WinBUGS Development Interface (WBDev) – Implementing your own univariate distributions

Dave Lunn, Chris Jackson
Imperial College School of Medicine, London, UK
d.lunn@imperial.ac.uk, chris.jackson@imperial.ac.uk

September 1, 2004

1 Introduction

This document explains how you can add new univariate distributions to WinBUGS (1.4) by 'hard-wiring' them into the system via compiled Pascal code. Although the "zeros trick" and the "ones trick", described in the *WinBUGS User Manual*, can both be used to implement new distributions, there are several major advantages to fully integrating them into the system instead. Firstly, built-in distributions can be evaluated much more quickly than distributions defined via the BUGS language. Second, hiding the details of a distribution's likelihood and/or prior contribution within 'hard-wired' components can lead to vastly simplified model code, which reduces the likelihood of coding errors occurring and is straightforward to maintain. Additionally, with *Component Pascal* we have the full flexibility of a general-purpose (modern) computer language to hand for specifying arbitrary distributional properties.

We demonstrate how to implement a new distribution as a 'hard-wired' component via a worked example in which we define a truncated normal distribution, truncated, on the left, at zero. The probability density function (pdf) is given by

$$p(x) = \begin{cases} 0 & x < 0 \\ \frac{\tau^{1/2}}{\sqrt{2\pi}} \exp\left\{-\frac{\tau}{2}(x-\mu)^2\right\} / \left\{1 - \Phi(-\tau^{1/2}\mu)\right\} & x \geq 0 \end{cases} \quad (1)$$

where, if the distribution weren't truncated, μ and τ would represent the mean and precision (inverse-variance), respectively. However, here we refer to μ as the *location* and τ as the *inverse-scale*. Some readers may be under the impression that this distribution could be specified straightforwardly in WinBUGS by applying the `I(.,.)` construct to `dnorm(.,.)`, e.g. `dnorm(mu, tau)I(0, )`. Whilst in some circumstances this may lead to the same results as the distribution given by (1), the `I(.,.)` construct was originally designed only to denote censored observations and shouldn't really be used in an attempt to model truncation. In the former case, the pdf of the associated distribution is unchanged by the specification of lower and upper bounds and those bounds are used simply to ensure that any samples drawn for the relevant, supposedly censored, quantity are within the observed range. However, it is clear from Eqn. (1) that under truncation a distribution's pdf *is* changed: since all pdfs must integrate to 1 over the distribution's support, then the pdf of a truncated distribution is given by that of the untruncated distribution divided by the integral of the untruncated distribution between the truncation points – hence the normalizing constant $1 - \Phi(-\tau^{1/2}\mu)$ in (1) above, where $\Phi(.)$ denotes the cumulative distribution function of the standard normal distribution. A key point here is that the normalizing constant is a function of both μ and τ, and so if μ and/or τ are unknown parameters, then the likelihood contributions to their full conditional distributions that would arise from specifying (1) and `dnorm(mu, tau)I(0, )` separately are fundamentally different.

It is important to note that there is no reason whatsoever why a truncated distribution cannot also be censored, i.e. we can still apply the `I(.,.)` construct to our new distribution. For example, a particular

model may include a quantity with (1) as its distribution that we have observed as being less than a ceratin value, `u`, say; in this case we should append `I(, u)`, or, equivalently, `I(0, u)`, to the distribution specifier.

2 System set-up

Computer code for the truncated normal distribution defined in Equation 1 above can be found in the file `WBDev/Mod/UnivariateTemplate.odc`. This is a *Component Pascal module* that under normal circumstances, i.e. if you had written it yourself, would need to be compiled and then 'linked' into the core WinBUGS software before it could be used from within the software – we have already done this for this example, however, to aid with this exposition (see later). Before discussing the new module in detail we provide instructions on how to set up your system so that Component Pascal code can be compiled.

1. Download *BlackBox Component Builder* from the following web-page:
   `http://www.oberon.ch/blackbox.html`
2. Unzip the downloaded file, if necessary. Install 'BlackBox' by double-clicking on the `Setup.exe` icon and following the instructions. The software should be installed into the new directory `Program Files/BlackBox`.
3. Open `My Computer` (or its equivalent) and navigate to the `Program Files/WinBUGS14` directory; then press `Ctrl+A` (or select `Select All` from the `Edit` menu) to select all files and sub-directories within the `WinBUGS14` directory. Now press `Ctrl+C` (or select `Copy` from the `Edit` menu) to copy those files and sub-directories.
4. Continue using `My Computer` to navigate to the `Program Files/BlackBox` directory and then press `Ctrl+V` (or select `Paste` from the `Edit` menu) to paste the copied files and sub-directories to this location. Select "`Yes to All`" if prompted about replacing existing files.
5. Now your copy of BlackBox should include the *full* functionality of WinBUGS 1.4 (including this `WBDev` interface) within it, and so the `BlackBox.exe` icon on the desk-top or that in the `Program Files/BlackBox` directory can be used either to run WinBUGS in the normal way or to conduct WinBUGS development work (or even more general Component Pascal programming).

3 New module – "WBDevUnivariateTemplate"

Now start your copy of BlackBox and open the new module:

`Program Files/BlackBox/WBDev/Mod/UnivariateTemplate.odc`

Note that to reduce the risk of errors creeping into the system we recommend that all other new components are also stored in the `WBDev/Mod` directory. As the name suggests, the `UnivariateTemplate` module can be used as a template for such new components, so long as they represent univariate distributions (see the file "`WBDev_functions.pdf`" for details on hard-wiring new logical functions, both scalar- and vector-valued, into the system). Please note that only those parts of the code that are currently marked in blue should be modified. The following notes pertain to areas of code labelled with the corresponding numbers within comment markers, i.e. `(*` and `*)`, e.g. `(* this is a comment in Component Pascal *)`.

`(*1*)` The first line of a Component Pascal module should always read `MODULE`, followed by the module's name, in this case `WBDevUnivariateTemplate`, followed by a semi-colon. The last line of the module should read `END`, followed by the module's name, followed by a *full stop* (period). All new

module names for new components of this type should begin with `WBDev`; the corresponding *file* names should be identical but with the `WBDev` prefix removed (they must also begin with at least one capital letter); all new files of this type should be saved in the `WBDev/Mod` directory.

`(*2*)`–`(*3*)` Various other modules can be 'imported' into each new module, which means that procedures and/or data structures defined in those modules can be used/exploited from within the new module. The `Math` module (line `(*3*)`) is an integral part of the BlackBox software since it defines many fundamental mathematical functions, which are called from within other modules via the syntax `Math.` followed by the relevant procedure name, e.g. `Math.Ln(.)` for natural logarithms, `Math.Sqrt(.)` for square roots – see lines `(*27*)`, `(*28*)` and `(*50*)`. Documentation regarding the `Math` module can be accessed by highlighting the word `Math` in BlackBox and selecting `Documentation` from the *second* `Info` menu (the first `Info` menu belongs to WinBUGS).

Similarly, `WBDevSpecfunc` and `WBDevRandnum` are utility modules that provide numerous procedures for defining and sampling from various probability distributions. `WBDevSpecfunc` provides several 'special functions' such as `WBDevSpecfunc.Phi(.)`, the cumulative distribution function of the standard normal distribution $\Phi(.)$ (see lines `(*28*)`, `(*51*)` and `(*52*)`), and `WBDevSpecfunc.LogGammaFunc(.)`, which returns the natural logarithm of the Gamma function, i.e. $\ln\Gamma(x) = \ln\left(\int_0^\infty t^{x-1}e^{-t}\mathrm{d}t\right)$. `WBDevRandnum`, on the other hand, provides many 'black-box' algorithms that can be used for generating pseudo-random numbers – see, for example, lines `(*63*)`, `(*65*)`, `(*67*)` and `(*69*)`. Each of these utility modules is documented separately in the `WBDev/Docu` directory: please see the files `Specfunc.odc` and `Randnum.odc` for full details, or highlight the appropriate module name in BlackBox and select `Documentation` from the second `Info` menu.

`(*4*)` Here we simply define meaningful names with which to reference the arguments of our new distribution – this is by no means essential but *is* considered to be 'good practice' as it reduces the likelihood of coding errors arising. These are constants that have 'global' scope within the module, i.e. they can be referred to from anywhere within the module. For details regarding their usage, please see the notes pertaining to the `LogFullLikelihood(.)`, `LogPropLikelihood(.)` and `LogPrior(.)` procedures below (lines `(*20*)`–`(*42*)`).

`(*5*)` `log2Pi` is a global variable used to hold the value of $\ln(2\pi)$, which is generally useful for specifying normal densities. Global variables are good places to store such 'constants' as they need be evaluated only once, when the module is loaded into memory, although their values may be required a great many times throughout the course of a particular analysis. Because of this fact, there is no need to remove the variable (and/or its definition) when defining distributions for which the value of $\ln(2\pi)$ is irrelevant. Note that the definition of the `log2Pi` variable takes place towards the foot of the new module, within the '`Init`' procedure – any other such variables should be defined in the same place.

`(*6*)`–`(*9*)` As the name suggests, we use the `DeclareArgTypes(.)` procedure to declare the types of arguments required to define the distribution of interest. In the case of our truncated normal distribution defined in Eqn. (1) above, the required arguments are the location parameter μ and the inverse-scale parameter τ. Thus we have two scalar arguments and so we set the `args` variable equal to `"ss"` (`s` denotes a scalar whereas a vector would be denoted by `v`).

`(*10*)`–`(*14*)` The `DeclareProperties(.)` procedure is used to specify two important pieces of information about the new distribution. First, whether the distribution is discrete or continuous; and, second, whether or not we can evaluate its cumulative distribution function, i.e. whether we can integrate the pdf, either numerically or analytically, in which case an algorithm to do so should be provided within the `Cumulative(.)` procedure – see the notes below for lines `(*43*)`–`(*53*)`. The `isDiscrete` variable should be set equal to "`TRUE`" if the distribution is discrete and "`FALSE`" if it is continuous. The `canIntegrate` variable should be set equal to "`TRUE`" if an algorithm to evaluate the cumulative distribution function is to be provided in the `Cumulative(.)` procedure, and "`FALSE`" otherwise.

`(*15*)` – `(*19*)` As the name suggests, the `NaturalBounds(.)` procedure should specify the natural bounds of the distribution, that is, regardless of any censoring that might be applied. (Note that '`INF`' and '`-INF`' can be used to denote $+\infty$ and $-\infty$, respectively, and that bounds for discrete distributions

should be inclusive.) This procedure receives, as an input parameter, a variable named '`node`', as do all other procedures in what follows. This variable represents a stochastic 'node' in the underlying graphical model that is distributed according to the new distribution. It is passed to the `NaturalBounds(.)` procedure because the distribution's bounds may be dependent on some of its arguments. For example, the maximum value that a binomial node can attain is given by the 'number of trials' argument.

`(*20*)`–`(*42*)` The `LogFullLikelihood(.)`, `LogPropLikelihood(.)` and `LogPrior(.)` procedures all return, via the '`value`' variable, the natural logarithm of a number that is proportional to the `node` variable's probability density function evaluated at the node's current value, with the node's arguments (or 'parents') also equal to their current values. The reason for having three procedures that all do essentially the same thing is that WinBUGS doesn't always require the same level of 'exactness'. Sometimes WinBUGS needs the log-pdf specifying *exactly*, i.e. including *all* normalizing constants, in which case the `LogFullLikelihood(.)` procedure is called by the core software. Other times, normalizing constants such as $\ln(2\pi)$ can be ignored, in which case `LogPropLikelihood(.)` is called. Often, however, only those factors of the node's pdf that are functions of the node's value are needed; for example, the $\frac{\tau^{1/2}}{\sqrt{2\pi}}$ and $1 - \Phi(-\tau^{1/2}\mu)$ terms in Eqn. (1) can be omitted, in these circumstances, since they do not involve x. In this latter case, the software calls the `LogPrior(.)` procedure – see lines `(*34*)`–`(*42*)`.

Of course, as there is no harm done in including normalizing constants when they are not actually required, one can always simply call `LogFullLikelihood(.)` from within both `LogPropLikelihood(.)` and `LogPrior(.)` to save coding, as we have done in this worked example for the `LogPropLikelihood(.)` procedure – see line `(*32*)`. However, considerable gains in efficiency can often be made by avoiding unnecessary calculations, especially in cases where normalizing constants are cumbersome to calculate.

Throughout these procedures we refer several times to two of the `node` variable's 'internal fields', namely '`value`' and '`arguments`'. The `value` field is a 'read-only' field that stores the node's current value. The `arguments` field, on the other hand, is an 'irregular' matrix where each row corresponds to one of the arguments declared in `DeclareArgTypes(.)` above (in the same order). If a particular argument is a vector (`v`) then the length of the corresponding row of `node.arguments` is equal to the length of that vector, whereas if an argument is a scalar (`s`) then the length of the corresponding row is 1. The procedure call `node.arguments[i][j].Value()` returns the value of the `j`th element of the `i`th argument. (Note that all array indices start at 0 in Component Pascal rather than 1.) Thus the values of μ and τ are obtained via the calls `node.arguments[location][0].Value()` and `node.arguments[inverseScale][0].Value()`, respectively, where '`location`' (= 0) and '`inverseScale`' (= 1) are the global constants defined on line `(*4*)`.

Note that at the beginning of each procedure, e.g. lines `(*21*)`–`(*22*)`, any number of 'local' variables can be declared for use within the procedure, so long as their names do not clash with other variable/procedure names – the compiler (`Ctrl+K`) will normally inform the programmer of any errors.

`(*43*)`–`(*53*)` In cases where we can evaluate the new distribution's cumulative distribution function, the `Cumulative(.)` procedure should be used to return, via the '`value`' variable, the value of that function at '`x`', where `x` is a real-valued input parameter. The cumulative distribution function of the distribution defined by (1) is

$$\frac{\Phi(\tau^{1/2}(x-\mu)) - \Phi(-\tau^{1/2}\mu)}{1 - \Phi(-\tau^{1/2}\mu)} \tag{2}$$

and so this is specified in our worked example via lines `(*51*)`–`(*52*)`. It is worth noting, however, that the `Cumulative(.)` procedure is only called by the software when it is attempting to calculate the deviance contribution from a censored observation. Hence, if deviance is not of interest and/or censoring is not an issue, then the details of the cumulative distribution function needn't be specified. If this is the case then we *must* specify "`canIntegrate := FALSE;`" in the `DeclareProperties(.)` procedure, and it is advisable to replace any code between the "`BEGIN`"

and "`END Cumulative;`" lines of the `Cumulative(.)` procedure with the "`HALT(126);`"[1] statement that is currently commented out on line `(*47*)`, just to 'catch' any unexpected calls to the procedure.

`(*54*)`–`(*71*)` The `DrawSample(.)` procedure should return, via the '`sample`' variable, a pseudo-random number from the new distribution. This is the only place where we may need to be concerned with the type of censoring applied, if any – this information is passed into the procedure, by WinBUGS, via the '`censoring`' variable, which is integer-valued. On line `(*60*)` we make use of the `node` variable's `Bounds(.)` procedure, which returns, via '`left`' and '`right`', the 'innermost' of the node's natural bounds and any applicable censoring bounds. These may be passed directly into any suitably flexible sampling procedure. However, in this case, to avoid passing possibly infinite bounds into procedures that may not know how to deal with them, we make use of a Component Pascal *CASE* statement to 'branch' on the various different types of censoring possible – lines `(*61*)`–`(*70*)`. In the two cases where the right-hand bound is infinite (but the left-hand bound is finite), i.e. when `censoring = WBDevUnivariate.noCensoring` and when `censoring = WBDevUnivariate.leftCensored`, `WBDevRandnum.NormalLeftTruncated(mu, tau, left)` is used to obtain a single sample from $\mathrm{N}(\mu, \tau^{-1})$ truncated on the left at '`left`'. When both bounds are finite, `WBDevRandnum.NormalTruncated(.)` is an appropriate procedure.

Please note that (almost) every Component Pascal statement ends with a semi-colon. Hopefully this brief example demonstrates sufficient use of the Component Pascal syntax that the reader is able to begin writing their own modules from this template. Detailed documentation on both BlackBox and the Component Pascal language can be accessed via the file:

`Program Files/BlackBox/Docu/Help.odc`

Further insight may also be gained by examining other new distributions in the set of "shared components" that can be downloaded from the WBDev web-site. Please read the instructions below before attempting to write your own modules.

4 Using "WBDevUnivariateTemplate" as a template

The following instructions should be followed closely when defining a new BUGS distribution via the `WBDevUnivariateTemplate` template:

1. Choose a name for the new component, `NewDistribution`, say (the new name *must* begin with a capital letter). Start your copy of BlackBox and open the `WBDev/Mod/UnivariateTemplate.odc` template from within it; then save the template under the new name, e.g. `WBDev/Mod/NewDistribution.odc` – **be careful not to overwrite an existing module!** Now modify the module name both at the top and at the bottom of the new file – change these from `WBDevUnivariateTemplate` to `WBDev` followed by the new component's name, e.g. `WBDevNewDistribution`. Save and compile the new component by pressing `Ctrl+S` (save) followed by `Ctrl+K` (compile) – there should be no compilation errors at this stage since only the module name has been changed.

2. Now modify the code in the new module according to the desired distributional form, i.e. declare the types of arguments required on the line labelled `(*8*)` and redefine the `DeclareProperties(.)`, `NaturalBounds(.)`, `LogFullLikelihood(.)`, `LogPropLikelihood(.)`, `LogPrior(.)`, `Cumulative(.)`, and `DrawSample(.)` procedures. You can save the new module at any time by pressing `Ctrl+S` (or by selecting `Save` from the `File` menu). You can also attempt to compile the code at any time by pressing `Ctrl+K` (or by selecting `Compile` from the `Dev` menu). If there are any compilation errors when you attempt to compile your code, each one will be marked in the code by a grey box with a white cross running through it. An error message pertaining to

[1] The numbers passed to calls of the `HALT(.)` procedure are interpreted, where possible: '126' means 'not implemented yet'.

the first error will be displayed on the status bar (which lies across the bottom of the BlackBox 'program window') and the cursor should automatically position itself next to the corresponding grey box. We advise that you deal with any compilation errors in order, but if for some reason this makes things awkward (or is not possible) then error messages for specific compilation errors can be obtained by clicking on the appropriate grey boxes – a single click shows the error message on the status bar whereas double-clicking reveals it within the code, in place of the grey box (double-click again to revert back to the grey box).

3. Once the new module has been successfully compiled (and saved) then it can be 'linked' into the WinBUGS software by modifying the file `WBDev/Rsrc/Distributions.odc`. The first line of this file contains the required entry for the truncated normal distribution defined in the `WBDevUnivariateTemplate` module:

   ```
   s ~ "dnorm.trunc0"(s, s)I(s, s)              "WBDevUnivariateTemplate.Install"
   ```

   The notation is described as follows: '`s`' means that the component represents a scalar (i.e. it is a univariate distribution) rather than a vector (`v`), whereas '`~`' indicates that the component represents a distribution as opposed to a function (`<-`); '`dnorm.trunc0`' is the name chosen for specifying the new distribution via the BUGS language; '`(s, s)`' tells the system that the distribution requires two scalar arguments as declared in the `DeclareArgTypes(.)` procedure; '`I(s, s)`' confirms that the `I(.,.)` construct, with two scalar arguments, can be applied (as is the case for all univariate distributions defined via the WBDev interface); and '`WBDevUnivariateTemplate.Install`' is the 'installation' procedure for the component – this is defined towards the bottom of the template module. Make a copy of this first line immediately beneath it, and replace `dnorm.trunc0` on the *new* line with the desired BUGS-language name for your new distribution, e.g. `dnew.distribution` (this is the name with which you wish to refer to the new distribution during future WinBUGS sessions). Now declare the new distribution's arguments and specify the name of the module where its 'installation' procedure can be found – replace `WBDevUnivariateTemplate` with your new module's name, e.g. `WBDevNewDistribution`. You should end up with something like

   ```
   s ~ "dnorm.trunc0"(s, s)I(s, s)              "WBDevUnivariateTemplate.Install"
   s ~ "dnew.distribution"(s, s)I(s, s)         "WBDevNewDistribution.Install"
   ```

   at the top of the `WBDev/Rsrc/Distributions.odc` file. Now save the new `WBDev/Rsrc/Distributions.odc` file – the new component will be available from the next time that BlackBox is started, so don't forget to shut down the software before trying to use your new distribution. The new distribution in the illustrative example above could be accessed from within WinBUGS via BUGS syntax similar to the following:

   ```
   model {
   ...
   x ~ dnew.distribution(mu, tau)
   ...
   }
   ```

   Good luck!