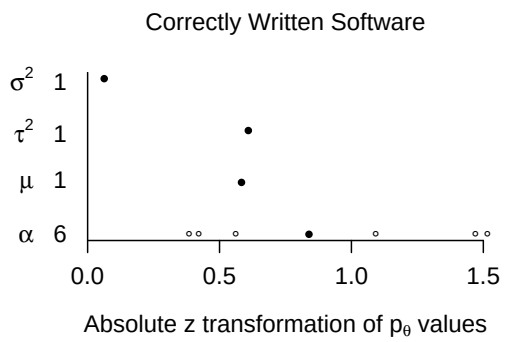

Figure 1: *z_θ statistics: Correctly written software. Each row represents a scalar parameter or batch of parameters; the circles in each row represent the z_θ statistics associated with that parameter or batch of parameters. Solid circles represent the z_θ statistics associated with the mean of that batch of parameters. The numbers on the y axis indicate the number of parameters in the batch. The z_θ statistics are all within the expected range for standard normal random variables.*

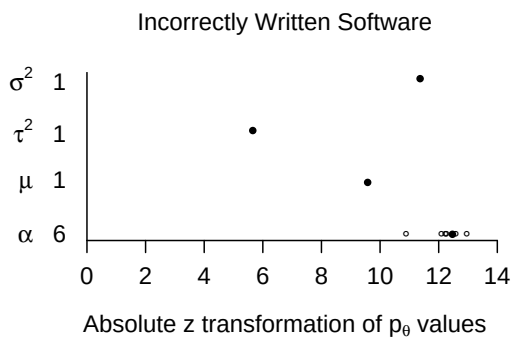

Figure 2: *z_θ statistics: Incorrectly written software (error sampling the parameter α). Each row represents a scalar parameter or batch of parameters; the circles in each row represent the z_θ statistics associated with that parameter or batch of parameters. Solid circles represent the z_θ statistics associated with the mean of that batch of parameters. The numbers on the y axis indicate the number of parameters in the batch. Values of z_θ larger than 2 indicate a potential problem with the software; this plot provides convincing evidence that the software has an error.*

Bibliography

S. Cook, A. Gelman, and D. B. Rubin. Validation of software for Bayesian models using posterior quantiles. *Journal of Computational and Graphical Statistics*, 2006. To appear. 11

J. Geweke. Getting It Right: Joint Distribution Tests of Posterior Simulators. *Journal of the American Statistical Association*, 99:799–804, 2004. 11

Samantha Cook, Andrew Gelman
Department of Statistics
Columbia University, NY, USA

Making BUGS Open

by Andrew Thomas, Bob O'Hara, Uwe Ligges, and Sibylle Sturtz

BUGS[1] (Bayesian inference Using Gibbs Sampling, Spiegelhalter et al., 2005) is a long running software project aiming to make modern Bayesian analysis using Markov Chain Monte Carlo (MCMC) simulation techniques available to applied statisticians in an easy to use Windows package. With the growing realization of the advantages of Open Source software we decided to release the source code of the BUGS software plus full program level documentation on the World Wide Web[2]. We call this release OpenBUGS (Thomas, 2004). We hope the BUGS user community will be encouraged to correct, improve and extend this software.

We follow a brief outline of how the BUGS software works with a more detailed discussion of the software technology used during the development of BUGS. We then try and explain why BUGS was developed using non-standard tools. We hope to convince the reader that although unfamiliar, our tools are very powerful and simple to use.

[1] http://www.mrc-bsu.cam.ac.uk/bugs/
[2] http://mathstat.helsinki.fi/openbugs/

Much of the ease of use of the BUGS software comes from its graphical user interface and the idea of the compound document as a container for different types of information. However, much is to be gained by interfacing BUGS with other software. R has many useful built-in and contributed functions but as yet little in the way of Bayesian analysis tools. The **BRugs** interface to the BUGS software plus a small suite of R functions is an attempt to improve this situation.

All of these (**BRugs** interface, the whole OpenBUGS software, and the R functions) have been organized for the R users' convenience in an R package also called **BRugs**. This package is distributed over the CRAN network. Its current version (0.2-5) is only available for Windows.

In R, the user with Internet connection can simply type

```
R> install.packages("BRugs")
R> library("BRugs")
```

and then happily start sampling, benefiting from the strengths of both OpenBUGS and R.

The **R2WinBUGS** package by Sturtz et al. (2005) already provides an approach to connecting BUGS and R. This has the disadvantage that it is impossible to interact during processing/sampling by WinBUGS in any way. If you need Gibbs sampling in R on other operating systems than Windows, we recommend to take a look at JAGS (Just Another Gibbs Sampler) by Plummer (2005).

How BUGS works

The software creates lots of objects, wires the objects together and then gets the objects to talk to each other. More formally a dynamic data structure, a directed acyclic graph, of objects is build to represent the Bayesian model. This graph is able to exploit conditional independence assumptions to efficiently calculate conditional probabilities. A layer of updater objects is created to sample parameters of the model and copy them into the graph data structure. Finally a layer of monitor objects can be created to monitor (watch) the values of the sampled parameters and provide summary statistics for them.

How is the graph of objects built? The user writes a description of the Bayesian model in the BUGS language. This model description is also a description of the graph of objects that BUGS should build. A compiler turns the textual representation of the Bayesian model into the graph of objects. Objects of base class 'updater' have a method which is able to decide if objects of that particular class can (and should) act as updaters for a particular parameter in the model based on the functional form of its conditional distribution.

Compilation and inference

Compilation of the description of a Bayesian model in the BUGS language involves a number of stages. Firstly lexical analysis, scanning, is performed to break the stream of characters representing the model into tokens. Secondly syntactical analysis, parsing, is performed to build a tree representation of the model. Thirdly the graph of objects is constructed by a post order traversal of the parse tree with objects whose values have been observed marked as data. Finally conditional independence is used to produce lists of graph objects that when multiplied together calculate conditional distributions.

BUGS uses MCMC simulation algorithms to make inference. These algorithms are computationally expensive but robust to details of the problem they are applied to. This robustness is an important property in a system such as BUGS which automatically chooses the inference algorithm. BUGS is able to match a wide choice of MCMC algorithm, such as single site Gibbs, slice sampling and continuously adapting block Metropolis, to the model parameters that need updating.

Software development

BUGS is written in the language Component Pascal (CP) using the BlackBox Component Builder from *Oberon microsystems*[3]. CP is a very modern compiled language with both modular and object orientated features. The language is highly dynamic with runtime loading and linking of modules. Compiled modules contain meta information that allows the module loader to verify that the loaded module provides the services required by the client. It is also an extremely safe language because of its very strong type system and automatic heap management (garbage collection).

CP software typically consists of many unlinked modules plus a small executable or dynamic link library that is able to load modules as required. The modules are arranged as a directed acyclic graph under the import (make use of) relation. Loading a module causes all modules in the sub-graph to be loaded. Module initialization code is executed when a module is loaded. Modules are grouped into subsystems with the subsystem name used as a prefix to the module name. Physically modules are represented by files with the location and name of the file derived from the module name. Each subsystem is kept in a separate subdirectory while the executable (or dynamic link library) is kept in the root directory.

The BlackBox Component Builder comes with several subsystems of modules which make the development of graphical user interfaces simple. More

[3] http://www.oberon.ch/

novel is the idea of a compound document, an editable text document that is able to contain graphics views. Graphics views can be developed by extending a view class. Graphics views can be made editable and special purpose drawing tools such as DoodleBUGS can be easily developed. About one quarter of the modules comprising OpenBUGS implement the graphical user interface and various graphics views used for output.

These GUI modules are only available for 32-bit Windows. The package **BRugs** does not make use of any GUI module. All other modules in the OpenBUGS distribution can be used under Linux on x86 based platforms as well.

Metaprogramming

Metaprogramming is self awareness for software. Software can ask itself questions. For example does module `Foo` export an item called `Bar`? What sort of item is `Bar`? Can such a thing be done with `Bar`? More formally we can ask if a particular module is loaded. If the module is loaded we can examine its metadata and then query this metadata. For example we could ask if a module `Foo` is loaded and if not load the module. Then we could ask if module `Foo` contains a procedure `Bar` with say no parameters and if so to call (execute) this procedure. Note that this process is safe: we do not just hope that `Foo` contains a `Bar` of the right sort (with a crash if this is not so).

BUGS makes use of metaprogramming in many places. These uses of metaprogramming fall into two broad groups: program configuration and interfacing. In the first group are support for the BUGS language, loading sampling algorithms and loading data reading algorithms. In the second group construction of GUI interfaces, implementing a scripting language and interfacing to R.

Each time the BUGS language parser comes across the name of a distribution it uses metaprogramming to load the module that implements this distribution. The link between distribution name and module name is stored in a configuration file called 'grammar'. A list of modules implementing sampling algorithms is stored in a file. When BUGS starts up this file is read and the appropriate modules are loaded. Currently BUGS can read data in two formats: the S-PLUS (Insightful Corporation, 2004) format and rectangular format. Again the modules that implement reading these formats are loaded at program start up. Other data reading option such as from SQL tables could be added.

Metaprogramming makes construction of the widgets typical of a GUI simple. For example a button is just a region of a window which responds to a mouse click by executing a procedure (without parameters). A string containing the module and procedure names is associated with the button and when the mouse is clicked metaprogramming is used to load the module and execute the procedure. Note in this approach no code is written to represent the button.

In a scripting language, typing a command at a prompt causes the system to execute some action. This involves some sort of interpreter. This is easily written using metaprogramming. The command in the scripting language is a string which is mapped into a series of procedures in the CP language. Metaprogramming is then used to load and execute these procedures. For example the command `modelCheck(^0)` in the BUGS scripting language gets mapped to

```
BugsCmds.SetFilePath('^0');
BugsCmds.ParseGuard;
BugsCmds.ParseFile
```

where `^0` is a holder for a string. The mapping between commands in the BUGS scripting language and the corresponding CP procedure is stored in a file, making the language extensible.

BRugs: Interfacing to R

The R interface to OpenBUGS is realized by a very small dynamic link library 'brugs.dll' corresponding to the 'WinBUGS.exe'. It exports a couple of `.C()` entry points, among those several for direct access to the BUGS scripting language. This way, it is possible to realize R functions that are very similar to commands in the BUGS scripting language, not only sharing the same names (e.g. `modelCheck()`) but also sharing almost the same (order of) arguments. Therefore, it was possible to implement a huge number of R functions that allow almost full control of OpenBUGS in R.

Commands and some data are passed directly from R to OpenBUGS by `.C()` calls. Some information is passed back from OpenBUGS to R as the value from these calls, as it is common practice in R programming and interfaces. Unfortunately, we still have to pass back some other information and results of sampling using temporary text files that are imported into R by `readLines()`, `read.table()`, `scan()` and friends. Transparently reporting error messages from OpenBUGS to the R user is another topic that needs further improvement – currently we are sometimes relying on a good guess for generating error messages.

BRugs provides at least five kinds of functions:

- basic functions (such as `modelCheck()`) corresponding to the BUGS scripting language mentioned above,
- functions (e.g. `write.datafile()`) to prepare R data and inits (in the form of dataframes, for example) for OpenBUGS adapted from the **R2WinBUGS** package (Sturtz et al., 2005),

- high level functions such as `BRugsFit()` which run a whole simulation using only one function call,
- functions (e.g. `buildMCMC()`) to prepare the data for output analysis using the **coda** package (Plummer et al., 2005), and
- some internal help functions to read the temporary buffer file, for example.

Using these functions, it is possible to run an interactive sampling and analysis session in R where you can sample, calculate some (intermediate) results and make convergence diagnostics, and sample further on if required.

For example, Weihs and Ligges (2006) used this capability of **BRugs** for some MCMC optimization in the following manner. In principle, after each 50 or 100 iterations (of OpenBUGS), the convergence of the error rate of the underlying model was calculated using linear regression (in R). If the coefficient was no longer significantly negative (i.e. convergence of the error), the extremely computational expensive iterations could be stopped, otherwise iterations continued in OpenBUGS again.

A BRugs session

For demonstration of the use of **BRugs** we use a normal hierarchical model for the rats data that is used throughout the WinBUGS manual (Spiegelhalter et al., 2005). The example is originally taken from section 6 of Gelfand and Smith (1990).

The WinBUGS manual is available in HTML format documentation from within R by calling `help.WinBUGS()`. Analogously, `help.BRugs()` starts up the **BRugs** manual (Thomas, 2004). For references on R functions, the usual help files such as `?help.BRugs` for function `help.BRugs()` itself are available.

After loading the **BRugs** package by

```
R> library(BRugs)
```

we change the working directory to simplify file specification in the next steps:

```
R> oldwd <- getwd()
R> setwd(system.file("OpenBUGS", "Examples",
+                    package = "BRugs"))
```

To initialize a model, the user types functions corresponding to the BUGS scripting language instead of clicking buttons. First, the model has to be checked. The model file for the rats model is given by 'ratsmodel.txt':

```
R> modelCheck("ratsmodel.txt")
```

Of course, it is also possible to specify the file by the absolute path (using forward slashes). Afterwards, data have to be loaded by `modelData()`. This function takes a file name as argument. R objects (named list of data or a vector or list of object names) can be written to such a file using `bugsData()`. For example:

```
R> data(ratsdata)
R> modelData(bugsData(ratsdata))
```

If data are stored in more than one file, the argument can be a vector of files as well or the function has to be called successively.

Now it is time to compile the model. In this example, we use three chains to run the MCMC simulation.

```
R> modelCompile(numChains = 3)
```

Initial values can be specified by calls to the function `modelInits()`. For more than one chain, one can either call `modelInits()` with a character vector of more than one filename (one for each chain) or call the function successively for each file containing initial values. For random effect nodes, the function `modelGenInits()` can generate appropriate inits.

In order to write files that contain initial values as accepted by OpenBUGS and the `modelInits()` function, the function `bugsInits()` can be used. Its argument is a list with one element for each chain. Each element of this list is itself a list of starting values for the OpenBUGS model, *or* a function creating (possibly random) initial values.

Therefore, we demonstrate the use of these three different approaches to specify initial values, one for each chain:

```
R> data(ratsinits)
R> modelInits("ratsinits.txt")
R> modelInits(bugsInits(list(ratsinits),
+    fileName = tempfile()))
R> initfoo <- function() {
+    list(
+      alpha = rnorm(30, mean = 250, sd = 1),
+      beta = rnorm(30, mean = 6, sd = 1),
+      alpha.c = runif(1, 140, 160),
+      beta.c = 10,
+      tau.c = 1,
+      alpha.tau = 1,
+      beta.tau = 1)
+ }
R> modelInits(bugsInits(initfoo,
+                        fileName = tempfile()))
```

The model is initialized now and we start with 1000 updates as a burn-in period:

```
R> modelUpdate(1000)
```

By default, sampled parameter values are discarded by WinBUGS after each iteration unless the user explicitly requests that the values are stored for later use. This is done with the `samplesSet()` function before running the simulation for further 2000 iterations.

```
R> samplesSet(c("alpha", "beta"))
R> modelUpdate(2000)
```

To analyse the results of this simulation, we can take a look at the summary statistics, similar to clicking `stats` in the Sample Monitor tool within WinBUGS.

```
R> samplesStats("*")
```

An asterisk (`"*"`) can be entered instead of a node name as shorthand for all the stored samples.

All these calls can be performed conveniently by a single call to the meta function `BRugsFit()`:

```
R> BRugsFit(data = ratsdata,
+    inits = initfoo,
+    para = c("alpha", "beta"),
+    nBurnin = 1000, nIter = 2000,
+    modelFile = "ratsmodel.txt", numChains = 3,
+    working.directory =
+      system.file("OpenBUGS", "Examples",
+                  package = "BRugs"))
```

It returns a list containing the summary statistic as `samplesStats()` as well as the Deviance Information Criterion (DIC, Spiegelhalter et al., 2002), a Bayesian extension of the Akaike Information Criterion to hierarchical models. The DIC can also be imported into R by the low level functions `dicSet()` (for setting) and `dicStats()` (for getting).

`BRugsFit()` is only one wrapper function summarizing a couple of functions from the whole **BRugs** framework. Users might want to come up with their own wrapper functions fitting their own purposes, or some plot functions appropriate for their analyses, for example based on the code of `BRugsFit()`.

Plots known from WinBUGS are also provided by **BRugs**, for example history of the simulation (`samplesHistory()`), plots of autocorrelations (`samplesAutoC()`), plots of smoothed kernel density estimates (`samplesDensity()`), etc. Of course, graphical parameters may be passed as additional arguments to these plot functions.

As an example, we plot the smoothed kernel density estimates for the first 6 components of node `"alpha"` (figure 1):

```
R> samplesDensity("alpha[1:6]")
```

To finish this example session, we reset the working directory by

```
R> setwd(oldwd)
```

Outlook

OpenBUGS has made modern Bayesian inference software available in an Open Source package. The software is also open in the sense that it has been designed so that new features such as distributions, sampling methods, and user interfaces can be easily added. An OpenBUGS user has already contributed Component Pascal modules to implement the generalised extreme value and generalised pareto distributions.

The R package **BRugs** links components of OpenBUGS into R. This allows users to combine the strengths of both applications and make use of them interactively.

Unfortunately, currently **BRugs** is only available for Windows. We hope to provide a Linux version of BRugs shortly along with Component Pascal development tools for Linux. Until these tools become available to the public there is no alternative to distributing binary versions of the package for Windows and Linux.

Acknowledgments

The work of Uwe Ligges has been supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 475.

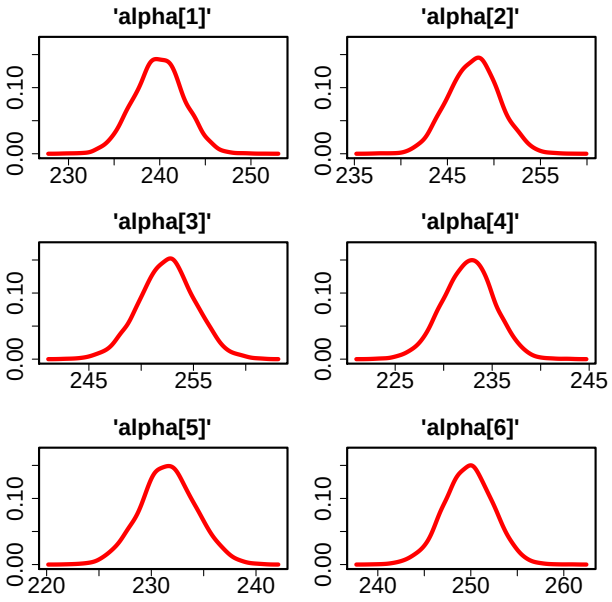

Figure 1: Density plot for the first 6 components of node `"alpha"`.

Bibliography

A. Gelfand and A. Smith. Sampling-based Approaches to Calculating Marginal Densities. *Journal of the American Statistical Association*, 85:398–409, 1990. 15

Insightful Corporation. *S-PLUS 6.2*. Insightful Corporation, Seattle, WA, USA, 2004. URL http://www.insightful.com. 14

M. Plummer. *JAGS: Version 0.90 manual*, 2005. URL http://www-ice.iarc.fr/~martyn/software/jags/. 13

M. Plummer, N. Best, K. Cowles, and K. Vines. *coda: Output Analysis and Diagnostics for MCMC*, 2005. R package version 0.10-3. 15

D. Spiegelhalter, N. Best, B. Carlin and A. van der Linde. Bayesian Measures of Complexity and Fit. *Journal of the Royal Statistical Society/B*, 64:583–639, 2002. 16

D. Spiegelhalter, A. Thomas, N. Best, and D. Lunn. *WinBUGS: User Manual, Version 2.10*. Medical Research Council Biostatistics Unit, Cambridge, 2005. 12, 15

S. Sturtz, U. Ligges, and A. Gelman. R2WinBUGS: A Package for Running WinBUGS from R. *Journal of Statistical Software*, 12(3):1–16, 2005. URL http://www.jstatsoft.org/. 13, 14

A. Thomas. *BRugs User Manual, Version 1.0*. Dept of Mathematics & Statistics, University of Helsinki, 2004. 12, 15

C. Weihs and U. Ligges. Parameter Optimization in Automatic Transcription of Music. In M. Spiliopoulou, R. Kruse, A. Nürnberger, C. Borgelt, and W. Gaul, editors, *From Data and Information Analysis to Knowledge Engineering*, pages 740–747, Berlin, 2006. Springer-Verlag. 15

Andrew Thomas, Bob O'Hara
Department of Mathematics & Statistics
University of Helsinki, Finland
ant@rni.helsinki.fi, bob.ohara@helsinki.fi

Uwe Ligges, Sibylle Sturtz
Fachbereich Statistik, SFB475
Universität Dortmund, Germany
<ligges,sturtz>@statistik.uni-dortmund.de

The BUGS Language

by Andrew Thomas

The BUGS language is a computer language not unlike the S language (Becker et al., 1988) in appearance, but it has a very different purpose.

Statistical models must be described before they can be used. A language to describe statistical models is needed by both the users of the model and the software that makes inference about the model. The language should be a formal language with well defined rules which can be processed automatically. It should not be concerned with the technology used to make inference about the model. We have developed a model description language called the BUGS language because of its use in the **B**ayesian inference **U**sing **G**ibbs **S**ampling (OpenBUGS) package. However, the BUGS language can be used outside the OpenBUGS software. For example, it is used in the JAGS package (Plummer, 2005) and has influenced other packages such as Bassist (Toivonen et al., 1999) and AUTOBAYES (Fisher and Schumann, 2003).

We choose to describe statistical models in terms of a joint probability distribution. Model description in terms of a joint probability distribution is both very general and very explicit. We consider these good points. We do not consider it a good idea to have a patchwork of specialized (maybe very elegant) notations for different types of model. We want to be able to combine small submodels to build larger models using a consistent notation. A small change to a model should not lead to a large change in the way that model is described. Examples of small changes to the model are: choice of sampling distribution, form of regression, covariate measurement error, missing data, interval censoring, *etc*. Explicitness is important in a model description language. There should be no doubt if two models are the same.

We hope the BUGS language will be useful to anyone who uses complex statistical models, and even to people who do not want to use the OpenBUGS package to make inference. Is the BUGS language really about statistics? OpenBUGS has many users who do not think of themselves primarily as statisticians, who are mainly interested in the deterministic skeleton of a model. We think that if a probabilistic model is used to explain observations, given this deterministic skeleton, that this is a form of statistics.

Influences

Formal languages have rules both for syntax and for semantics. For the BUGS language, syntax has been influenced by the S language (Becker et al., 1988) and