OpenBUGS

OpenBUGS Development Interface (UDev): Implementing your own functions

Dave Lunn, Chris Jackson & Steve Miller,
MRC Biostatistics Unit,
Institute of Public Health,
University Forvie Site,
Robinson Way,
Cambridge,
CB2 0SR,
`david.lunn@mrc-bsu.cam.ac.uk`
`chris.jackson@mrc-bsu.cam.ac.uk`
`steve.miller@mrc-bsu.cam.ac.uk`

OpenBUGS
`http://sourceforge.net/projects/openbugs/`

July 23, 2010

Contents

1 Introduction

This document explains how you can implement arbitrarily complex logical functions in OpenBUGS by 'hard-wiring' them into the system via compiled Component Pascal code. The facility to implement new distributions will be included in a future release.

There are three main advantages to doing this: first, 'hard-wired' functions can be evaluated much more quickly than their BUGS-language equivalents; second, the full flexibility of a general-purpose computer language is available for specifying the desired function, and so piecewise functions, for example, can be specified straightforwardly whereas their specification via the BUGS language (using the `step(.)` function) can be somewhat awkward; finally, the practice of hiding the details of complex logical functions within 'hard-wired' components can lead to vastly simplified OpenBUGS code for the required statistical model, which reduces the likelihood of coding errors and is easier both to read and to modify.

We demonstrate how to implement such 'hard-wired' components via a worked example in which the following function becomes a single element of the BUGS language.

$$C(t) = \begin{cases} 0 & t < 0 \\ \frac{D}{V}\frac{k_a}{k_a - k_e}\left[\exp(-k_e t) - \exp(-k_a t)\right] & t \geq 0 \end{cases} \tag{1}$$

This is known in the field of pharmacokinetics as a "one compartment open model with first-order absorption". Here $C(t)$ denotes the concentration, at time t, of drug in blood plasma following (oral) administration of a dose D; the system parameters V, k_e and k_a denote the drug's volume of distribution, elimination rate constant, and (first-order) absorption rate constant, respectively. For reasons of identifiability we parameterise the model in terms of $\theta = \log(V, k_e, k_a - k_e)$ so every possible combination of real values (positive or negative) for the elements of θ gives rise to physically feasible values for V, k_e and k_a and generates a *distinct* concentration-time profile.

2 System set-up

Before doing anything, you will need to install some software.

Please read the document "Use of BlackBox with OpenBUGS", and follow the instructions.

3 New module – "UDevScalarTemplate"

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

`OpenBUGS/UDev/Mod/ScalarTemplate.odc`

assuming that OpenBUGS has been installed to the folder `OpenBUGS`.

Note that to reduce the risk of errors creeping into the system we recommend that all other new components are also stored in the `UDev/Mod` directory. As the name suggests, the `ScalarTemplate` module can be used as a template for such new components, so long as they represent scalar-valued functions (see later for details regarding vector-valued functions). 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 `(*` and `*)`:

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

(*2*) 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 one. The `Math` module is an integral part of the BlackBox software that 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.Exp(.)` for exponentials. see the `Evaluate` procedure of this module for examples of their use.

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 OpenBUGS).

(*3*) The `Signature(.)` procedure is used to declare the types of arguments required to define the function of interest. In the case of our one compartment pharmacokinetic model defined in Eq. 1 above, the required arguments are: the parameter vector θ, the dose D, and the time t. Thus we have a vector followed by two scalars and so we set the `signature` variable equal to `"vss"` (`v` for vector; `s` for scalar).

(*4*) The `Evaluate` procedure is used to define a variable called `value`, which stores the function's current value (given the current values of its arguments). Throughout the procedure we make use of a variable called `func`, which represents the function itself. In particular, we refer several times to one of its 'internal' fields, `arguments`. This is an 'irregular' matrix where each row corresponds to one of the arguments declared in `Signature(.)` above (in the same order). If a particular argument is a vector (`v`) then the length of the corresponding row of `func.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 `func.arguments[i][j].Value()` returns the value of the `j`th element of the `i`th argument. Thus the value of $\log k_e$, for example, can be obtained via the call `func.arguments[0][1].Value()` – because $\log k_e$ is the second element (index = 1) of the θ vector, which is the function's first argument (index = 0).

Note: Array indices start at 0 in Component Pascal rather than 1. Thus if the array has N elements, the array index will have values $i = 0, ..., N-1$.

(*5*) Here we define three constants that allow us to index the various rows of `func.arguments` via meaningful names rather than directly by the relevant numbers themselves, i.e. we can use the names `parameters`, `dose` and `time` in place of `0`, `1` and `2` to access the function's first, second and third arguments, respectively. This is by no means essential but *is* considered to be 'good practice' as it reduces the likelihood of coding errors arising.

(*6*) Note that any number of 'local' variables can be declared and used to aid in specifying the new function, 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.

(*7*) This is a standard "IF/THEN/ELSE" statement in Component Pascal; note that the "ELSE" branch can be omitted where appropriate – see below.

(*8*) This is a standard "IF" statement in Component Pascal – here we simply set equal to zero any negligibly small values that have been calculated as negative due to machine precision errors.

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 `Help` menu.

Further insight may also be gained by examining our second template, which shows how to implement new components to represent vector-valued functions – see later. Please read the instructions below before attempting to write your own modules. In addition, there are several further examples outlined towards the end of this document.

4 Using "UDevScalarTemplate" as a template

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

1. Choose a name for the new component, `NewFunction`, say (the new name *must* begin with a capital letter). Start your copy of BlackBox and open the `UDev/Mod/ScalarTemplate.odc` template from within it; then save the template under the new name, for example

   ```
   UDev/Mod/NewFunction.odc
   ```

   Do not overwrite an existing module! Now modify the module name both at the top and at the bottom of the new file – change these from `UDevScalarTemplate` to `UDev` followed by the new component's name, e.g. `UDevNewFunction`. 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 function:

   (a) declare the types of arguments required in the `Signature()` method at the line labelled `(*3*)`, and

   (b) redefine the `Evaluate(.)` procedure starting at line `(*4*)`.

   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 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 OpenBUGS software by modifying the file `UDev/Mod/External.odc`. This file contains `Register` commands for every new function to be added to the BUGS language.

   For the `ScalarTemplate` example, the command is:

   ```
   Register("udev.scalartemplate", "UDevScalarTemplate.Install");
   ```

   where the first string is the name to be used in the BUGS language, and the second is the procedure in the module which BUGS uses to access the function.

   In the case of the new function in the module `UDevNewFunction` the command would be:

   ```
   Register("my.new.function", "UDevNewFunction.Install");
   ```

   In order to use the new function, the file `External.odc` must be compiled (Control-K), and you must exit from BlackBox. The function will be available when BlackBox is started again.

   In this example the new function could be accessed in a model via BUGS language code of the following form:

   ```
   model {
     #  ...
     value <- my.new.function(x, par[1:p])
     #  ...
   }
   ```

5 Vector-valued functions

Vector-valued functions can be 'hard-wired' into the system by making use of a different template, which can be found in `UDev/Mod/VectorTemplate.odc`. Here we define a version of our one compartment pharmacokinetic model that is now vector-valued by virtue of the fact that we now wish to evaluate it at a vector of times rather than a single time. Except for a few minor points discussed below, the details of implementation are exactly analogous to those for scalar-valued functions.

(*1*) Module name.

(*2*) The `args` variable is now set equal to `"vsv"` rather than `"vss"` since the function's third argument has changed from a single time value to a vector of time values.

(*3*) The `Evaluate(.)` procedure must now return *an array* of values, via the '`values`' variable, rather than a scalar (previously via '`value`').

(*4*) Component Pascal's `LEN(.)` function returns the length of the specified argument, so long as that argument is a one-dimensional array: thus `LEN(func.arguments[times])` is the number of times at which the function is to be evaluated. If the array has more than one dimension then `LEN(.)` returns the length of the *first* dimension – see the BlackBox documentation for more details.

(*5*)–(*7*) This section forms the basic structure of a Component Pascal `WHILE`-loop. At line `(*6*)` the command `INC(i)` increments the value of `i` by one. During each 'pass' through the loop, the `t` variable is set equal to one of the times at which the function is to be evaluated and the corresponding evaluation is performed by making use of the 'temporary' variable `val`. At the end of each pass, `values[i]` ($\mathtt{i} = 0, ..., \mathtt{numTimes} - 1$) is set equal to `val`.

Finally, the function must be associated with a string to enable it to be called in the BUGS language, to be put into the file `External.odc` using the statement:

```
Register("udev.vectortemplate", "UDevVectorTemplate.Install");
```

Appendices

These appendices contain brief descriptions of some functions created using the templates described above, and are included in the OpenBUGS distribution. Modules are included for all of them and so may be used for reference.

A DJLPKIVbol2, DJL2CompDisp

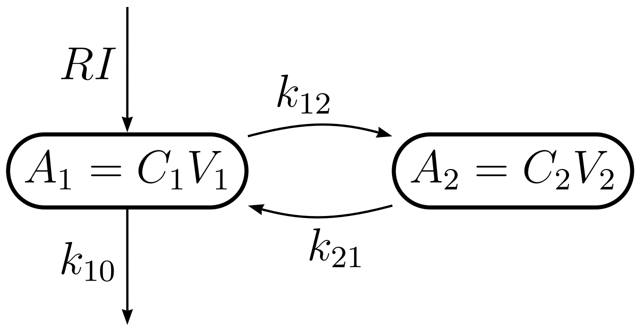

Figure 1: Two compartment pharmacokinetic model with input into (RI) and elimination from (k_{10}) the central compartment only. Compartmental volumes, concentrations, and amounts of drug are denoted by V_i, C_i and A_i, respectively.

Source code in the files:

```
UDev/Mod/DJLPKIVbol2.odc
UDev/Mod/DJL2CompDisp.odc
```

Consider the two compartment pharmacokinetic model shown in Fig. 1. This can be expressed mathematically as

$$\frac{\mathrm{d}A_1}{\mathrm{d}t} = RI + k_{21}A_2 - k_{12}A_1 - k_{10}A_1 \qquad (2)$$

$$\frac{\mathrm{d}A_2}{\mathrm{d}t} = k_{12}A_1 - k_{21}A_2 \qquad (3)$$

For the case in which the input function (RI) represents an intravenous bolus, i.e. where all of the dose is injected into venous blood as quickly as possible: $RI = 0, A_1(0_+) = D$ (where D denotes the administered dose), the concentration of drug in the central compartment at time t is given by

$$C_1(t) = \frac{D}{V_1}\left\{A\mathrm{e}^{-\lambda_1 t} + (1 - A)\mathrm{e}^{-\lambda_2 t}\right\} \qquad (4)$$

where (after some algebra):

$$\lambda_1 = \frac{1}{2}\left\{k_{12} + k_{21} + k_{10} + \sqrt{(k_{12} + k_{21} + k_{10})^2 - 4k_{10}k_{21}}\right\};$$
$$\lambda_2 = k_{12} + k_{21} + k_{10} - \lambda_1;$$
$$\text{and } A = \frac{\lambda_1 - k_{21}}{\lambda_1 - \lambda_2}.$$

For the purposes of Bayesian inference, arguably the most appropriate parameterisation for this model is in terms of the following 'disposition' parameters:

$$\theta = \log(CL, Q, V_1, V_2)$$

where $CL = k_{10}V_1$ and $Q = k_{12}V_1 = k_{21}V_2$. This is because each possible combination of real values (positive or negative) for the elements of θ is then not only physically feasible but also gives rise to a *distinct* concentration-time profile. Thus an assumption of multivariate normality for θ is appropriate.

The `UDevDJLPKIVbol2` component takes values of t, D and $\psi = (\lambda_1, \lambda_2, A, V_1)$ as arguments (in the order ψ, t, D) and returns the corresponding value of C_1. An example of its usage is as follows:

```
conc <- udev.djlivbol2(psi[1:4], time, dose)
```

The component is designed to be used in conjunction with the `DJL2CompDisp` component, although this is not essential.

The `UDevDJL2CompDisp` component maps the current value of θ to

$$\psi = (\lambda_1, \lambda_2, A, V_1),$$

which is useful in the evaluation of *all* standard two compartment models. This component is then used in OpenBUGS as:

```
psi[1:4] <- udev.djl2compdisp(theta[1:4])
```

B DJLPKIVbol3, DJL3CompDisp

Source code in the files:

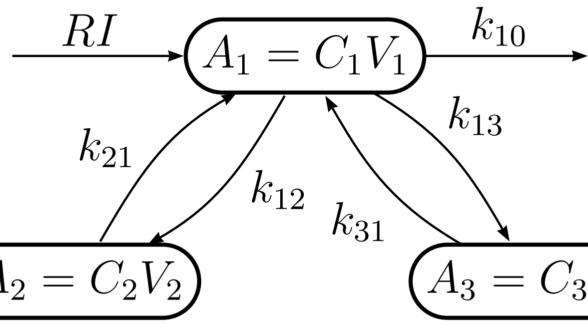

Figure 2: Three compartment pharmacokinetic model with input into (RI) and elimination from (k_{10}) the central compartment only. Compartmental volumes, concentrations, and amounts of drug are denoted by V_i, C_i and A_i, respectively.

```
UDev/Mod/DJLPKIVbol3.odc
UDev/Mod/DJL3CompDisp.odc
```

Consider the three compartment pharmacokinetic model shown in Fig. 2 above. This can be expressed mathematically via:

$$\frac{\mathrm{d}A_1}{\mathrm{d}t} = RI + k_{21}A_2 + k_{31}A_3 - k_{12}A_1 - k_{13}A_1 - k_{10}A_1 \tag{5}$$

$$\frac{\mathrm{d}A_2}{\mathrm{d}t} = k_{12}A_1 - k_{21}A_2 \tag{6}$$

$$\frac{\mathrm{d}A_3}{\mathrm{d}t} = k_{13}A_1 - k_{31}A_3 \tag{7}$$

For the case in which the input function (RI) represents an intravenous bolus, i.e. where all of the dose is injected into venous blood as quickly as possible: $RI = 0, A_1(0_+) = D$ (where D denotes the administered dose), the concentration of drug in compartment 1 at time t is given by

$$C_1(t) = \frac{D}{V_1}\left\{A\mathrm{e}^{-\lambda_1 t} + B\mathrm{e}^{-\lambda_2 t} + (1 - A - B)\mathrm{e}^{-\lambda_3 t}\right\} \tag{8}$$

where (after considerable algebra):

$$A = (k_{31} - \lambda_1)(k_{21} - \lambda_1)/(\lambda_2 - \lambda_1)(\lambda_3 - \lambda_1), \tag{9}$$

$$B = (k_{31} - \lambda_2)(k_{21} - \lambda_2)/(\lambda_1 - \lambda_2)(\lambda_3 - \lambda_2) \tag{10}$$

$$\lambda_1 = \max(\lambda_1', \lambda_2', \lambda_3'), \quad \lambda_2 = \mathrm{median}(\lambda_1', \lambda_2', \lambda_3') \quad \lambda_3 = \min(\lambda_1', \lambda_2', \lambda_3') \quad (11)$$

$$\lambda_i' = \frac{\alpha}{3} - 2\sqrt{-\frac{a}{3}} \times \cos\left\{(\phi + 2(i-1)\pi)/3\right\}, \quad i = 1, 2, 3 \quad (12)$$

$$\phi = \cos^{-1}\left(-b/2\sqrt{-a^3/27}\right), \quad a = \beta - \frac{1}{3}\alpha^2, \quad b = \frac{2}{27}\alpha^3 - \frac{1}{3}\alpha\beta + \gamma \quad (13)$$

$$\alpha = k_{13}+k_{12}+k_{10}+k_{31}+k_{21}, \quad \beta = k_{31}(k_{12}+k_{10}+k_{21})+k_{21}(k_{13}+k_{10}), \quad (14)$$

and $\gamma = k_{21}k_{31}k_{10}$. The `DevDJLPKIVbol3` component takes as arguments values of and $\psi = (\lambda_1, \lambda_2, \lambda_3, A, B, V_1)$, t and D (in that order) and returns the corresponding value of $C_1(t)$. An example of its usage in BUGS language code would be

```
conc <- udev.djlivbol3(psi[1:6], time, dose)
```

For the purposes of Bayesian inference, arguably the most appropriate parameterisation for this model is in terms of the following 'disposition' parameters:

$$\theta = \log(CL, Q_{12}, Q_{13}, V_1, V_2, V_3 - V_2)$$

where $CL = k_{10}V_1$, $Q_{12} = k_{12}V_1 = k_{21}V_2$, and $Q_{13} = k_{13}V_1 = k_{31}V_3$. This is because each possible combination of real values (positive or negative) for the elements of θ is then not only physically feasible but also gives rise to a *distinct* concentration-time profile. Thus an assumption of multivariate normality for θ is appropriate.

The `UDevDJL3CompDisp` component maps the current value of θ to

$$\psi = (\lambda_1, \lambda_2, \lambda_3, A, B, V_1),$$

which is useful in the evaluation of *all* standard three compartment models. An example of the new component's usage (from within OpenBUGS) is as follows:

```
psi[1:6] <- udev.djl3compdisp(theta[1:6])
```

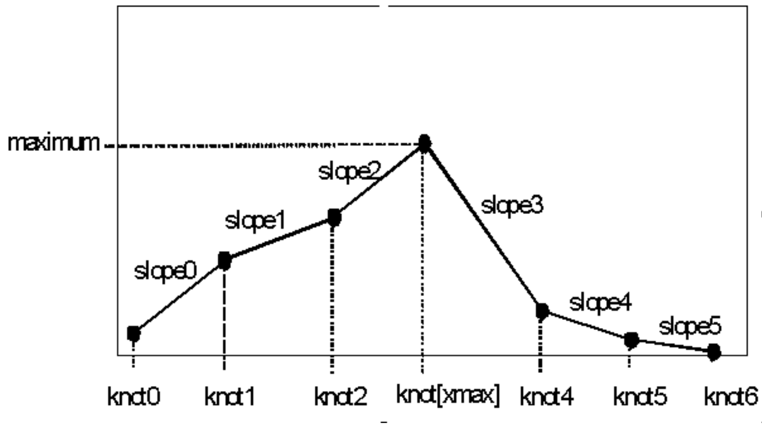

Figure 3: Figure for ELP

C Elicitor Piecewise Function

Source code in the file:

```
UDev/Mod/ElicitorPiecewise.odc
```

Mary Kynn, Department of Mathematics and Statistics, Fylde College, Lancaster University, (m.kynn@lancaster.ac.uk), February 13, 2004

Consider the piecewise continuous linear function shown in figure Figure 3). It is anchored at some maximum value and then constructed outwards from the maximum, where each section is given a gradient and begin/end points (knots). For example

$$f(x) = \begin{cases} s_0(x-40) + s_1(40-50) + s_2(50-60), & 0 < x \leq 40.0 \\ s_1(x-50) + s_2(50-60), & 40 < x \leq 50 \\ s_2(x-60), & 50 < x \leq 60 \\ s_3(x-60), & 60 < x \leq 75 \\ s_4(x-75) + s_3(75-60), & 75 < x \leq 81 \\ s_5(x-81) + s_4(81-75) + s_3(75-60), & 81 < x \leq 100 \end{cases} \tag{15}$$

The piecewise function can be called from OpenBUGS by the command

```
variable <- udev.elicitorpiecewise(x, numberKnots,
                                   xmaxKnot, slopes[], knots[])
```

So for the example described above

```
numberKnots = 7
xmaxKnot = 3
slopes = [s0, s1, s2, s3, s4, s5]
knots = [0, 40, 50, 60, 75, 81, 100]
```

The function tests to see in which piece the argument `x` lies and returns the corresponding value.

D SparseMatrixVectorProduct

Juan J. Abellán, Department of Epidemiology and Public Health, Imperial College London, (j.abellan@imperial.ac.uk), 5 October 2004.

Source code in the file:

```
UDev/Mod/SparseMatrixVectorProduct.odc
```

The module `UDevSparseMatrixVectorProduct` performs the multiplication Av of a $n \times m$ sparse matrix A and a vector $v \in \mathbb{R}^m$. One of the many possible representations is the compressed sparse row format. Using this method, the matrix A is stored in the following way:

1. `ra`: a real array of s elements containing the non-zero elements of A, stored in row order. Thus, if $i < j$, all elements of row i precede elements from row j. The order of elements within the rows is immaterial.

2. `ja`: an integer array of s elements containing the column indices of the elements stored in `ra`.

3. `ia`: an integer array of $n+1$ elements containing pointers to the beginning of each row in the arrays `ra` and `ja`. Thus `ia[i]` indicates the position in the arrays `ra` and `ja` where the ith row begins. The last $(n+1)$-th element of `ia` indicates where the $n+1$ row would start, if it existed.

This module is associated with the function name `udev.sparsemat`, and its arguments are the three vectors `ra`, `ja`, and `ia` characterizing A, and the vector v. It should be called in the following way:

```
u[1:m] <- udev.sparsemat(ra[], ja[], ia[], v[])
```

Note the dimension `1:m` within the square brackets in the left-hand side of the assignment.