OpenBUGS Differential Equation Solver

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

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

February 10, 2014

Note 1: Whilst we have endeavoured to ensure that our differential equation solving algorithms are robust, there is no guarantee that a given system of equations can be solved accurately for all possible sets of system parameters. Hence, it is sometimes *essential* to specify informative priors.

Note 2: This document assumes familiarity with the basic concepts of modelling using WinBUGS/OpenBUGS and the BUGS language.

Note 3: This document should be read in conjunction with "Use of BlackBox with OpenBUGS"

Contents

Part I

Basic examples

1 First example; exponential decay

This section will introduce the capabilities of OpenBUGS in dealing with ordinary differential equation models by using the simple model

$$\frac{dx}{dt} = -\lambda x, \quad \lambda > 0. \tag{1}$$

The inference problem we are presented with is a series of noisy measurements $\{y_i\}$ made at corresponding time points $\{t_i\}$, $i = 1, \ldots, N$, and we wish to estimate the value of λ. Suppose that we assume that the measurements are normally distributed about the exact values

$$y_i \sim N(x_i, \tau^{-1})$$

with precision τ. The exact solution to Eq.(1) is $x(t) = x(t_0)\exp(-\lambda t)$ where $x(t_0)$ is the initial value.

OpenBUGS is capable of simulating data from a model with values of parameters specified, and (more importantly) performing inference on a model when parameters are unknown. In this document we will be doing both; firstly simulating data with known parameter values and then doing an inference calculation on the data thus generated. This approach helps to confirm that calculations are being performed correctly since the inference should recover the parameter values that were used to generate simulated data.

1.1 BUGS language version

To simulate the observations described above, the following OpenBUGS code is used:

```
model
{
  solution[1:ngrid, 1:ndim] <- ode(init[1:ndim],
                                   tgrid[1:ngrid],
```

```
                                    D(C[1:ndim], t),
                                    origin,
                                    tol)
  D(C[1], t) <- -lambda * C[1]

  for (i in 1:ngrid)
  {
    obs[i] ~ dnorm(solution[i, 1], tau)
  }
}
```

The basic features of this model are as follows:

1. `ndim` is the number of differential equations in the system; unity for this example.
2. `ngrid` is the number of time points N
3. `solution[] <- ode()`. The `ode()` function gives the solution of the differential equation system. Its arguments are initial values (`init`), the time grid (`tgrid`), the right-hand sides of the differential equations (`D`), the time origin (`origin`) at which the initial values are specified, and a numerical tolerance (`tol`).
4. `D(C[1], t) <- - lambda * C[1]` is the right hand side of the differential equation. Later examples show models with more than one differential equation.
5. The values of the time origin (`origin`), initial conditions for the ODE (`init`), the tolerance (`tol`) and the precision (`tau`) are constants, which are specified in the model data.

When the simulation has been run for a certain number of MCMC iterations (e.g. 100), we can save the observations generated by the run. Choose the option `Save state` from the `Model` menu. This gives the current values of the observations `obs` which can then be used as data for the inference computation.

The model, data, initial values (for the MCMC computation), and simulated observations are in the file

```
Diff/Examples/Example01.odc
```

The inference version of the model is as follows:

```
model
{
  solution[1:ngrid, 1:ndim] <- ode(init[1:ndim],
                                   tgrid[1:ngrid],
                                   D(C[1:ndim], t),
                                   origin,
                                   tol)
  D(C[1], t) <- -lambda * C[1]
  log.lambda ~ dnorm(0.0, tau.lambda);
  lambda <- exp(log.lambda)

  for (i in 1:ngrid)
  {
    obs[i] ~ dnorm(solution[i, 1], tau)
  }
  tau ~ dgamma(a, b)
}
```

This differs from the simulation model in that prior distributions are specified for the parameter `lambda` and the precision `tau`. The parameter `lambda` must be non-negative, and so is modelled as log-normal. The observations generated from the simulation model are given in the data, and MCMC initial values are specified for `log.lambda` and `tau`. The data and initial values are given in the same file as the model.

We now run OpenBUGS on this model for 8000 iterations in total. The node statistics are obtained by selecting iterations 4001 to 8000 (since the updating phase of the BUGS Metropolis algorithm is 4000 iterations by default):

	mean	sd	MC error	val2.5pc	median	val97.5pc	start	sample
lambda	1.171	0.2815	0.009479	0.7421	1.135	1.791	4001	4000
tau	14.76	3.004	0.03991	9.446	14.55	21.11	4001	4000

The exact values used in the simulation run were `lambda` = 1 and `tau` = 10.0.

1.2 Compiled component version

A second way of placing this model into the BUGS framework is to use a compiled module written in the language in which BUGS itself is written (Component Pascal).

When the code is written following the form of the template that we describe later on in this document, one merely has to specify the number of differential equations and a function to compute their right hand sides. One advantage of this approach is that it is much faster to execute. In addition, the specification of the system of differential equations is removed from the BUGS language code, making it considerably more readable.

The code must be placed in what is referred to as a module; the file for this example is in

```
Diff/Mod/Exponential.odc
```

The complete instructions for writing your own modules are in a later section of this document, but for the moment the two most important points are (highlighted in blue)

1. The number of equations in the system is specified near the beginning of the file

   ```
   CONST
     nEq = 1;
   ```

2. The procedure (or function) which computes the right hand side of the differential equation is called `Derivatives()`. The arguments are the array of parameter values (`parameters`), the array of current values of the solution (`C`), the value of the time variable (`t`) and the derivatives (`dCdt`). The computation for this example is fairly simple:

   ```
   lambda := parameters[0];
   dCdt[0] := - lambda * C[0];
   ```

   It is also worth noting at this point that array indices in Component Pascal go from 0 to $N-1$ where N is the length of the array. In this example the arrays `parameters` and `C` have one element only, which is accessed using the index `0`. A more complex example where these arrays have more than one element is discussed later in this document.

Now for this example, the model in the BUGS language is

```
model
{
  solution[1:ngrid, 1:ndim] <- diff.exponential(init[1:ndim],
                                        tgrid[1:ngrid],
                                        parameters[1:nparam],
                                        origin,
                                        tol)
  log.lambda ~ dnorm(0.0, tau.lambda)
  parameters[1] <- exp(log.lambda)

  for (i in 1:ngrid)
  {
    obs[i] ~ dnorm(solution[i, 1], tau)
  }

  tau ~ dgamma(a, b)
}
```

and is entirely analogous with the previous case. Differences are

1. The new module is put into the BUGS language and associated with the function name `diff.exponential`
2. You do not have to specify the right hand sides of the differential equations (because they are in the new module)
3. The parameter `lambda` has to be passed to the new function in an array which has been called `parameters` in this example. Again, the parameter `lambda` has been made log-normal.
4. There is a quantity called `nparam` in the data for the model specifying the size of the `parameters` array

The model, data, MCMC initial values, and observations (from the simulation of the previous section) are the file

```
Diff/Examples/Example02.odc
```

2 A more complex example; Lotka-Volterra

In this section we will take a look at the Lotka-Volterra system of equations

$$\frac{dx}{dt} = x(\alpha - \beta y), \qquad \frac{dy}{dt} = -y(\gamma - \delta x)$$

where $\alpha, \beta, \gamma, \delta$ are positive real quantities. For a simulation run we would specify the values of $\alpha, \beta, \gamma, \delta$ as constants, and for an inference run attempt to estimate their values from observations of x and y. Note that systems of differential equations do not require all variables to be observed to perform an inference computation; in more realistic cases we might only observe one variable of a system of many equations.

The main point of interest of this model compared to the example of the previous section is that we now have two differential equations rather than one. The following two sections show how to generate simulated observations of the system and how to use those observations for inference

2.1 Simulation

The observations are simulated as normally distributed about the exact values of the solution with some specified precision; the ODE initial conditions are specified, as are values of the parameters `alpha`, `beta`, `gamma`, `delta`, and the precisions `tau.x` and `tau.y`.

```
model
{
  solution[1:ngrid, 1:ndim] <- ode(init[1:ndim],
                                   tgrid[1:ngrid],
                                   D(C[1:ndim], t),
                                   origin,
                                   tol)
    D(C[1], t) <-  C[1]*(alpha - beta*C[2])
    D(C[2], t) <- -C[2]*(gamma - delta*C[1])
    for (i in 1:ngrid)
    {
      obs_x[i] ~ dnorm(solution[i, 1], tau.x)
      obs_y[i] ~ dnorm(solution[i, 2], tau.y)
    }
}
```

and the data for the simulation are

```
list(
  ndim = 2,
  origin = 0,
  tol = 1.0E-3,
  ngrid = 51,
  init = c(0.1, 0.1),
  tgrid = c(0.00000, ... ),
  alpha = 1.0,
  beta = 1.0,
  gamma = 1.0,
  delta = 1.0,
  tau.x = 10,
  tau.y = 10
  )
```

where the contents of the array `tgrid` have been omitted. The numerical values of the observations `obs_x` and `obs_y` can be obtained again by using the option `Model|Save State`. The complete simulation model and data are in the file

```
Diff/Examples/Example03.odc
```

2.2 Inference

The inference model must specify priors for the parameters $\alpha, \beta, \gamma, \delta$ and associated precisions. Since $\alpha, \beta, \gamma, \delta$ must be non-negative the prior chosen for them is log-normal. The right hand sides of the differential equations and the observations are as before.

```
model
{
  solution[1:ngrid, 1:ndim] <- ode(init[1:ndim],
                                   tgrid[1:ngrid],
                                   D(C[1:ndim], t),
                                   origin,
                                   tol)
```

```
    alpha <- exp(log.alpha)
    beta <- exp(log.beta)
    gamma <- exp(log.gamma)
    delta <- exp(log.delta)
    log.alpha ~ dnorm(0.0, 0.0001)
    log.beta ~ dnorm(0.0, 0.0001)
    log.gamma ~ dnorm(0.0, 0.0001)
    log.delta ~ dnorm(0.0, 0.0001)

    D(C[1], t) <-  C[1]*(alpha - beta*C[2])
    D(C[2], t) <- -C[2]*(gamma - delta*C[1])
    for (i in 1:ngrid)
    {
      obs_x[i] ~ dnorm(solution[i, 1], tau.x)
      obs_y[i] ~ dnorm(solution[i, 2], tau.y)
    }
    tau.x ~ dgamma(a, b)
    tau.y ~ dgamma(a, b)
}
```

The model, data and MCMC initial values are also in the file

```
Diff/Examples/Example03.odc
```

The model is run for 8000 iterations, and the first 4000 iterations excluded from computation of the node statistics, to give the following results:

	mean	sd	MC error	val2.5pc	median	val97.5pc	start	sample
alpha	0.9966	0.008229	5.824E-4	0.9805	0.9966	1.013	4001	4000
beta	1.0	0.02211	0.001634	0.9596	0.9995	1.045	4001	4000
delta	1.042	0.04157	0.003245	0.9639	1.042	1.127	4001	4000
gamma	1.038	0.05362	0.004295	0.9384	1.037	1.145	4001	4000

where the values of the parameters $\alpha, \beta, \gamma, \delta$ were all set to unity in the simulation. They have been recovered very accurately by the inference computation.

2.3 Compiled component version

A compiled component version of this model can be created to do the same computations. The models, data and MCMC initial values for both simulation and inference models are supplied in the file

```
Diff/Examples/Example04.odc
```

Similarly to the compiled component version of the exponential example, the right hand sides of the differential equations do not have to be specified in the BUGS code. The parameters of the problem (α, β, γ and δ) are passed though an array called `parameters`.

The compiled component module itself is in the file

```
Diff/Mod/LotkaVolterra.odc
```

and again, the most important parts of this are (shown in blue)

1. The number of equations is set to be `nEq := 2`
2. The procedure (or function) which computes the right hand sides of the ODE system is `Derivatives()`. The arguments are the arrays of parameters and current values of the solution (`parameters` and `C` respectively), the time variable `t`, and the output array of derivatives `dCdt`.

As with the exponential model of the previous section, the new module has to be associated with a command in the BUGS language so that it can be used. This is chosen to be `diff.lotvol` and is again placed in the file

```
Diff/Mod/External.odc
```

One important point that should be noted here is how array indices are dealt with. Consider the body of the `Derivatives()` procedure:

```
alpha := parameters[0];
beta := parameters[1];
gamma := parameters[2];
delta := parameters[3];
dCdt[0] := C[0] * (alpha - beta * C[1]);
dCdt[1] := - C[1] * (gamma - delta * C[0])
```

Firstly, the quantities `alpha`, `beta`, `gamma` and `delta` are set from the array `parameters`. The order of the parameters in this array is the same as the order in which they are given in the array in the BUGS language. Then we use these to compute the two values of the derivatives `dCdt[0]` and `dCdt[1]` which correspond to $\frac{dx}{dt}$ and $\frac{dy}{dt}$ respectively.

Array indices in Component Pascal go from 0 to $N-1$ where N is the length of the array; thus when we have four quantities in the array `parameters`, the corresponding array indices are $0, 1, 2, 3$ for `alpha`, `beta`, `gamma`, `delta` respectively. This is the same convention as used in C/C++.

Similarly, when we access the two elements of the array `C`, the indices 0 and 1 are used. The same applies to the array `dCdt` which also has two elements.

2.3.1 Initial values

In all previous examples, the initial conditions of the differential equations have been constants and specified as data. It is possible to estimate these quantities as well, along with the other parameters of a model. So if we take the Lotka-Volterra system, the initial values have to be non-negative so will be modelled as log-normal. Thus the model must be modified by the addition of priors for these quantities.

To implement this in the inference model using the compiled component, we firstly remove the `init` array from the model data, and then add the following:

```
log.init[1] ~ dnorm(0.0, 0.0001)
log.init[2] ~ dnorm(0.0, 0.0001)
init[1] <- exp(log.init[1])
init[2] <- exp(log.init[2])
```

Then add initial values for the array `init` to the inference MCMC initial values. The model, data, and MCMC initial values for this example are given in the file

```
Diff\Examples\Example05.odc
```

Part II

Advanced applications

3 Five compartment pharmacokinetic model

This example makes use of a physiologically based pharmacokinetic (PBPK) model for a hepatically cleared drug given orally to some hypothetical animal. The purpose of this model is to demonstrate the OpenBUGS Differential Equation Interface – all of the physiological parameter values specified are arbitrary. The model is depicted in Figure 1 and can be described mathematically via:

$$\frac{\mathrm{d}C_{PP}}{\mathrm{d}t} = K_{I_{PP}}C_{ART} - K_{O_{PP}}C_{PP}$$

$$\frac{\mathrm{d}C_{RP}}{\mathrm{d}t} = K_{I_{RP}}C_{ART} - K_{O_{RP}}C_{RP}$$

$$\frac{\mathrm{d}C_{GU}}{\mathrm{d}t} = K_{I_{GU}}C_{ART} - K_{O_{GU}}C_{GU}$$

$$\frac{\mathrm{d}C_{LI}}{\mathrm{d}t} = \left(Q_H C_{ART} + K_{O_{GU}}V_{GU}C_{GU} + RA - RM\right)/V_{LI} - K_{O_{LI}}C_{LI}$$

$$\frac{\mathrm{d}C_{LU}}{\mathrm{d}t} = K_{I_{LU}}C_{VEN} - K_{O_{LU}}C_{LU}$$

$$\begin{aligned}\frac{\mathrm{d}C_{VEN}}{\mathrm{d}t} &= (K_{O_{PP}}V_{PP}C_{PP} + K_{O_{RP}}V_{RP}C_{RP} + K_{O_{LI}}V_{LI}C_{LI} \\ &\quad - K_{I_{LU}}V_{LU}C_{VEN})/V_{VEN}\end{aligned}$$

$$\frac{\mathrm{d}C_{ART}}{\mathrm{d}t} = (K_{O_{LU}}V_{LU}C_{LU} - Q_{TOT}C_{ART})/V_{ART}$$

Here $C_{\cdot}$ denotes the concentration of drug within the indicated compartment, $K_{I_T} = Q_T/V_T$ and $K_{O_T} = Q_T/V_T K_{P_T}$, where Q_T, V_T and K_{P_T} denote the rate of blood flow, the volume, and the tissue-to-blood partition coefficient associated with tissue/compartment T, respectively. In addition, Q_H represents the liver's *arterial* blood supply ($Q_H = Q_{LI} - Q_{GU}$; see Fig. 1) and Q_{TOT} is the total cardiac output, which is equal to Q_{LU} as all blood must flow through the lungs to be oxygenated. (Note that for mass balance we must also have that $Q_{TOT} = Q_{LU} = Q_{PP} + Q_{RP} + Q_{GU} + Q_H$.) Also, for this particular example, the rate of absorption is given by $RA =$

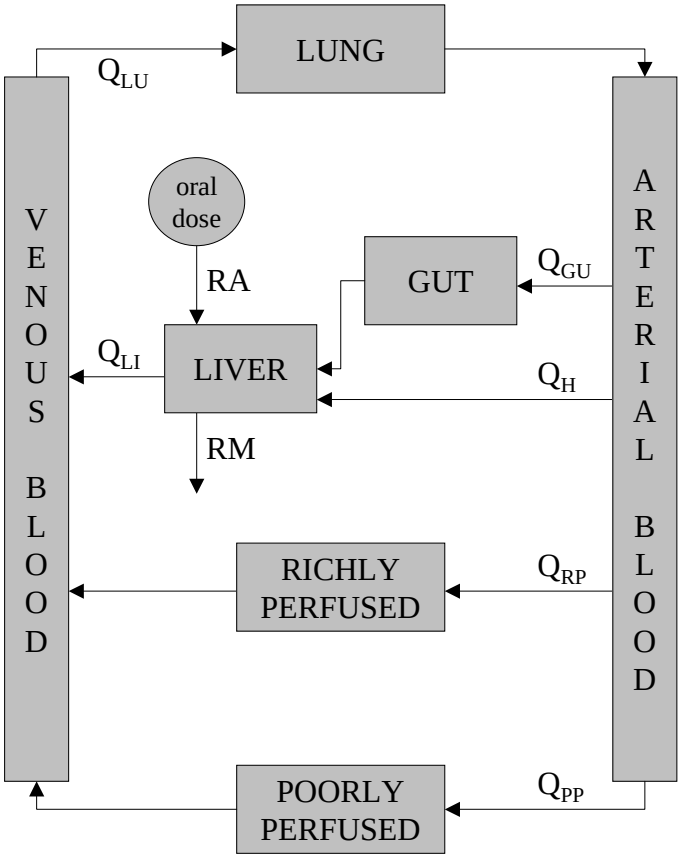

Figure 1: Physiologically based pharmacokinetic model for a hypothetical, orally administered, hepatically cleared drug. Note that orally administered drug is generally absorbed across the gut wall into the hepatic portal vein. In order to separate any drug that is being absorbed thus from that which is *recirculating* into the hepatic portal vein, via the tissues of the gut, we tend to assume that as far as the model is concerned oral absorption processes input drug directly into the "LIVER" compartment, as shown, rather than the "GUT" compartment.

$k_a \times F \times \text{Dose} \times \exp(-k_a t)$ (first-order absorption) and the rate of metabolism, RM, is given by $V_{max} \times C_{LI}/(K_m + C_{LI})$ (Michaelis-Menten metabolism), where k_a, F, V_{max} and K_m are the first-order absorption rate constant, the fraction of the administered dose that enters the liver, the maximum rate of metabolism, and the concentration of drug in the liver at which exactly half the maximum rate of metabolism occurs, respectively.

This example comes in two parts: in 'part 1' we use the above model to simulate a small set of data; in 'part 2' we use the same model to perform Bayesian inference on those data.

3.1 Five compartment model: Simulation

Start your copy of OpenBUGS and open the file

`Diff/Examples/FiveCompartmentModel01.odc`

The following notes describe/explain various parts of the model code – the line numbers to which they relate are given in the right-hand margin of the above file. The file contains code for both parts of the example, colour-coded as follows: code that is only relevant to part 1 of the example is shown in blue; code that is only relevant to part 2 of the example is shown in red; and code that is relevant to both parts is shown in black. Relevant points are

2: `n.grid` is the number of time-points and `dim` is the number of differential equations. The BUGS language command `ode()` calls the differential equation solver. The arguments are as in previous examples.

3–10: Specification of the system of ordinary differential equations (ODEs) to be solved. The compartment names are mapped to numbers in the data block beneath the model code (e.g. `PP = 1`, `RP = 2`, etc.) Note that the rate of oral absorption (`RA`) and the rate of metabolism (`RM`) are defined outside this equation block (on lines 16–18).

21: `diff.five.comp(.)` is the BUGS language command for the compiled component. Its arguments are the same as those in previous examples except that the number of parameters is different.

30: We want our data set to comprise simulated concentrations of drug in venous blood and so we specify `solution[i,VEN]` as the mean for each `data[i]` (`i` = 1, ..., `n.grid`).

33: For simulating the data we specify a fixed value of 10000 (`tau.simulate`) for the `tau` parameter whereas for Bayesian inference (part 2) we specify a (minimally informative) gamma prior.

34–37: A log-normal prior with a precision, on the log-scale, of `log.param.prec` = 13.81 is specified for each K_{P_T}, $T = 1, ..., 6$. This level of precision corresponds to up to two-fold variation (from the median) across the central 99 per cent of the distribution – thus the prior probability that a given partition coefficient is greater than twice or less than half the value of the prior median is 0.01. The prior median is specified via prior point estimates for each K_{P_T} contained in `data.KP[.]` – see the block of data beneath the model code.

38–39: The prior specified for `log.ka` is exactly analogous to that specified for each element of `log.KP[.]`.

3.1.1 Running the simulation

In order to reproduce the selected results shown at the foot of the model file, you should monitor the `log.KP`, `log.ka` and `solution` variables and run the simulation for 1000 iterations. The plot is of simulated venous blood concentrations versus time and was generated by entering `solution[,6]` and `grid` in the `node` and `axis` fields, respectively, of the `Comparison Tool` dialogue box (select `Compare...` from the `Inference` menu), and by then pressing the `model fit` command button. The data set to be used in part 2 of our example is given by the current value of the `data` vector and can be obtained by selecting `Save State` from the `Model` menu – note that the current values of `log.KP[.]` and `log.ka`, i.e. those used to generate the data, are also given.

3.2 Five compartment model: Inference

The OpenBUGS code required in order to run the inference calculation on the data generated in the previous section can be found in the file

```
Diff/Examples/FiveCompartmentModel02.odc
```

The model in this file is the inference version of the problem and is also calling the compiled component version of the model only. This is the code in red, and is described in detail below.

Note that the data for this model now contains the numerical values which were obtained by the `Save State` command in the previous section.

The model is then run for 8000 iterations and the first 4000 again excluded from the computation of the node statistics. The table of results is shown after the MCMC initial values.

The plot below this is of 'model predicted' together with 'observed' venous blood concentrations against time – this was generated by entering `solution[,6]`, `data` and `grid` in the `node`, `other` and `axis` fields, respectively, of the `Comparison Tool` dialogue box, and by then pressing the `model fit` command button.

3.3 Compiled component

The new module is called `DiffFiveCompModel` and can be found in the file

```
Diff/Mod/FiveCompModel.odc
```

This is a Component Pascal module of the same form as the two previously discussed. As you can see, the main differences between this one and the previous examples (shown in blue) are

1. Number of equations, now set to be `nEq = 7`
2. `Derivatives()` function now computes the differential equation right hand sides as given by the equations at the beginning of this section
3. Some constants are declared at the beginning of the module to describe
   (a) labels for the various parameters in the equations (`Q_PP` *etc*)
   (b) labels for the compartments (`PP` *etc*)

These bits are highlighted in blue in the file. This module of course has to be associated with a command in the BUGS language, `diff.five.comp`, and again this is done in the file

```
Diff/Mod/External.odc
```

4 Change points example

All of the previous examples have first derivatives which are smooth functions of time (i.e. the second derivatives, with respect to time, are all continuous). OpenBUGS is capable of handling situations where this is not the case but it requires additional information regarding the times at which discontinuities occur. It needs this information so that it can break up the problem into "blocks" of time where, throughout each block, all second derivatives are continuous. Because specifying such models is somewhat more complicated than with their smooth counterparts, OpenBUGS provides an additional BUGS-language component (`ode.block`) for solving these problems. The example from this section is also provided as a compiled component (`DiffChangePoints`).

4.1 Population PK Model

Our model for this example is designed to illustrate several features of 'advanced' OpenBUGS use, such as the specification of population models, handling discontinuities in time, and allowing compartments to empty into each other. In pharmacokinetics, a population model is required when, in order to draw inferences about the target population, a study is conducted whereby a number of healthy volunteers or patients each receive one or more doses of the drug under investigation and concentration measurements are taken from each one.

Typically we have a (parametric) structural model that describes the shape of the concentration-time profile and we model variability throughout the study population by allowing the parameters of that structural model to differ between individuals. The structural model for this example is shown in Figure 2. The human body is represented by a single compartment ("Compartment 1") and as such is assumed to be homogeneous – the blood and all organs/tissues contain the same concentration of drug, C, which is given by the total amount of drug in the body at time t, $A_1 = A_1(t)$, divided by an *apparent* volume of distribution, V. Drug is eliminated (or *cleared*) from the body via a 'flow-rate' of CL (*clearance*) – the rate of elimination, i.e. the rate at which drug leaves the body, is given by $\frac{dA_{el}}{dt} = CL \times C = CL \times A_1/V$. Compartments 2 and 3 contain doses of drug to be administered, to Compartment 1, at times $t = t_{b2}$ and $t = t_{zo}$, respectively. These have been incorporated purely for illustrative purposes – they are actually unnecessary as we could easily specify the model using Compartment 1 alone (with an

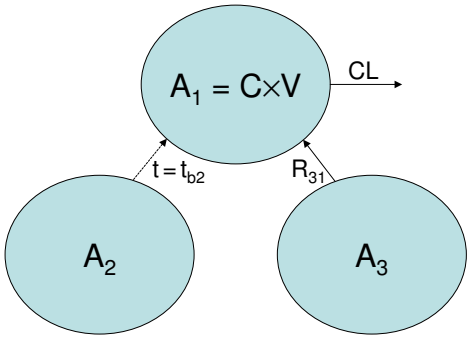

Figure 2: Structural three compartment model used in the change points example – see text for details.

appropriate sequence of initial conditions for each block of time).

The full sequence of events that generates our model is described as follows. At time $t = 0$, the volunteer/patient receives an intravenous bolus dose [1] of size D. At time $t = t_{b2} > 0$ a second bolus, of the same size, is administered. This is the process represented by the dashed arrow from Compartments 2 to 1 in Figure 2 – dashed arrows represent processes whereby one compartment empties into another instantaneously at the specified time ($t = t_{b2}$ in this case) whereas normal arrows denote *rates* of movement of drug. At time $t = t_{zo} > t_{b2}$ a zero-order process is initiated between Compartments 3 and 1 whereby the dose initially in Compartment 3 (again, D) is transferred to Compartment 1 at a constant rate given by D/TI, where TI denotes the duration of the zero-order input process. Clearly this process must terminate at $t = t_{zo} + TI$. All of this means that the somewhat abstract quantity R_{31} in Figure 2, which simply represents the rate of movement of drug from Compartments 3 to 1, is a *piecewise smooth* function of time. Explicitly, it is given by

$$R_{31} = \begin{cases} 0 & 0 \leq t < t_{b2} \\ 0 & t_{b2} \leq t < t_{zo} \\ D/TI & t_{zo} \leq t < t_{zo} + TI \\ 0 & t \geq t_{zo} + TI \end{cases}$$

More generally, we can write $R_{31} = pw(v, o, t)$, where v and o are vectors containing each smooth function and the times at which they begin, respec-

[1] Intravenous boluses are such that the whole dose is administered, as quickly as possible, directly into venous blood – the time taken to deliver the dose is generally modelled as zero and the rate of input is effectively infinite.

tively; in this case, $v = (0, 0, D/TI, 0)$, $o = (0, t_{b2}, t_{zo}, t_{zo} + TI)$ and $pw = v_b$, where

$$b = \sum_i i \times I(t \in [o_i, o_{i+1})) \qquad (o_5 = \infty),$$

that is, b simply specifies which 'block' of time, defined by a pair of 'change-points', contains the current value of t. OpenBUGS provides a new BUGS-language component called `piecewise(.)` that can be used for specifying arbitrary piecewise-smooth functions within differential equation models. Only the vector v is required as an input parameter, however. This is because the 'origins' in o are passed into the associated `ode.block(.)` component instead, so that it knows where to break the problem up into 'smooth' sub-problems (t is defined/controlled by the ODE solving algorithm). For this reason, the o vector passed into `ode.block(.)` must comprise the union of all 'change-points' in the model and each instance of `piecewise(.)` must be defined in terms of a smooth function vector (v) of the same length. This is why R_{31} above incorporates components for both $[0, t_{b2})$ and $[t_{b2}, t_{zo})$ even though it has the same value throughout both intervals: `ode.block(.)` needs to know that a discrete event occurs at $t = t_{b2}$ and so all of its piecewise parents (just R_{31} in this case) must be 'split' accordingly.

The system equations are as follows:

$$\begin{aligned} \frac{\mathrm{d}A_1}{\mathrm{d}t} &= R_{31}(t) - CL \times A_1/V \\ \frac{\mathrm{d}A_2}{\mathrm{d}t} &= 0 \\ \frac{\mathrm{d}A_3}{\mathrm{d}t} &= -R_{31}(t) \end{aligned}$$

(Note that neither bolus dose is represented in the system equations. This is because bolus doses are instantaneous and are thus best described via the initial conditions, as we discuss below.) The structural model is completed by the specification of a sequence of initial conditions for each compartment. First in the sequence are the initial conditions proper, which pertain to the time origin given by o_1. Note that the easiest way in which to model the first intravenous bolus dose is to simply specify $A_1(o_1) = A_1(0) = D$ in the initial conditions. At each subsequent origin (e.g. o_2, o_3, o_4) we may, generally speaking, wish to apply an instantaneous adjustment to the system to account for certain types of discrete event that may have occurred, such as the administration of a bolus dose (which cannot be represented by a finite rate). In OpenBUGS we specify such adjustments by declaring an amount (of

whatever quantity the differential equations are to be solved for) to be added to each compartment at the appropriate time. If no adjustment is specified (or if a value of zero is declared) then the relevant part of the system is left unchanged. Thus for the example above, we specify the following matrix of 'initial conditions', where rows correspond to origins (o_1, o_2, o_3, o_4) and the columns represent compartments:

$$\begin{pmatrix} D & D & D \\ +A_2 & -A_2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

The $-A_2$ in column 2 represents the change to the solution to the second differential equation at the relevant time, i.e. $t = t_{b2}$ (the second 'origin'). Thus the amount of drug in Compartment 2 instantaneously changes from $A_2(t_{b2-})$ to zero at $t = t_{b2}$ (because its value is reduced by $A_2(t_{b2-})$). In contrast, the amount of drug in Compartment 1 changes from $A_1(t_{b2-})$ to $A_1(t_{b2-}) + A_2(t_{b2-})$. In other words, Compartment 2 instantaneously empties into Compartment 1 at time $t = t_{b2}$. Since Compartment 2 initially contains an amount D of drug and $\frac{dA_2}{dt} \equiv 0$, this is equivalent to an intravenous bolus dose being administered, to Compartment 1, at $t = t_{b2}$.

The *statistical* model is defined as follows. First note that the measurable quantity here is *concentration* of drug in Compartment 1, i.e. $C = A_1/V$. This is a function of the dose D, time t, and additional parameters, namely CL, V and TI, that we collectively denote by θ: $C = C(\theta, t, D)$. For this example we will allow both θ and D to vary between individuals, although the doses are assumed known, as they would be in practice (normally). Suppose we have n concentration measurements, indexed by j, from each of K individuals, indexed by i. We denote these by y_{ij}, $i = 1, ..., K$, $j = 1, ..., n$. Often in pharmacokinetics, the size of the error associated with each concentration measurement is proportional to the underlying true concentration and so we tend to model the natural logarithm of the data as a function of $\log C$. At the first stage of the statistical model we assume

$$\log y_{ij} = \log C(\theta_i, t_{ij}, D_i) + \epsilon_{ij}, \qquad i = 1, ..., K, \quad j = 1, ..., n,$$

where t_{ij} denotes the time at which y_{ij} was collected and the ϵ_{ij} terms are independent and identically distributed normal random variables with mean zero and variance τ^{-1} (i.e. precision $= \tau$). At the second stage of the model we have

$$\theta_i \sim \mathrm{MVN}_3(\mu, \Omega), \qquad i = 1, ..., K,$$

where μ and Ω are the population mean and the variance-covariance of pharmacokinetic parameters, respectively. At the *final* stage of our population model, appropriate multivariate normal, Wishart and gamma priors are specified for μ, Ω^{-1} and τ, respectively.

4.2 Change points example, BUGS code (A)

In this section, the BUGS code for a simplified version of the above model is described. In order to make clear the differential equations, we first consider a single individual. This model can use either BUGS language code for the differential equations or the compiled component. Use of the compiled component version is described in the next section.

The model, data and inits are in the file

```
Diff/Examples/ChangePointsModel01.odc
```

with line numbers on the right hand side. Its relevant features are:

1: `n.grid` is the number of time points

2,3: `log.data[j]` is the measured log-concentration at time point `j`, `log.model[j]` is the log of the solution to the differential equation $C(\theta, t_j, D)$

5,6: `c.language` and `c.hard.wired` are the solutions of the differential equations computed from the BUGS language and compiled component versions respectively

8-15: parameters of the differential equations; the first three are multivariate normal and the last is fixed at the dose value D

16-20: BUGS language version of the solution; we call the function `ode.block()` instead of `ode()`; since we are now dealing with time blocks, the `origins` parameter is now an array rather than a scalar

21-25 The compiled component version of the solution is associated with the command `diff.changepoints`. Necessary parameters are passed to it in the array `parameters` which includes the values of θ and D

26-28: BUGS language specification of the right hand sides of the differential equations, same notation as before.

29-33: Specification of the piecewise constant function associated with the time "blocks"

34-38: Specification of the ODE initial conditions; in the array `inits` the row index labels the time origins and column index labels the compartments. The first row are the true initial conditions of the problem. Subsequent rows represent additive adjustments which are made to the each compartment at the corresponding time point.

For example, at the second time origin compartment 2 is emptied into compartment 1, represented here by an adjustment of $-A[2]$ to compartment 2 and an adjustment of $+A[2]$ to compartment 1. This is the "second bolus" of the model. (Elements of the array `inits` which are not specified are taken to be zero.)

39-42: Origins of the time "blocks"

4.3 Change points example, BUGS code (B)

This section concerns the BUGS language code for the "population" version of the model. The code is in the file

```
Diff/Examples/ChangePointsModel02.odc
```

and the line numbers are on the right hand side. There are two main differences between this and the previous version of this model. First is that the arrays for the solution of the differential equations (and various other quantities in the model) now have an index i representing individuals from $1, ..., K$. There is also an associated loop over i in the model code.

Secondly, with a population model we can now make inferences on the population distribution of θ_i. Thus one does not have to specify values for `mu` and `omega.inv`, but can specify prior densities for them instead. This is described in detail below.

Relevant points about this model are:

6: This loop is over individuals, the number of which is `n.ind`

8-17: The arrays storing the solution of the differential equations $C(\theta_i, t_{ij}, D_i)$ now require an index i to distinguish between different individuals

19-23: Parameters for the differential equations; the arrays also require an index i for individuals

27-30: The solution of the differential equations computed by the BUGS language specification and the compiled component

27,28,31-33: Note that when the BUGS language specification of the problem is used, a variable `t` is used to represent the time variable. When a population model computation is performed, the index `i` representing individuals **must** be included, specified using the notation `t[i]`

34-38: Piecewise function associated with the "blocking" of the differential equations; again, the index i is used in the array

39-47: Initial values and time block origins. These have the same interpretation as above, and the arrays also have the index i

49-51: Prior density for the parameters of the multivariate normal population distribution of θ_i. These are outside of the loop over individuals i. We require two quantities, the mean vector `mu` and the precision matrix `omega.inv` which are multivariate normal and Wishart distributed respectively.

The remainder of the code in this file gives data and MCMC initial values for the computation. Summary statistics are shown at the end of the file for an example run.

4.4 Change points example, compiled component

Component Pascal code for the system of differential equations used in this example can be found in the file `Diff/Mod/ChangePoints.odc`. This module is associated with the BUGS language command `diff.changepoints` as specified in the file `Diff/Mod/External.odc`.

This module is linked into the OpenBUGS software via the new BUGS-language component `diff.changepoints()`, this being specified in the file `Diff/Mod/External.odc`.

We close this section with a few notes on the Component Pascal code contained within `Diff/Mod/ChangePoints.odc`. Relevant parts of the code are highlighted in blue.

1. The number of equations is

```
CONST
  nEq = 3;
```

2. The `Derivatives()` procedure. As discussed previously, OpenBUGS splits the time-grid up into m, say, (user-specified) blocks of time, throughout which it can assume all of the ODEs to be smooth. The `e.Block()` procedure automatically returns an integer value informing us of which block of time is currently of interest to the ODE solver; we use the integer variable '`block`' in order to store this value, an integer between zero and $m-1$ (inclusive). This can be used in order to define any piecewise functions that may be required to fully specify the equations. In this procedure we make use of a Component Pascal "CASE" statement to define the value of `R31`: for each possible value of `block` $(0, 1, 2, 3)$, a corresponding value for `R31` is specified.

3. The "`Adjust`" procedure allows us to specify additive adjustments to the system of differential equations to be applied at the beginning of specific blocks of time – the procedure is called automatically by OpenBUGS as soon as the ODE solver begins work on a new 'block'. Again we make use of an integer variable named `block`, and of the `e.Block()` procedure, to store the index of the current block of time. This is then used to form a CASE statement whereby any necessary adjustments can be associated with the correct change-point/origin. Note that within the `Adjust` procedure the possible values of `block` are $1, 2, ..., m-1$; that is, adjustments cannot be made at the beginning of 'block 0' because that is the time origin, when the initial conditions proper apply. Note also that 'empty statement sequences' can be specified if no adjustment is required, as is the case at the beginning of the blocks indexed by `block = 2` ($t = t_{zo}$) and `block = 3` ($t = t_{zo} + TI$). At the beginning of the block indexed by `block = 1`, i.e. at time $t = t_{b2}$, we wish to allow Compartment 2 to empty instantaneously into Compartment 1. As we have direct access to the numerical solution to the ODEs (i.e. the `C[.]` array/vector) within the `Adjust` procedure, this is somewhat easier than with the BUGS-language specification. We describe the Compartment 1 adjustment in much the same way (`C[0] := C[0] + C[1]`), but, rather than specifying `C[1] := C[1] - C[1]`, as we have essentially done with the BUGS-language specification, we can write `C[1] := 0` instead.

Part III

Custom Compiled Components

5 Create your own module

This section describes how to create your own compiled module to compute the right hand sides of a system of differential equations, how to link it into OpenBUGS and make it available for use.

5.1 BlackBox Developer

In order to create any compiled components for OpenBUGS the BlackBox development environment is used. This must be installed on your computer, as described in the document "Use of BlackBox with OpenBUGS" — so please read that before proceeding.

5.2 Template module

The starting point for a differential equations component is a template module which contains all of the code necessary to create the custom module except for the specification of the number of differential equations and the equations themselves (*via* the `Derivatives()` function).

It is to be found in the file

```
Diff/Mod/TemplateModule.odc
```

The code in this module does not do anything. It is merely for you to copy as a starting point for your own module. The detailed steps to follow in order to do this are below.

5.3 Steps to follow

In the template module, there are comments (enclosed by the `(*` and `*)` signs) giving descriptions of various bits of code. The most important thing

to note is that where something is described as not to be altered by the user, please do not alter it!

To edit and compile code, we use the BlackBox application started through the link/shortcut created in the document "Use of BlackBox with OpenBUGS". To compile any file of Component Pascal code, make sure that file has focus, and type `Control-K`.

The detailed steps to create a new module are:

1. Open the file, and save it as another name first. The rest of this example assumes the file has been saved as `Diff/Mod/MyModule.odc`

2. Change the module name. This must correspond with the file name such that if the file name is `Diff/Mod/MyModule.odc` then the module name inside the file **must** be `DiffMyModule`. This must be changed at the top of the file and right at the end:

   ```
   MODULE DiffTemplateModule;
       (* everything else in here *)
   END DiffTemplateModule.
   ```

   should be replaced with

   ```
   MODULE DiffMyModule;
       (* everything else in here *)
   END DiffMyModule.
   ```

   Note the use of the semi-colon and the full stop. The various compiled components in previous sections are of this form.

3. Now with the file `Diff/Mod/MyModule.odc` open, the first thing to change is the number of equations in your system. This is near the top of the file

   ```
   CONST
     nEq = 0;
   ```

   and was set to zero in the template module. For example, it is 1 for the exponential model, 2 for the Lotka-Volterra system, and so on.

4. Now the function/procedure that computes the right hand sides of the differential equations can be filled in. The argument list of this function must not be changed. Variables local to the call to `Derivatives()` can be declared after the `VAR`; for example

   ```
   VAR
     x: REAL;
     i: INTEGER;
   ```

   creates a floating point (8-byte, same as `double` in C/C++) and an integer variable (4-byte, same as `int` in C/C++).

5. Array indices in Component Pascal go from 0 to $N - 1$ when there are N elements in the array, the same convention as in the C/C++ language. See the Lotka-Volterra compiled component for an example.

6. Some useful mathematical functions are available in the module `Math` and can be accessed by typing `Math.Sqrt(x)`, *etc.* To see what is available in the module `Math`, double click on the name in the import list at the beginning of the module and then right click, and select the option `Documentation`. This will open the documentation file of that module containing function names and descriptions of what they do.

7. Now we select a command to use to make available the new code in the BUGS language. Open the file

   ```
   Diff/Mod/External.odc
   ```

   and add a line to it of the form

   ```
   Register("my.command", "DiffMyModule.Install");
   ```

   which means that you will be able to use the new code with the command `my.command` in the BUGS language. Note that it is up to you to make sure that this string does not conflict with anything that is in BUGS already. A good way to do that would be to use your or your company's name as part of the string.

   For the exponential decay model from the first section of this document, we use:

   ```
   Register("diff.exponential", "DiffExponential.Install");
   ```

8. When a new item has been added to `External.odc` in this way, the file must be compiled in order for the command to be available in OpenBUGS. Type `Cntrl-K` after any alteration to this file.

5.4 Examples of `Derivatives()` functions

For examples of code, have a look at the exponential, Lotka-Volterra and five-compartment models.

5.5 Debugging tips

1. The number of parameters which are passed to `Derivatives()` is determined by the BUGS language model code. So for example, if your model code contains something like

   ```
   solution[...] <- my.command(..., parameters[1:2], ...)
   parameters[1] <- 1.0;
   parameters[2] <- 2.0;
   ```

   and the `Derivatives()` procedure attempts to do

   ```
   x1 := parameters[1];
   x2 := parameters[2];
   ```

   there will be an error because array indices start at 0. If you get an error related to reading the array `theta`, something like this might be the source of the problem.

2. Any of the arguments of `Derivatives()` or any of the quantities computed therein may be printed out to the Log window (opened from the `DevInfo` menu) by use of the facilities in module `Out`. Suppose you have a variable called x, then it might be printed out by adding the lines of code

   ```
   x := 0.0; (* or whatever it should be *)
   Out.String("x = "); Out.Real(x, 8); Out.Ln();
   ```

   Remember that this will be done every time `Derivatives()` is called, which for an ODE integration embedded within a MCMC computation is a lot.