormula" can be simple such as `GAMMA` or complex such as `GAMMA || (ALPHA && !INTERCEPT)` which will, e.g., evaluate to FALSE and TRUE respectively when the argument `cases=c("ALPHA","AGE2")` is included in the various `rube` functions. This uses standard R evaluation including OR (`||`), AND (&&), NOT (!) and parentheses. Each element in

the "subs" vector is defined TRUE and all missing elements are defined to be FALSE. The caseFormula ***is*** case sensitive.

An example of the multi-line form is:

```
IFCASE(GAMMA || (ALPHA && !INTERCEPT))
a <- b + c
d <- e + f
ELSEIFCASE(ALPHA)
a <- b + c
d <- e - f
ELSECASE
a <- b
d <- f
ENDCASE
```

which evaluates to

```
a <- b + c
d <- e + f
```

when the argument `cases=c("ALPHA","AGE2")` is used, and

```
a <- b + c
d <- e - f
```

when the argument `cases=c("ALPHA","INTERCEPT")` is used.

4. The heart of the `Rube` package is the `rube`() function which is a wrapper for the `bugs()` function of R2WinBUGS or the `jags() and jags.parallel()` functions of R2jags. The functionality of `rube`() includes the ability to generate data on the fly and to more flexibly generate parameter starting values on the fly (see below), as well as extensive abilities to modify WinBUGS or JAGS code on the fly (see above). I recommend that you always use `rube` to run WinBUGS or JAGS to perform MCMC from on your data using your model. Here are two examples (both at http://www.stat.cmu.edu/∼hseltman/rube/#examples).

   - basicExample.R.
   - Rasch.R.

5. `Rube` is very flexible in terms of the form in which the WinBUGS or JAGS model may be specified. It allows either a file with the model specification or a string vector or a single vector separated by newline (\n) characters. The latter is convenient for keeping your model definition in the same file as your code, where you just define your model in this form:

```
myModel = "model {
  for (i in 1:N) {
  prec.y[i] <- pow(sig.y[i], -2)
    Y[i] ~ dnorm(x[i], prec[i].y)
  }
}"
```

(But with this string variable form of the model, you must remember to reload the text defining the string whenever you change the text.)

To examine this very long string in R, use `model(myModel)`. To see a particular version of the model based on evaluating any conditional code and/or pseudo-functions, use a form like

```
model(myModel, cases=c("singleX","gamPrior"),
      varList=list(RHS1="A+B", RHS0="A+B-1"))
```

Also note that `model()` may be run on a `rube`() result (whether successful or not) to see the model code used for that `rube`() run.

6. `Rube` allows data to be provided as a data.frame. In this case, an "N" variable is automatically created which equals the number of rows in the data.frame. A special feature for data in the form of data.frames is provided for the case where a small number of additional data value must be provided in addition to the data.frame columns and "N": any numeric attributes of the data.frame are added to the data list automatically. Here is an example:

```
m = "model {
       known.prec <- pow(known.sd, -2)
       for (i in 1:N) {
         x[i] ~ dnorm(mu, known.prec)
       }
       mu ~ dnorm(0, MU.PREC=1/10^2)
     }"
rube(m)
d = data.frame(x = rnorm(30, 5, 1))  # or some read.table()
rube(m, data=list(d, known.sd=1.1))
# alternate approach
attr(d, "known.sd") = 1.1
rube(m, data=d)
i = function() list(mu=rnorm(1,0,5))
```

```
rube(m, d, i)
# recommended initial run conditions to investigate burnin:
r = rube(m, d, i, "mu", n.burn=0, n.thin=1, n.iter=1000)
p3(r)
```

7. Particularly for simulation studies, the ability to provide data as a data-generating function with optional arguments is a useful alternative to the usual method of providing data as a list. `Rube` allows its data argument to be a function that returns a list. Optional arguments to the function can be provided using the "dataParams" argument to `rube`(). Here is a trivial example:

```
m = "model {
       for (i in 1:N) {
         x[i] ~ dnorm(mu, 1)
       }
       mu ~ dnorm(0, MU.PREC=1/10^2)
     }"
rube(m)
d = function(ns, truemu) {
  dtf = data.frame(x = rnorm(ns, truemu, 1))
  return(dtf)
}
rube(m, d, dataParams=list(ns=20, truemu=6))
i = function() list(mu=rnorm(1, 0, 5))
rube(m, d, i, dataParams=list(ns=20, truemu=6))
rube(m, d, i, "*", dataParams=list(ns=20, truemu=6))
```

8. As in R2WinBUGS or R2jags, parameter starting values can be set with either a data list or a function, but `rube` allows for passing arguments to this function. This allows alternate methods of starting value generation in different situations, possibly fully or partially data driven. In complex Bayesian problems, this can be extremely helpful.

9. WinBUGS and JAGS fail (sometimes after the MCMC is complete) when extra information is supplied in various forms. `Rube`() automatically culls extra data, parameters, and "parameter.to.save" elements before passing them to WinBUGS or JAGS to avoid this problem. (For JAGS error are also prevented by removing non-stochastics from the initializations.) This means, e.g., that you can construct your (complete) data once, and use that same object even if you drop some covariates from your model. Similarly, you can have a single parameter initialization function, and when you drop parameters from the model, `rube`() will adjust and prevent

the WinBUGS or JAGS error. (`Rube`() does give a warning message when it drops elements to avoid any confusion about what information gets through to WinBUGS or JAGS.)

10. The object returned from the `rube`() function is self-documenting. It contains components called model, startTime, runTime, and bugs.seed that can be examined. It also contains the full analysis of the relationship between the model code, the data, and the initialization, which can be viewed using the generic `check()` function on the `rube` object. The final model code can be seen using `model()` function on the `rube` object. In addition the `rube` object has the LC()/FOR() model formulas in its "varList" component.

11. The `rube` package tends to preserve the `bugs()` defaults, but it does reverse the default for a random number seed. By default `rube`() generates a random number seed, supplies it to WinBUGS or JAGS, and stores it in the `rube`() result as the bugs.seed component. If you want the `bugs()` default, which gives exactly the same MCMC results for each run, set `bugs.seed=NULL` in the `rube`() command line. You can also use `bugs.seed=foo$bugs.seed` to get identical MCMC results as for the prior run whose `rube`() results are stored in "foo".

    Another difference is that the 'parameters.to.save' argument can be set to `"*"` to save all pararmeters, or, e.g., `c("*", "bage", "RIs")` to save all but the "bage" parameter and the "RIs" parameter vector.

12. The `rube` package provides a function called `p3()` which plots the trace, AR plot, and histogram for each monitored parameter. For multidimensional parameters, initially a random subset of the parameters is shown in a panel display, with new random subsets generated by a single click and with a detailed view of individual parameters also available. The related set of parameters generated by each LC() command (excluding the intercept) are also displayed together in a panel. In addition, the viewGroups= argument of `p3()` can be used to group other parameters. `p3()` allows you to add new and view parameters that are compuatble from old ones (see below).

13. The `rube` package provides a function called `priPost()` which is a prior/posterior density plotter to compare the posterior density of various distributions to their prior densities. This includes some "compound" prior distributions, e.g., a beta or gamma that has two gammas as its parameters.

14. The `compare()` function takes a list of `rube` objects as its input and graphically compares results across objects on a parameter-by-parameter basis.

15. As an aid to learning WinBUGS, `rube` has a function called `getBugsExample()` which reads the model(s), data, and initializations of the standard WinBUGS examples into R. (This cannot be done with simple copy-and-paste from WinBUGS because any data in matrices or arrays are improperly imported into R.)

16. Functions for **exploration of reasonable prior distributions** are in hyperExplore.R (but will be added to `rube` in the near future).

    - betaHyperExplore() allow interactive setting of the four hyperparameters of a beta distribution parameterized by two gamma distributions. A histogram of the prior density and an image plot of the means and standard deviations of the betas that are being mixed to produce the final composite beta are shown. The final explored parameters are returned.
    - gammaHyperExplore() allow interactive setting of the four hyperparameters of a gamma distribution parameterized by two gamma distributions. A histogram of the prior density and an image plot of the means and standard deviations of the betas that are being mixed to produce the final composite gamma are shown. The final explored parameters are returned.
    - betaHyperPlot() shows the histogram of a composite beta prior density from four gamma hyperparameters or the posterior histogram from the "bugs" object.
    - gammaHyperPlot() shows the histogram of a composite gamma prior density from four gamma hyperparameters or the posterior histogram from the "bugs" object.

5 Using `rube`: Details of a Typical Analysis

5.1 Writing Model Code

The first step in WinBUGS or JAGS analysis is writing a model specification (model code) file. You can also enter the model directly into a string variable (see above). Here are some tips for writing models in `rube`. The main goals are to improve clarity and maintain only a single model description rather than several (or many).

- Use the "#" sign where needed for commenting. It can be at the beginning of a line or in the middle. Everything after the "#" sign is ignored.
- You may use a semicolon (";") to enter multiple statements on a line. (Do this sparingly or you will sacrifice clarity.)
- As in R, you can break a long statement across multiple lines by carefully assuring that the statement(s) before the break cannot be interpreted as a valid statement, e.g., by ending with a plus sign ("+").
- In `rube` you also have the option of indicating that a line continues by placing two backslashes ("\\") at the end of the line. This is a safer way to continue lines, and may be the only clear way to conditionally add a term to an expression

using IFCASE. (Although WinBUGS supports almost any form of haphazard line breaking, `rube()` enforces a clearer set of forms based on the use of the semicolon, R-style incomplete lines, and the explicitly line continuation symbol.)

- Whenever you enter a hyperparameter, it is good practice to indicate that it is a default value by using the form `NAME=VALUE`, e.g., `PREC=0.001`. The use of capital letters is a common convention. Use separate names for values that might be changed individually, or a common name for values that will be changed en-mass. As an example if "myModel" contains

  ```
  yt[i] <- (z + log(y[i]) / FACTOR=20
  ```

  it will be interpreted as

  ```
  yt[i] <- (z + log(y[i]) / 20
  ```

  when you run `rube(myModel)`, but as

  ```
  yt[i] <- (z + log(y[i]) / 200
  when you run {\tt rube(myModel, subs=list(FACTOR=200))}.
  ```

- If you are making a coding choice more complex than a simple default use conditional coding. For example, to code a transformation as either log or square root use

  ```
  yt[i] <- (z + IFCASE(SQRT.XFORM) sqrt(y[i]) ELSECASE log(y[i]) ENDCASE) / FACTOR=20
  ```

  This is called the in-line form of conditional coding. For larger blocks of code (or if it seems clearer to you than the in-line form), the block form of conditional coding is used. For example:

  ```
  IFCASE(!LOGNORM)
  mu[i] ~ dnorm(tau[i], xi)
  ELSECASE
  tauE[i] <- exp(tau[i])
  mu[i] ~ dlnorm(tauE[i], xi)
  ENDCASE
  ```

  which will result in running the first line of code if "LOGNORM" is not defined in the `cases=` argument to `rube()` or similar functions, and the second and third lines if it is defined.

  This multi-line version of conditional coding allows any number of code lines in each block. You can have a single optional block by excluding the ELSECASE statement,

and you can have several alternate cases by including one or more ELSEIFCASE() statements.

For both types of conditional coding, you can have different identifiers inside different IFCASE() or ELSEIFCASE() statements and/or re-use the same identifiers. The identifiers are case sensitive and the choice of code is made at run-time by adding or excluding identifiers in the `rube(..., cases=)` statement. For example you might have `rube(...)` to use the standard normal (not defining LOGNORM) or `rube(..., cases="LOGNORM")` to use the log normal distribution, or `rube(..., cases=c("LOGNORM","OTHERS"))` in a more complicated situation. Also note that the condition can be simple like the `IFCASE(!LOGNORM)` above, or quite complicated such as `IFCASE(!LOGNORM || (OTHERS && !FOO))`.

- Any `rube` code may include the `LC()` "linear combination" pseudo-function to simplify model writing while flexibly allowing covariate choice and at the same time assure meaningful parameter names in model results.

  The syntax is:
  `LC(prefix, FORMULA, suffix, index)`
  This creates a linear combination of parameters and data based on the FORMULA as defined at run-time. In practice, you will replace the word FORMULA with some meaningful name (though FORMULA might be appropriate if there is only one `LC()` statement in your code). Again, upper case is a common convention. The other elements of the pseudo-function are entered verbatim (i.e, they are constants). The FORMULA is defined at run time in the `varList=` function option, and resembles the right hand side of a standard R model formula, i.e, it contains covariate names separated by "+", ":" (for interactions) and/or "*" (for main effects plus interaction). You may also use the R syntax `(A+B+C)^n` to indicate that you want main effects plus all interactions up to order n. Complex forms such as

  ```
  varList = list(FORM1 = "A*B + (C+D+E)^2 + A:C + (F+G+H)^2")
  ```

  are allowed.

  The linear combination is created by multiplying the variable names (with `[index]` appended if "index" is not blank) by the corresponding parameter names. The latter are created by concatenating the prefix, variable name from the FORMULA, and suffix. The suffix may be blank Interaction parameters are created by multiple concatenation, and interaction variables are created as products (so **warning:** they are only appropriate for two level categorical variables, or k-level variables recoded as k-1 indicators, or for quantitative variables). An intercept is created as "prefix0suffix" unless a -1 is included in the FORMULA.

  As an example

  ```
  mu[i] <- LC(b,FORM1,pop,j)
  ```

will be interpreted as

```
mu[i] <- b0pop + bApop*A[j] + bBpop*B[j] + bABpop*A[j]*B[j]
```

if `varList=list(FORM1="A*B")` is defined, and as

```
mu[i] <- bApop*A[j] + bBpop*B[j]
```

if `varList=list(FORM1="A+B-1")` is defined, and as

```
mu[i] <- b0pop + bApop*A[j] + bBpop*B[j] + bCpop*C[j] +
         bABpop*A[j]*B[j] + bACpop*A[j]*C[j] + bBCpop*B[j]*C[j]
```

if `varList=list(FORM1="(A+B+C)^2)")` is defined.

- The `rube` pseudo-function `FOR()` is used to create statements such as prior distribution definitions for all of the parameters created by an `LC()` formula (though it need not be limited to that use). The syntax is
`FOR(prefix, FORMULA, suffix, codeText)`
where prefix, FORMULA, and suffix are the same as for `LC()` and codeText is any valid WinBUGS or JAGS model declaration code, but with a question mark ("?") at the place where a parameter name is to be substituted at run-time. One statement is created for each parameter implied by the value of the FORMULA at run-time.

  Note that the `FOR()` statement must be the only statement on its line in the code.

  As an example

  ```
  FOR(b, FORM1, pop, ? ~ dnorm(0, POP.PREC=0.001))
  ```

  will be interpreted as

  ```
  b0pop ~ dnorm(0, 0.001)
  bApop ~ dnorm(0, 0.001)
  bBpop ~ dnorm(0, 0.001)
  ```

  if `varList=list(FORM1="A*B")` is defined, and as

  ```
  bApop ~ dnorm(0, 0.001)
  bBpop ~ dnorm(0, 0.001)
  ```

  if `varList=list(FORM1="A+B-1")` is defined.

- The `rube` pseudo-function `BASE()` can be used to create a baseline indicator variable for a formula. (This may have limited use, but I needed it for one project.) The syntax is:
  `BASE(FORMULA,index)`
  This creates a product of the form `(1-var[index])` for each main effect variable in the FORMULA. So

  ```
  isBase[i] <- BASE(FORM1, i)
  ```

  will be interpreted as

  ```
  isBase[i] <- (1-A[i]) * (1-B[i])
  ```

  if `varList=list(FORM1="A+B")` is defined.

5.2 Checking Models

The usual way to use `rube` is to first use the simple `rube(myModel)` form to perform initial syntax checking of your model. The problems detected include

- Not starting with `model` and not enclosing the model in curly braces.
- Unbalanced parentheses, brackets or braces.
- Misspelling a distribution name or using the wrong number of parameters. (Note: Standard WinBUGS or JAGS distributions plus, for WinBUGS, those in http://www.winbugs-development.org.uk/ are allowed.)
- Misspelling a function name or using the wrong number of parameters. (Note: Standard WinBUGS or JAGS functions plus, for WinBUGS, those in http://www.winbugs-development.org.uk/ are allowed.)
- Incorrect syntax of logical (assignment), stochastic, or for statements.
- Inconsistent use of vectors or matrices.
- Use of the same variables inside and outside "for" statements.
- Nesting of the same variable in "for" statements.
- Use of vector indices that are inactive (in another "for" statement).
- Invalid formulas, such as A + + B.
- Incorrect use of the censoring syntax I(,).

The object returned from a call to `rube()` is a "rube" object. If you `print()` the object, the data, constants, stochastics, and problems will be shown. The `summary()` gives additional information, namely the "for" variables, the initializations, and the concerns (possible problems), as well as including the actual code lines the caused each problem. You can also check the `$check$model` component which allows you to locate errors in context by line number. Note that line numbers refer to the processed file as reflected in "tempModel.txt" which is produced in the current directory. Due to the complex nature of the code pre-processor, `rube()` does not currently give the line number in your original unprocessed code file.

For all calls to `rube()` the options `subs=mySubsList`, `cases=myCaseVector`, and `varList=myVarList` may be used. Currently extra elements in these options are ignored. The subs and cases options are never needed: leaving them out gives the default values for substitutions (usually hyperparameters) and for the CASE selections. If any `LC()` and/or `FOR()` pseudo-functions are used, a missing varList value corresponds to an intercept only formula.

After your model is syntactically correct, it is a good idea to cross check the data and the initializations with the code. A function call that does not specify the parameters.to.save argument, such as `rube(myModel, data=myData, inits=myInits)` will create a summary showing the relationships between the model components. The objects myData and myInits are either lists, function calls, or file names of objects stored using `dget()`. Note that, by default, after running WinBUGS (bug not JAGS) through `rube` the data are in "data.txt", and the initializations are in "inits1.txt" in the working directory so, e.g., `inits=dget("inits1.txt")` will analyze your model using the last set of initializations.

If you use a function for the data, the function arguments may be whatever you like, and you can use the `rube()` optional argument `dataParams=myList` to provide non-default values in the form of a named list. Here is an example:

```
genData = function(nsubj, beta, sig=1) {
  x=runif(nsubj, 10, 30)
  y=rnorm(nsubj, beta[1]+beta[2]*x, sig)
  return(list(x=x, y=y, N=nsubj))
}
myModel="model {
  for (i in 1:N) {
    mu[i] <- beta0 + beta1*x[i]
    y[i] ~ dnorm(mu[i], prec)
  }
  beta0 ~ dnorm(0, 0.001)
  beta1 ~ dnorm(0, 0.001)
  prec ~ dgamma(0.01, 0.01)
}"
myInits = function() list(beta0=rnorm(1,0,0.5), beta1=rnorm(1,0,0.5),
```

```
                                prec=rnorm(1,1,0.1))
rube(myModel, genData, myInits, dataParams=list(nsubj=50, beta=c(0,1)))
```

Each time you run the `rube`() statement above, the `genData()` function is run once to generate a new dataset. The arguments to the `genData()` function come from the "dataParms" argument and the the value of sigma is taken from the function default (sig=1) because "sig" is not listed in "dataParams".

If you use a function for the inits, you may specify any of the following as arguments: "data" to get the data in list form, "cases" to get whatever is passed to the `rube`() function in the "cases" argument, and "extra" to get whatever is passed into `rube`() as the argument `startExtra=myStartList` to provide non-default values.

A call to `rube`() with the data and inits arguments will identify definite problems such as:

- Inconsistent variable dimensions
- Identical components of a matrix defined (i.e., not NA) for matrices present in both the data and the initializations.
- Data or initializations incompatible with their distributions.
- Data or initializations not present in the model (but this is not necessarily a problem if "culling" is in effect, as explained below).

In addition potential problems (labeled as "concerns") such as data or initializations that are unlikely under their distribution (outside of 3 s.d. from the mean) are flagged.

Finally various summaries of variables are produced (use summary() rather than print() to see them all) which will aid you in assessing the appropriateness of your model and in identifying the problem(s) if WinBUGS or JAGS fails.

There are separate sections for constants, data, and other stochastics. Among other things, the summaries of the data and other stochastic variables include, for each variable, the distribution name, and, when `rube` is able to determine them, the variable size (using RxC notation for matrices), the parameter values, the number of missing values, the censoring limits (if present), and theoretical mean and s.d. If the variable is data or if it is a variable with user initialized values, the initialized mean, s.d., and range are calculated. Finally "Flags" are calculated as follows:

- Values inconsistent with the distribution are flagged "!!!" and a listed as a critical problem (i.e, running WinBUGS or JAGS will not be attempted).
- Values outside of mean +/- 3 s.d. are flagged "**" and listed as a "concern".
- Values outside of mean +/- 2 s.d. are flagged "*".

For censored values, estimated means and s.d.s are produced by simulation (for some distributions).

Hints for detecting common problems: When looking for problems it may also be useful to examine the list of "for" variables and the list of uninitialized data. Also, you may find spelling errors for variables by noticing multiple version of what was intended to be one variable.

5.3 Running Models

To actually run a WinBUGS or JAGS model, use the function call `rube(myModel, data=myData, inits=myInits, parameters.to.save=myParameters)` where "myParameters" is a string vector of parameters that will be monitored by WinBUGS or JAGS and returned to `rube`() to be analyzed in R. (Unlike R2WinBUGS or R2jags, `"*"` may be used as the 'parameters.to.save' to save all parameteres.) If all goes well, the return object for this form of the `rube`() call is a "bugs" object as well as a "rube" object. In this case, the `summary()` call will show the results of the MCMC. If WinBUGS or JAGS fails, you will get an informative message, and a summary of the "rube" object will show the model check information instead of MCMC results.

The `rube`() function uses the same defaults as WinBUGS `bugs()` function or JAGS `jags()` function , but some of these are not very appropriate and interact with each other in strange ways. **I recommend that you control the number of iterations as follows.** Explicitly specify the arguments n.burnin to specify the total number of burn-in MCMC iterations to be discarded, n.iterations to specify the total iterations to run, and n.thin to specify than only every n.thin'th iteration will be saved after the burn-in.

By default three forms of **culling** are automatically turned on, but you do have the option of turning them off if desired. Data culling removes variables from the data if they are not used in the model. This is *very useful* because you do not need to manually adjust your data object when you change which covariates you are using. Without culling, WinBUGS or JAGS will fail if the data contain any variables not used in the model. You will get a `rube` warning when data are culled, but this warning may be suppressed with `cullWarn=FALSE`.

Similar culling is present for initializations, again simplifying the construction of initialization functions. In other words, if your intialization list contains parameters that are not in the model, `rube`() will remove them so that `bugs()` does not fail.

Culling of parameter.to.save serves the same purpose: you can use a variable containing all possible parameters, then let the culling process remove those that are not used based on the active varList covariate formulas.

You have some control over the way syntax checking is done when parameter.to.save is included (i.e., when WinBUGS or JAGS is run) thorough setting the `modelCheck` variable. The default is "always". You can set this to "onFail" or "never" to save run-time for a large set of simulations by suppressing some model checking. In the case of "onFail", the model checking will be done if the call to WinBUGS or JAGS fails, so that a `summary()`

of the result will be informative. Note that if you use the "never" setting no checking, and therefore no culling, is done.

The final "rube" object may be printed (the summary method currently just does printing) which gives a result similar to, but better than the print function that comes with R2WinBUGS or JAGS. The object may also be examined manually (usually through the $sims.array component and the $runTime and $startTime components), or examined with the `rube` functions `p3()`, `check()`, `model()`, `compare()`, and/or `priPost` (see below or the package help files for more information).

6 Details of Additional Functions

- The `getBugsExample()` function reads the model code, data, and initializations from a WinBUGS example ".odc" file. The required argument is a quoted string indicating which example you want. You may specify either the exact name of the example file (with or without the file suffix), or you may specify a portion of the file name in which case the first partial match will be returned. If you installed WinBUGS in a non-default location, use the "bugs.directory" argument to specify the correct location or use the BUGSDIR environmental variable as explained above "Environmental Variables".

  The return value is a list with components "model1", "data1", "inits1", and possibly additional, higher numbered components if the example has more that one model.

  The model component returned by `getBugsExample()` is a multi-line string which can be viewed using code such as:

  ```
  print(ali$model1)
  ```

  or

  ```
  model(ali$model1)
  ```

  Also, `getBugsExample()` corrects the model code to standard form with more reasonable line breaks when the original form is non-compliant with `rube` standards.

  The data component fixes matrices and arrays so that they will be in the correct "column major" order for R.

  The inits component is just a list, not a function, so these examples are more suitable for single chain analysis unless you write an initialization function.

  Note that in some cases `getBugsExample()` misidentifies the component, e.g., calling initializations as data, so some manual correction may be necessary.

  The simplest way to run an example is demonstrated here:

```
ali = getBugsExample("Aligator")
with(ali, summary(rube(model1, data1, inits1)))
result = with(ali, rube(model1, data1, inits1,
              parameters.to.save=c("alpha","beta"), n.chains=1))
print(result)
p3(result)
```

- The `p3()` function provides overview and detailed plots of the MCMC results returned to `rube`() from WinBUGS or JAGS through `bugs()`. Its required argument is a "rube" object. As shown in figure 1 an interactive plot is produced with information about one parameter (or one group of parameters) at the top and a menu at the bottom. The title identifies the parameter and its distribution.

  The menu shows clickable buttons for all monitored model parameters (i.e., those specified under "parameters.to.save" when running `rube`()), and it has a "Quit" button which is used to return to your interactive R session.

  For parameters with many values (i.e., vectors or arrays), the menu button shows the range of indices in square brackets. Clicking on any parameter button displays plots of the MCMC results for that parameter. For single-values parameters the "detailed mode" plot (see below) is presented when the parameter button is clicked. For multiple-valued parameters, the "summary mode" plot is initially displayed.

  `p3()` automatically creates parameter groups, and thus initially uses the "summary mode" for the related components of each LC() linear combination formula in your model. (The intercept, if any, is excluded from this group, because it is commonly on a different scale.) This is demonstrated in figure 1 by the "b[DEMOG]" button which corresponds to an LC() group which has prefix "b" and formula "DEMOG". In addition to the automatic creation of parameter groups, you can create groups manually using the "viewGroups=" argument to `p3()`, e.g., `viewGroups = list(main=c("bA","bB","bC"), aux=c("bT","bU"))` will produce two parameter groups labeled "main" and "aux", and these will each use summary mode to display the related parameters together.

  For single-valued parameters, the detailed mode plot consists of a trace of the MCMC values of each chain vs. iteration number. (The iterations shown are sequential regardless of the value of n.skip.) The colors in this panel only serve to distinguish the various chains.

  The second panel of the detailed mode plot shows the autocorrelation function for each chain, again using colors to distinguish chains. Dotted lines indicate confidence intervals for significantly non-zero correlations. If there are multiple chains, then the Gelman R-hat value is shown at the upper right hand corner using a color code of green for OK, orange for warning, and red for high (all based on the values of the "rHatGrayZone" which defaults to c(1,1.2)).

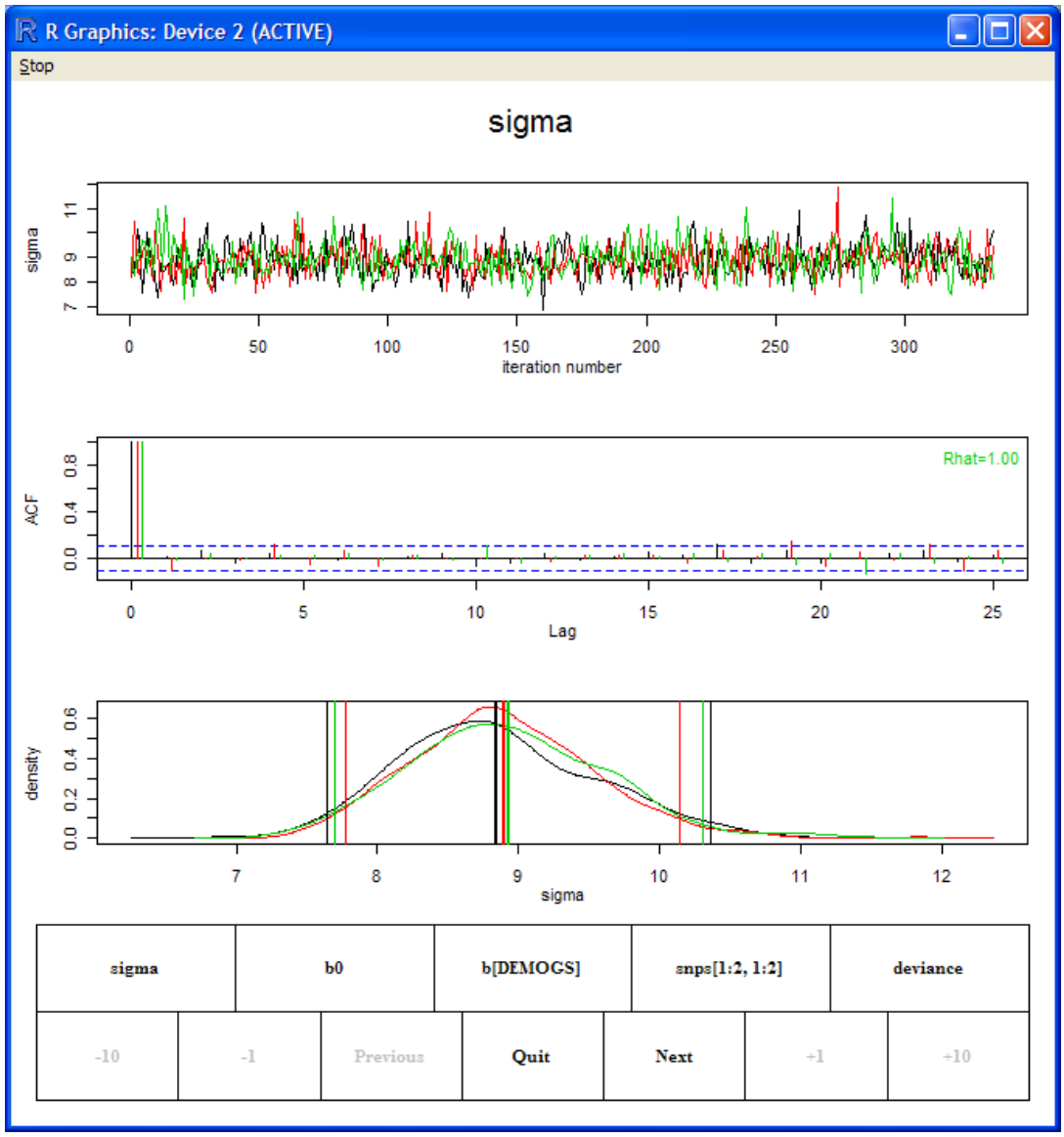

Figure 1: p3() showing detailed mode for a single-valued parameter.

The third panel of the detailed mode plot shows a density plot with the posterior credible interval marked based on "PIprob" which defaults to 0.95. If you supply values in some or all positions of the "true" argument vector, the corresponding plots will have that value marked with a dotted vertical line on the density plot. This is useful to MARK either a null value or the true value for simulated data.

For parameters with many values, clicking on the the menu button shows the summary plot, as in figure 2. The summary plot has two or three vertically arranged panels, depending on whether a single or multiple MCMC chains were run. The summary plot shows results for a random sample of parameter elements based on the "nSample" parameter (default is 20). Of course all elements are shown if the number of elements is less than "nSample". Clicking the button again shows a new random sample. The "intelligent parameter matching" feature assures that you switch to another parameter with the same number of elements, the same subset of elements is initially shown. You can just click again to see a different subset of elements.

The first panel shows the 95% posterior credible interval for each chain of each parameter element with different colors designating the different chains. The second panel shows the how many lags have significantly non-zero autocorrelation, again with different colors designating the different chains. If there are multiple chains, there will be a third panel showing Gelman's r-Hat for each parameter element (whether in the currently displayed parameter subset or not), with colors designating whether the value is high enough to indicated a mixing problem (rather than relating to individual chains). As for detailed value mode, the color code is green/orange/red for OK vs. gray zone vs. elevated values.

When the displayed parameter is multiple-valued, the menu buttons marked "-1", "-10", "+1" and "+10" can be used to enter detailed mode and to move through the detailed results of each individual element of the multiple-valued parameter.

The "Previous" and "Next" buttons move between parameters as an alternative to clicking on specific parameter buttons.

The "params" argument of `p3()` is useful for restricting the size of the menu when very many parameters are present or to select a single parameter for printing the plot with no menu present. You can enter either a string vector or a numeric vector. E.g., "alpha" plots only the alpha parameter. Or "10:15" examines only the 10th through the 15th parameters.

The "only" argument of `p3()` in conjunction with the "params" parameter plots only a single element of a single parameter or a set of specific elements for a vector parameter. The value is an (absolute) index into the parameter mentioned.

The "drop" argument removes values from the beginning of every chain and is useful if burn-in was insufficient. The value you supply is taken as the number of *recorded*

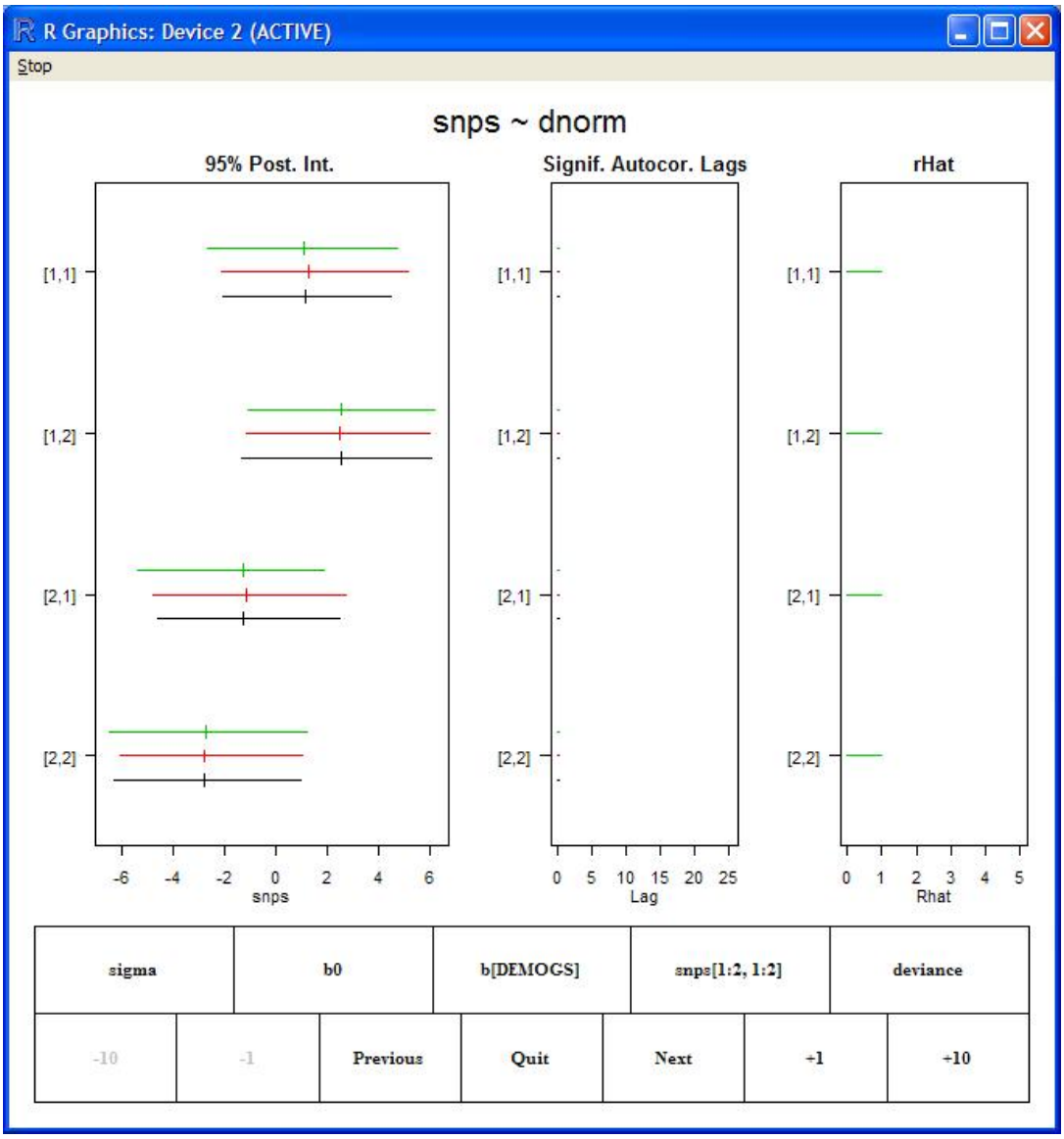

Figure 2: p3() showing summary mode for a multi-valued parameter.

values to drop, i.e., the same number of values is removed regardless of the value of n.burnin in the `rube`() call.

The "noMenu=TRUE" option suppresses the menu which allows you to print the `p3()` plots without the menu.

The "add=NULL" argument can be used to view the posterior of parameters that were not included in the WinBUGS or JAGS run, but which are computable from existing parameters. The argument must be a named list containing one or more expressions or strings defining expressions that include the names of existing parameters. Expressions may refer prior expressions in the list. Currently vector parameters are not allowed, and there is not yet a way to view the summary() of the new parameters.

As an example, consider the case where you save 'log.sd' and want to see the posterior of the sd itself. This is achieved with either of these forms:

```
p3(myRubeObject, add=list(sd="exp(log.sd)")
p3(myRubeObject, add=list(sd=expression(exp(log.sd)))
```

As another example, consider a logistic regression with covariate age centered at 40 and a male indicator variable. To see the posterior of the probability of success for a 50 year old male use:

```
p3(myRubeObject, add=list(Pr50M="1/(1+exp(-(b0+(50-40)*bage+bmale)))")
```

- The `compare()` function takes two or more "rube" objects and compares them graphically. The comparison is similar to the chain comparisons of `p3()` when multiple chains are present.

  The syntax is

  ```
  compare(rubeResult1, rubeResult2, ...)
  ```

  The optional parameter "chains" selects which chain from each "rube" object is used in the comparison (default is chain 1 from each model) and its length should be 1 or the number of objects in the list. The "params", "nSample", "true", "PIprob", "rHatGrayZone", and "noMenu" options are the same as for `p3()`.

  Note that parameters that appear in only one model are not accessible, and that the title will suppress the name of the distribution if it differs across models.

  If the MCMC runs being compared are not from the same model, a warning message is issued.

- The `merge(rubeResult1, rubeResult2, ...)` function takes two "rube" objects and merges them into a single object. Iterations are trimmed from the beginning of the longer MCMC simulation. The merging works as if the number of chains run is equal to the sum of the number of chains from the individual runs. Some information is (currently) lost, including the the DIC. The recomputed n.eff is computed using 'coda', but differ from the R2WinBUGS or JAGS result. The same applies to MCMCerr for R2WinBUGS, but new MCMCerr values is not implemented in R for JAGS objects.

- The `stitch(rubeResult1, rubeResult2, ...)` function takes two "rube" objects from the same model and with the same number of chains (possibly with differences in which parameters were saved) and merges them into a longer single object. The intention is that 'rubeResult2' is a continuation of 'rubeResult1', probably using 'inits=rubeResult1$last.values', and that there is no burn-in for the second run. Some information is (currently) lost, including the the DIC. The recomputed n.eff is computed using 'coda', but differs from the R2WinBUGS or JAGS results. The same applies to MCMCerr for R2WinBUGS, but new MCMCerr values is not implemented in R for JAGS objects.

- The `shrink(rubeResult, n.thin=1, drop=0, chainsDropped=NULL, ignore=NULL, keep=NULL, minimal=FALSE)` function takes a "rube" object and produces a smaller rubeObject, either to remove more burn-in, or to save space, e.g., by further thinning, dropping of chains, or dropping of $sims.matrix and $sims.list, and a few other fairly useless components of the rube (bugs) object. Some information is (currently) lost, including the the DIC. The recomputed n.eff is computed using 'coda', but differs from the R2WinBUGS or JAGS results. The same applies to MCMCerr for R2WinBUGS, but new MCMCerr values is not implemented in R for JAGS objects.

- The `augment(rubeResult, add=NULL, minimal=FALSE)` function takes a "rube" object and produces a rubeObject with additional parameters that are computable from the existing parameters. The 'add' argument is a named list, where the names are the formulas defining new elements. The formulas can be entered either as expressions (e.g., /tt expression(log(b0))) or as quoted strings (e.g., /tt "expression(log(b0)"). If the formulas refer to a vector parameter, a correpondingly sized vector of new parameters is created. Here is an example that assumes that 'varRI' is an existing scalar and that 'LO' is an existing vector (log odds for each subject). This function call creates a new /rube object that has a new scalar parameter called 'sdRI' and a new vector parameter of probabilities.

```
newRubeResult <- augment(oldRubeResult,
                         add = list(sdRI = expression(sqrt(varRI)),
                                    pr = "1/(1+exp(-LO))"))
```

Some information is (currently) lost, including the the DIC. The recomputed n.eff is computed using 'coda', but differs from the R2WinBUGS or R2bugs results. The same applies to MCMCerr for R2WinBUGS, but new MCMCerr values is not implemented in R for JAGS objects.

- `mergeLists()` is a convenience function that merges two lists of named values in such a way that
  - the result contains all element present in only one of the lists
  - for elements on both lists, the result contains the value from the second list.

  The main use is with "subs" and/or "varList" arguments to `rube`() and other functions with these arguments. For example if "v" is a varList of names model formulas for the "LC()" pseudo-functions of a model, including `DEMOGS="male+age"` among others, then

  ```
  rube(..., varList=mergeLists(v, list(DEMOGS="male*age")))
  ```

  will add in the interaction.

- `priorExplore()` is an interactive function to explore prior distributions graphically. It has no arguments. It initially has a menu of discrete distributions and a button to switch to continuous distributions. You can click to select a distribution and see its pdf or pmf graphically, along with mean and sd.

  Then you can click on the parameter slider bars to see the effects of changing parameter values. Click Quit to return to the normal interactive R prompt.

- `priPost()` makes a plot comparing the prior to the posterior for a particular model parameter. The syntax is

  ```
  function (post,
            dist = c("Normal", "Uniform", "Gamma", "Beta",
                     "sdFromGammaPrecision", "HyperBeta", "HyperGamma"),
            pripar,
            main = NULL, xlab = NULL, xlim = NULL,
            nsim = 2000, digits = 1, true = NULL)
  ```

  For "post" you need a vector of posterior values, normally in the form of the $sims.list["myParameter"]] component of a `rube` object. For a vector or array parameter you can follow this with an index or indices in square brackets. (As opposed to the sims.array form, the sims.list form includes all of the chains in a single vector.)

Unless the distribution is "Normal", you need to specify it in the "dist" argument.

The "pripar" argument is required and is a numeric vector including the parameters as follows:

- Normal: mean and standard deviation
- (Continuous) Uniform: lower and upper limits
- sd from gamma precision: gamma shape and scale parameters
- Gamma: shape and scale parameters
- Beta: alpha and beta parameters

The "sdFromGammaPrecision" takes a gamma prior on the precision parameter of a normal distribution, and plots it on the s.d. scale. The "post" value should be the posterior standard deviation, usually generated by a model statement such as `sd <- pow(precision, -2)`.

The HyperBeta prior refers to a beta prior distribution with independent gamma hyperpriors on alpha and beta. The "pripar" in that case is either a matrix with the alpha hyper-parameters in the first column and the two beta hyper-parameters in the second column, of a list for two length-two number vectors with the hyper-parameters.

The HyperGamma prior refers to a gamma prior distribution with independent gamma hyperpriors on the shape and scale. The "pripar" in that case is either a matrix with the shape hyper-parameters in the first column and the two scale hyper-parameters in the second column, of a list for two length-two number vectors with the hyper-parameters.

You can set the title with "main" and the x-axis label and limits with "xlab" and "xlim". For the hyperBeta and hyperGamma. the "nsim" argument is used to set the number of simulations used to generate the estimated distribution. The "digits" argument controls the number of digits in the display of the mean and s.d. The "true" argument allows specification of the true mean (for simulated data).

- `showDefaults()` is a convenience function that takes a model (as a file name or text string) as its argument and returns the names and values of all defaults of the "NAME=VALUE" form (see the substitution pseudo-function, above). Duplicates that have different default values are flagged. If the code uses the IFCASE pseudo-function(s) then defaults from all of the various conditional code are shown unless the "CASES=" argument is set as a string or string vector to select particular cases.

- `model()` is a generic convenience function that applies all of the pre-processing pseudo-functions based to the code specified by its "model" argument. It uses the "subs=", "cases=", and "varList=" arguments to produce a specific version of the model and display it. There are method functions defined for `model()` that apply

to `rube` objects, 'bugsCheck' objects and strings. The string may be either model code or the name of a file containing the model code.

Note that `rube`() operates by pre-processing the model code, writing it to a file (named tempModel.txt by default), then passing that file name to WinBUGS or JAGS, the most recently used pre-processed code is available for viewing in tempModel.txt. From R, you can view this file by running `model("tempModel.txt")` or my running `model()` on the `rube`() result itself.

- The `unlister()` function is a convenience function related to the WinBUGS or JAGS idea that ragged arrays of data in hierarchical models can most easily be handled by putting the data into vectors, then using a corresponding "index" vector to specify which upper level unit each value belongs to.

  The `unlister()` function takes a list called "data", and looks for an "unlist" element in the form of a named list of strings. Each string is a variable name in "data", and those particular elements of data, which are presumably lists, are converted to a vector with unlist, and then renamed to the name in the unlist element that matches the variable.

  This procedure, which only seems complicated, allows generation of a data object that contains ragged arrays in list form, and then manual or automatic (see "special features of data generating functions", below) conversion to the "indexed" form.

  Here is code for a simple example:

```
N = 10
t1 = sample(5:7, N, rep=TRUE)
t2 = sample(9:12, N, rep=TRUE)
x = apply(cbind(t1,t2), 1, function(tt) tt[1]:tt[2])
id = rep(1:N, sapply(x,length))
reg = function(x, b, sig) {
        y = rnorm(length(x), b[1] + b[2]*x, sig)
        return(round(y,2))
      }
y = lapply(x, reg, b=c(1,3), sig=2)
myData = list(N=N, id=id, x=x, y=y, unlist=list(X="x", Y="y"))
sapply(myData,length)
#     N     id      x      y unlist
#     1     54     10     10      2
print(myData)
print(cbind(myData$id,unlister(myData)$Y)[1:4,])
#     [,1]  [,2]
#[1,]    1 16.66
#[2,]    1 20.58
```

```
#[3,]    1 26.28
#[4,]    1 27.33
```

- Full details of calling `rube`()

  Here is the full list of `rube`() parameters with defaults and explanations.

  - model: the model definition code. This can be a text string with on or more newlines, or the name of a file containing the model code.
  - data=NULL: the WinBUGS or JAGS data. This can be a named list of data values (possibly including NAs), a data.frame, the name of a file containing data saved using `dput()` or a function that generates a named list of data values. A data.frame will be converted to a list of its columns plus "N" for the number of rows. The file name form is useful to supply the data most recently used by `rube`() which is in "data.txt" by default. The function form is useful for simulating multiple datasets. See "Special features of data generating functions", below. If data culling is in effect (the default case), then any supplied data variable not needed by the current model will be automatically dropped to prevent a WinBUGS or JAGS error. This allows use of a single data list for different models with different covariates.
  - inits=NULL: the initial parameter values for the MCMC run. This can be a named list of data values (possibly including NAs), the name of a file containing data saved using `dput()` or a function that generates a named list of initial values. The file name form is useful to supply the chain initializations most recently used by `rube`() which is in "inits1.txt", "inits2.txt", etc. by default. For details on the function form of "inits", see "Special features of inits functions", below. If inits culling is in effect (the default case), then any initializations that refer to parameters not in the model will be automatically dropped to prevent a WinBUGS or JAGS error. This allows use of a single inits list or function for different models with different parameters.
  - parameters.to.save=NULL: the MCMC parameters that are to be saved and returned to the user. This is a vector of string names of parameters (without use of "[]"). Setting these values rather than leaving them blank is what changes the behavior of the `rube`() function from model checking to running of the MCMC. If PTS culling is in effect (the default case), then any values that refer to non-parameters will be automatically dropped to prevent a WinBUGS or JAGS error
  - n.chains=3: the number of MCMC chains to run. This defaults to 3. Each chain is given a different set of initial values if "inits" is a function or a list of lists. Otherwise a warning is given (unless n.chains=1) that multiple chains have the same initial values.

- n.iter=2000: the total number of MCMC iterations per chain. This number includes burn-in iterations and non-saved iterations (if n.thin>1).
- n.burnin=n.iter/2: the number of initial iterations that are discarded as burn-in.
- n.thin: the thinning rate. Only one of every n.thin iterations is saved and the rest are discarded.
- n.sims: (see bugs() documentation; I recommend using n.iter, n.burnin, and n.thin instead of n.sims)
- bin: the number of saved (not burnin and not thinned) iteration before each disk write. Warning: currently this overrides all of the above, so setting it too large may produce a very lengthy run.
- debug=FALSE: sets the debug state of `bugs()`. Generally there is no benefit to setting debug=TRUE, because `rube`() will read the state of `bugs()` if it fails. Sometimes `rube`() will recommend turning debug=TRUE on if the state of `bugs()` is not readable. This parameter does not affect JAGS.
- modelCheck="always": the conditions under which model checking is performed. The default is "always". The "never" option can speed up multiple simulations, but culling is not available, and model checking information is not available on failure. The "onFail" option runs full model checking only when the MCMC fails, but always runs simplified checking to the degree necessary to perform culling, if culling is requested.
- cullData=TRUE: a logical flag indicating whether or not culling of unused data is performed. (If un-referenced data are passed to WinBUGS or JAGS it will fail.)
- cullInits=TRUE: a logical flag indicating whether or not culling of unused initializations of stochastic parameters is performed. (If un-referenced initializations are passed to WinBUGS or JAGS it will fail.)
- cullPts=TRUE: a logical flag indicating whether or not culling of parameters.to.save is performed. (If un-referenced parameters.to.save are passed to WinBUGS or JAGS it will fail.)
- cullWarn=TRUE: a logical flag indicating whether or not culling should generate a warning message.
- DIC=TRUE: a logical flag indicating whether or not DIC should be calculated
- digits=5: the number of significant digits in the output
- codaPkg=FALSE: (see `bugs()` documentation) `Rube` won't fully work with codaPkg=TRUE.
- subs=NULL: a named list of substitutions for default values of the form "NAME=VALUE" in the model code

- cases=NULL: a string vector of case variables that will be defined as TRUE, then used to evaluate IFCASE() and ELSEIFCASE() pseudo-functions in the model code
- varList=NULL: a named vector of model formulas (right hand side only) for substitution into the LC(), FOR(), and BASE() pseudo-functions in model code
- bugs.directory: the location of WinBUGS. This defaults to `C:\\Program Files\WinBUGS14` or the contents of the Windows environmental variable BUGSDIR if it exists.
- useWINE, WINE, newWINE, WINEPATH: (see `bugs()` documentation) untested in `rube`
- program="auto": the MCMC program. Under auto, jags is used if package R2jags is loaded, otherwise WinBUG is used. Set to "WinBugs" or "jags" to override "auto". (OpenBugs is untested.)
- parallel=FALSE: with jags, parallel=TRUE runs jags in parallel if the inits= value is a function.
- progress.bar="gui" sets the jags (non-parallel) progress bar (see jags documentation)
- RNGname="Wichmann-Hill" set the jags random number generator (see jags documentation)
- wd=getwd(): the current working directory. This is the location that files are read from and written to. It is easiest to run `rube` from the directory containing your data files and model files. The default is for `rube` to first check the environmental variable BUGSWD, which becomes the working directory if it is defined and 'wd' is not specified. On some networks WinBUGS may not be able to see the getwd() directory (e.g., if the getwd() return value starts with four slashes). In that case specify a normal Windows directory starting with a drive letter and a colon. The symptom of this problem is that WinBUGS starts but hangs instead of running your model.
- bugs.seed, over.relax: (see `bugs()` documentation)
- dataParams=list(): a named list of arguments to be passed to the data generating function when "data" is a function
- initsExtra=NULL: a names list of arguments to be passed to the initialization function when "inits" is a function
- warnUseless=FALSE: a logical flag to indicate whether useless "varList" and "subs" elements should generate a warning.
- modelFile="tempModel.txt": the name of the file that contains the processed model after all pseudo-functions are processed. This file will be produced in the working directory and passed to WinBUGS or JAGS by name. Examining this file is a good way to see all of the results of model pre-processing.

- ignore=NULL: a string vector of partial error messages to be ignored. This is a "safety valve": if `rube()` list problems and indicates that it will not run WinBUGS or JAGS, you can include (part of) the error message(s) in the ignore vector to force `rube()` to call WinBUGS or JAGS.

- **Special features of data generating functions** make it easier to run simulation studies. When simulating data to test a Bayesian model, you might want to make a data generating function that has several parameters that specify custom data generation options and then create a data.frame that has columns for the options with one row for each data setup you want to simulate. With such a setup, code like

  ```
  result = rube(mdl, datFun, initFun, PTS, dataParams=myDtf[i,])
  ```

  in a for loop would run each simulation.

  To accommodate hierarchical models with ragged arrays using the indexing approach (see `unlister()`, above) the `unlister()` function is run on the data whenever an "unlist" element is present. This option exists because it is often useful to keep the data in the ragged array format except when used as MCMC data, e.g., for plotting the data or for data-based initialization functions.

  There is a mechanism for including additional "housekeeping" information in a data object. Anything stored in a list element named "drop" will be removed from the data, and then added to the final `rube` object as an "extra" element. This is useful in simulation studies, e.g., to store true parameters values with the data, which are then available for viewing in the final `rube` object along with the MCMC results.

- **Special features of inits functions** make them very flexible. The `inits()` function is used to create a named list of initial values for model parameters. The function may have no arguments or any combination of "data", "cases", and "extra". `Rube()` will supply its data, cases, and initsExtra arguments as needed when it runs the inits function. The function will be run once for each MCMC chain (as specified by the n.chains argument to `rube()`).

  Here is an example that assumes that the data contain "x" and "score" values, that a switch to a log scale is indicated by a "LOGSCALE" value in the cases vector (presumably also used in `IFCASE(LOGSCALE)` in the model code), and with an optional specification of `OVERDISPERSE=TRUE` in the `rube()` "initsExtra" argument.

  ```
  initReg = function(data, cases, extra) {
    y <- data$score
    if ("LOGSCALE" %in% cases) {
      y <- log(y)
    }
  ```

```
  RMSE = summary(lm(y ~ data$x))$sigma
  sd <- 0.2
  if (!is.null(extra$OVERDISPERSE)) sd <- 2
  return(list(sigma=max(0.02, rnorm(1, RMSE, sd)), mu=rnorm(1, 0, sd)))
}
```

So some alternate typical calls might be:

```
rube(myModel, myData, initReg, myParameters, cases="LOGSCALE",
     initsExtra=list(OVERDISPERSE=TRUE))
rube(myModel, myData, initReg, myParameters)
rube(myModel, myData, initReg, myParameters, cases="LOGSCALE")
rube(myModel, myData, initReg, myParameters, initsExtra=list(OVERDISPERSE=TRUE))
```

7 Problems/Suggestions

Please try this package out, and let me know of any problems or suggestions using the email address hseltman at cmu.edu.