Practicals

Health Economics in R

Karolinska Institutet, Stockholm, 9–10 June 2022

Cities partnerships Programme Seed Funding 2021-22

Nathan Green (UCL)

Contents

Preface

All the material needed to do these practical exercises is provided for you at the course website or in a zip file. This has one folder for each practical session / chapter of this document, and contains program code and data for BUGS and R.

Some of the practical sessions have a file containing solutions to the exercises, in the corresponding folder. For some of the exercises, solutions are not necessary, since they simply consist of stepping through a script that has been provided.

0.1 Acknowledgements

This material is part taken from the annual Summer School in Bayesian Health Economics. Lectures and practical were created by Gianluca Baio, Howard Thom, Anna Heath, Nicky Best, Chris Jackson and others. Thanks to all those who have contributed to this work.

1

A (too short) introduction to R

This is a very brief introduction to R (which can be downloaded from the website www.r-project.org) and its capabilities. It will be extremely focussed on the characteristics that are instrumental to do health economic evaluations using a combination of R, BUGS and some useful packages (such as BCEA). Thus it is by no means exhaustive!

When you open the R terminal, you are presented with the possibility of typing commands. You may want to open a text editor (e.g. the simple one built into the R Windows interface) in which you can type directly these commands, and save them to a script for future use. Another possibility is to use R from within a more sophisticated "integrated development environment", such as RStudio (www.rstudio.com), which has many more features than the basic Windows interface.

In any case, R is a very powerful tool; more importantly, it is free and you can find a wealth of documentation on the internet. R has a set of built-in commands, which you can use for basic operations. However, there are also many add-on packages containing sets of functions designed to perform specific statistical tasks. These packages can be installed to *your* R by typing the command

```
> install.packages("package_name")
```

(assuming you have an internet connection and noticing that the symbol > indicates the beginning of a line of code in R). This command only needs to be executed one time. Once a package is installed in your local library (a collection of packages) you can make it available to the current R session by typing the command

```
> library(package_name)
```

For these practicals, you will need to install and load the packages:

- BCEA, which can be used to post-process the results of a (Bayesian) model to perform a health economic evaluation.
- R2OpenBUGS, which can be used to interface R and BUGS.

You do this by typing in your R terminal the commands

```
> install.packages("BCEA")
> install.packages("R2OpenBUGS")
> library(BCEA)
> library(R2OpenBUGS)
```

Both BCEA and R2OpenBUGS will automatically load other packages that they *depend on* — this means that in order to work, they need to access functions that are part of other packages.

If you wish so, you can use JAGS in the practicals. To this end (and assuming you have actually installed the current version of JAGS to your computer), you will need to also install the package R2jags, which you can do by typing in your R terminal

```
> install.packages("R2jags")
```

Notice that if you decide to use JAGS instead of BUGS, you will need to slightly modify some of the commands — we describe this in more details later in this manual.

Once a package is loaded to your R workspace, you can type the command help(package_name), which will open a window displaying a description of the package. For example help(BCEA) provides some basic information (including details of the current version). You can use the command help also on specific functions within the package, e.g. typing help(bcea) describes in detail how to use the bcea function (notice that in this case the package name is typeset in uppercase, while the function is lowercase!).

The very basic commands that are required to do a typical R session working with BUGS and BCEA will be given and described later or in the scripts that we refer to in the practicals.

2

Decision trees

This is a gentle introduction to implementing decision trees for health economics in R. This is meant as an early practical to familarise with using R and some of the basic concepts.

2.1 Running and analysing deterministic results

First run the simple depression model stored in file `practical.R` in the decision tree folder. Ensure you understand each line of this file. Recall that the model equations are

```
costs <- c.treat+p.rec*(1-p.rel)*c.rec+p.rec*p.rel*c.rel+(1-p.rec)*c.norec
effects <- p.rec*(1-p.rel)*q.rec+p.rec*p.rel*q.rel+(1-p.rec)*q.norec
```

Look at the values of the matrix parameters (e.g. `p.rel`, `c.treat`) going into this equation.

- What is the Net Benefit?
- Calculate the incremental costs and effects.
- What is the ICER?

2.2 Running and analysing probabilistic results

First run the probabilistic depression model stored in file `practical_probs.R` in the decision tree folder. Ensure you understand each line of this file.

Look at the values of the matrix parameters (e.g. `p.rel`, `c.treat`) going into this equation using the `colMeans()` function. This takes a mean of the columns matrices; for example, `colMeans(p.rel)` will give the mean probability of relapse on each of the three treatment options. Look at the mean of vectors (e.g. `c.rec, c.rel`) using the `mean()` function. A quick way to check if a data structure is a matrix or vector is to use `dim()`, the dimensions of a matrix, as this will be `NULL` for a vector.

- Can you tell which treatment has the highest average probability of recovery or lowest probability of relapse?
- Of cost of no recovery, relapse, and recovery, which has the highest mean?
- Of QALY associated with no recovery, relapse, and recovery, which has the highest mean?
  Now that you understand the inputs to the costs and effects, use the `colMeans()` function to find the treatment with lowest costs and highest effects. The net benefit at Âč20,000 is defined as
  `net.benefit <- 20000*effects - costs`
  Note that this multiplies 20000 by all the elements of the effects matrix and subtracts the corresponding elements of the costs matrix.
- Which intervention has the highest mean net benefit and should be recommended for treatment of depression?

2.3 Using BCEA to compare depression treatment strategies.

We will now use the `BCEA` package to analyse the effects and costs matrices in`depression_psa.RData`. This will contrast the difficulty of simply comparing mean costs, effects, and net benefits (exercise 1) with a fully Bayesian and probabilistic interpretation of the results. First load the `BCEA` package using

```
library(BCEA)
```

If BCEA has not yet been installed youâĂŹll need to call `install.packages(âĂBCEAâĂİ)` first.

- First use the `bcea()` function to generate a bcea object summarising the costs and effects Use the options `ref=1` to specify that âĂIJno treatmentâĂİ is the reference and interventions=t.names to specify the appropriate names of the interventions.
- Apply `summary()` to the object created by `bcea()` in part (a) above. Use the option `wtp=20000` so that the willingness-to-pay for the net benefit is Âč20,000 (default in BCEA is Âč25,000) This gives comparisons of CBT and antidepressants to no treatment. The âĂIJEIBâĂİ is expected incremental benefit at the `wtp=20000`, the âĂIJCEACâĂİ (cost-effectiveness acceptability curve) is the probability that the reference of âĂIJno treatmentâĂİ has highest net benefit (most cost-effective) at the specified willingness-to-pay, and the ICER is the incremental cost-effectiveness ratio. The last of these can be compared with the standard willingness-to-pay threshold of Âč20,000. On these measures, how do CBT and antidepressants compare to no treatment?
- Now apply `bcea()` and `summary()` to compare the CBT and antidepressants option. To do this, first use `bcea()` but with `ref=2`, giving comparisons relative to CBT. Now use `summary()` to get the EIB and CEAC of antidepressants relative to CBT. Which option would be recommended at a willingness-to-pay threshold of Âč20,000? Note that the ICER is difficult to interpret due to negative incremental costs, so only focus on EIB and CEAC.
- As there are three decision options, it may be better to compare them simultaneously, rather than doing the pairwise comparisons of (b) and (c). Pass the `bcea` object created in part (a) to the `mulit.ce()` function and store the result. Now use `ceac.plot()` on the output of `multi.ce()`. This gives the probability that each of the three options has the highest net benefit for a range of willingness-to-pay thresholds. Which treatment has the highest probability of being most cost-effective at the Âč20,000 threshold?

3

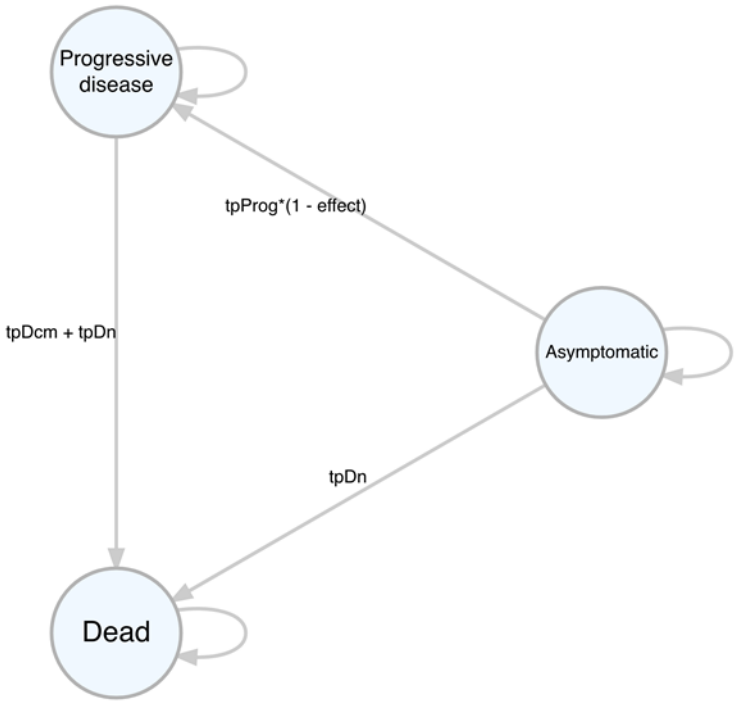
Introduction to Markov models

This is rather long practical to implement a complete Markov model for cost-effectiveness analysis. Extensions to the basic model will be made as we go. The R code is available in a separate file `practical.R` and the full code in `practical_solutions.R` so you don't have to cut and paste everything.

3.1 Three-state model

This model is take from an example used in Briggs A, Sculpher M. Introducing Markov models for economic evaluation. PharmacoEconomics 1998; 13(4): 397-409. and Briggs AH. Handling uncertainty in cost-effectiveness models. PharmacoEconomics (2000). The model is freely available as an Excel spreadsheet which we provide. In the following we will ask you to use the spreadsheet to complete the R code provided.

Consider a three-state model is used to describe how patients transition between health states with a chronic disease. Patients begin in an asymptomatic health state, which means that the patient has developed the chronic disease but has so symptoms. Patients can stay in the asymptomatic health state or move to a progressive health state which means that the patient is experiencing symptoms of the disease. Patients can move from the asymptomatic health state to the dead health state at the same rate as the general population without the disease. Patients can stay in the progressive health state or move from the progressive health state to dead, but at an increased risk of death. The dead state is an absorbing state as a patient cannot change from being dead.



- Take a look at the original Excel spreadsheet and understand how it implements the model and what the particular characteristics are. What are the initial values? How does it incorporate age-dependent transitions?

Open the R script "practical.R" in the markov model folder. Look over the code and try and understand what the commands are doing. Next we will step through the script.

We will assume that a cohort of 1000 patients receives a drug and compares to a cohort of 1000 patients that does not receive a drug to assess how patients move throughout the health states over time with a chronic disease. To set up the model in R some definitions are needed first. We outline the R code used to define the number of treatments (prefixed with `t_`) and their names (prefixed with `n_`), the number of states (prefixed with `s_`) and their names. The number of cycles starting at 1 (not 0 as in spreadsheet models) is specified, as well as the initial age of patients in the cohort beginning at 55.

```
t_names <- c("without_drug", "with_drug")
n_treatments <- length(t_names)

s_names  <- c("Asymptomatic_disease", "Progressive_disease", "Dead")
n_states <- length(s_names)

n_pop <- 1000

n_cycles <- 46
Initial_age <- 55
```

The unit costs and unit utilities associated with each of the health states as well as the cost of the drug need to be defined. The utility of being in the dead health state is 0 so does not need to be defined here. Costs begin with a c and utilities with a u. The discount rate for costs and outcomes are also included at a rate of 6%.

- Initialise the variable using the values in the spreadsheet.

```
cAsymp <- ?
cDeath <- ?
cDrug <- ?
cProg <- ?
uAsymp <- ?
uProg <- ?
oDr <- ?
cDr <- ?
tpDcm <- ?

# transition matrix variables
tpProg <- 0.01
tpDcm <- 0.15
tpDn <- 0.0138
effect <- 0.5
```

This process of defining treatment names, states and cycles is similar to that in MS Excel. One addition in R is that space for when calculations are performed is needed. It is like creating the cells in MS Excel. By creating matrices, empty space for costs and utilities in each health state for patients with or without the drug is specified. The structure of the matrices for the cost of transition to a health state, and cost and QALYs accrued being in health state are the same. We describe the cost of transitioning to a state, in this case only for the dead state. The first argument defines a vector of values for each health state and for each cohort, similar to a column of values in a parameters sheet in Excel. The argument `byrow = TRUE` makes sure that all the states for the first cohort are defined first and then all the states for the second cohort. The `trans_c_matrix` creates empty space and assigned the cost of Âč1000 for transitioning to the Dead state. The first line is for the cohort who are without a drug, and the second line for the cohort that are with a drug.

```
# cost of staying in state
state_c_matrix <-
  matrix(c(cAsymp, cProg, 0,
           cAsymp + cDrug, cProg, 0),
         byrow = TRUE,
         nrow = n_treatments,
         dimnames = list(t_names,
                         s_names))
```

```
# qaly when staying in state
state_q_matrix <-
  matrix(c(uAsymp, uProg, 0,
           uAsymp, uProg, 0),
         byrow = TRUE,
         nrow = n_treatments,
         dimnames = list(t_names,
                         s_names))

# cost of moving to a state
# same for both treatments
trans_c_matrix <-
  matrix(c(0, 0, 0,
           0, 0, cDeath,
           0, 0, 0),
         byrow = TRUE,
         nrow = n_states,
         dimnames = list(from = s_names,
                         to = s_names))
```

Space is also needed to define the transition probabilities between health states. A matrix is created as above but with another dimension for the movements between health states. The resulting matrix shows transitions between each health state dependent on the cohort being with or without a drug. We then insert specific values.

- The following arrays are 3-dimensional. Why do you think the dimensions have the order they do? Print the array to screen for a clue.

```
# Transition probabilities
p_matrix <- array(data = 0,
                  dim = c(n_states, n_states, n_treatments),
                  dimnames = list(from = s_names,
                                  to = s_names,
                                  t_names))

## assume doesn't depend on cycle
p_matrix["Asymptomatic_disease", "Progressive_disease", "without_drug"] <-
    tpProg
p_matrix["Asymptomatic_disease", "Dead", "without_drug"] <- tpDn
p_matrix["Asymptomatic_disease", "Asymptomatic_disease", "without_drug"] <-
     1 - tpProg - tpDn
p_matrix["Progressive_disease", "Dead", "without_drug"] <- tpDcm + tpDn
p_matrix["Progressive_disease", "Progressive_disease", "without_drug"] <- 1
     - tpDcm - tpDn
p_matrix["Dead", "Dead", "without_drug"] <- 1

# Matrix containing transition probabilities for with_drug
p_matrix["Asymptomatic_disease", "Progressive_disease", "with_drug"] <-
    tpProg*(1 - effect)
p_matrix["Asymptomatic_disease", "Dead", "with_drug"] <- tpDn
p_matrix["Asymptomatic_disease", "Asymptomatic_disease", "with_drug"] <- 1
    - tpProg*(1 - effect) - tpDn
p_matrix["Progressive_disease", "Dead", "with_drug"] <- tpDcm + tpDn
p_matrix["Progressive_disease", "Progressive_disease", "with_drug"] <- 1 -
    tpDcm - tpDn
p_matrix["Dead", "Dead", "with_drug"] <- 1


# Store population output for each cycle

# state populations
pop <- array(data = NA,
             dim = c(n_states, n_cycles, n_treatments),
             dimnames = list(state = s_names,
                             cycle = NULL,
                             treatment = t_names))
```

The transition probability matrix is defined upfront in this scenario but later we will see how to very it during run time.

- Set the starting state populations. What are the indices? What are the assigned values?

```
# _arrived_ state populations
trans <- array(data = NA,
               dim = c(n_states, n_cycles, n_treatments),
               dimnames = list(state = s_names,
                               cycle = NULL,
                               treatment = t_names))

trans[, cycle = 1, ] <- 0
```

3.2 Output Variables

A population matrix (`pop`) and transition matrix (`trans`) are created with an additional dimension so there is blank space for each health state, with or without a drug, for each of the 46 cycles. Below shows this process for a generic array (`cycle_empty_array`).

```
# Sum costs and QALYs for each cycle at a time for each drug

cycle_empty_array <-
  array(NA,
        dim = c(n_treatments, n_cycles),
        dimnames = list(treatment = t_names,
                        cycle = NULL))

cycle_state_costs <- cycle_trans_costs <- cycle_empty_array
cycle_costs <- cycle_QALYs <- cycle_empty_array
LE <- LYs <- cycle_empty_array    # life-expectancy; life-years
cycle_QALE <- cycle_empty_array   # qaly-adjusted life-years

total_costs <- setNames(c(NA, NA), t_names)
total_QALYs <- setNames(c(NA, NA), t_names)
```

3.3 Running population only model

For simplicity, to start with we will show how to simulate the number of individuals in each state over time for each treatment without also calculating the other outputs of interest, like the overall costs and QALYS, which we will do later. This will focus on how we iterate over all of the combinations and how we can simplify versions of this. The R code consists of several nested 'for' loops.

Firstly, we can explicitly loops by treatment, time, 'from' state and 'to' state. For each combination of these we take the product of the previous time point population and the associated transition probabilities. Once all subpopulations who move to a given state from all other states are simulated we then sum these.

```
for (i in 1:n_treatments) {
  for (j in 2:n_cycles) {
    for (s in 1:n_states) {
      for (k in 1:n_states) {
        pop_s_from_[k] <- pop[k, cycle = j - 1, treatment = i] * p_matrix[k
            , s, treatment = i]
      }
      pop[state = s, cycle = j, treatment = i] <- sum(pop_s_from_)
    }
  }
}
```

The above formulation is perhaps the most transparent but can be be trivially simplified to obtain an alternative, condensed version by moving the sum operation inside the inner loop, taking advantage of the vector operation in R.

```
for (i in 1:n_treatments) {
  for (j in 2:n_cycles) {
    for (s in 1:n_states) {
```

```
      pop[state = s, cycle = j, treatment = i] <-
        sum(pop[, cycle = j - 1, treatment = i] * p_matrix[, s, treatment =
            i])
    }
  }
}
```

3.4 Running the full model

Let us now run the full model and combine all the information and code from the previous sections. A loop is first created over treatments and then a second loop repeats from cycle number 2 to cycle 46 (n_cycles), as cycle 1 was already defined above, equivalent to cycle 0 rows in Excel models. Recall the matrix multiplication operator %*%, used here to calculate the state population at the next time step (pop) and the number of individuals who transition between states (trans). The cross-product operator allows us to remove all of the 'for' loops over states and the separate summation step.

```
for (i in 1:n_treatments) {

  age <- Initial_age

  for (j in 2:n_cycles) {

    pop[, cycle = j, treatment = i] <-
      pop[, cycle = j - 1, treatment = i] %*% p_matrix[, , treatment = i]

    trans[, cycle = j, treatment = i] <-
      pop[, cycle = j - 1, treatment = i] %*% (trans_c_matrix * p_matrix[,
          , treatment = i])

    age <- age + 1
  }

  cycle_state_costs[i, ] <-
    (state_c_matrix[treatment = i, ] %*% pop[, , treatment = i]) * 1/(1 +
       cDr)^(1:n_cycles - 1)

  # discounting at _previous_ cycle
  cycle_trans_costs[i, ] <-
    (c(1,1,1) %*% trans[, , treatment = i]) * 1/(1 + cDr)^(1:n_cycles - 2)

  cycle_costs[i, ] <- cycle_state_costs[i, ] + cycle_trans_costs[i, ]

  LE[i, ] <- c(1,1,0) %*% pop[, , treatment = i]

  LYs[i, ] <- LE[i, ] * 1/(1 + oDr)^(1:n_cycles - 1)

  cycle_QALE[i, ] <-
    state_q_matrix[treatment = i, ] %*% pop[, , treatment = i]

  cycle_QALYs[i, ] <- cycle_QALE[i, ] * 1/(1 + oDr)^(1:n_cycles - 1)

  total_costs[i] <- sum(cycle_costs[treatment = i, -1])
  total_QALYs[i] <- sum(cycle_QALYs[treatment = i, -1])
}
```

The discount rate is also incorporated into the model here easily as each cycleâĂŹs costs and QALYs will depend on the cycle number. These repeated steps are performed for each of the two treatments (1:n treatments) and the total costs and QALYs over the lifetime of the model can then be calculated for each treatment.

3.5 Plot results

Displaying the results after running the model is easy in R. . These results will therefore assume that the strategy where the cohort are without a drug is the standard of care or base case analysis. Swapping with and without the drug will change this around.

- The incremental cost-effectiveness ratio (ICER) requires the incremental costs and incremental QALYs. Calculate the ICER value.

  Create a simple cost-effectiveness plane as follows:

```
plot(x = q_incr/n_pop, y = c_incr/n_pop,
     xlim = c(0, 1500/n_pop),
     ylim = c(0, 12e6/n_pop),
     pch = 16, cex = 1.5,
     xlab = "QALY difference",
     ylab = "Cost difference (Âč)",
     frame.plot = FALSE)
```

3.6 Cycle-dependent probability matrix

Transition probabilities in a Markov model can either be time-independent, such as tpDm, or time-dependent. Next, define the transition probabilities in the full model depend on age and cycle. To begin with let us only depend on cycle and fix the age varying variable `tpDn` at a middle value. Define the transition probability from Asymptomatic to Progressive disease, previously just `tpProg`, to depend on cycle such that the new transition probability is `tpProg` $\times$ `cycle`.

To account for the time dependency, the hard-coded array in MS Excel has been replaced with a function in R named `p_matrix_cycle` which is called at each cycle iteration. By simply replacing a fixed array with a function, this will decouple the calculation of the transition matrix and the higher-level cost-effectiveness calculations. This makes changes and testing to either part easier and more reliable.

Note that, strictly speaking, in R this is an array but we have named the data structure `p_matrix` to emphasise that it provides what is known in Markov modelling as the transition probability matrix. The time dependent probabilities are those that depend on the age of the cohort. A lookup function is used to describe the transition probability from Asymp to Dead using 6 age group categories.

```
p_matrix_cycle <- function(p_matrix, cycle,
                           tpProg = 0.01,
                           tpDcm = 0.15,
                           tpDn = 0.0138
                           effect = 0.5) {

  # Matrix containing transition probabilities for without_drug
  p_matrix["Asymptomatic_disease", "Progressive_disease", "without_drug"]
      <- tpProg*cycle
  p_matrix["Asymptomatic_disease", "Dead", "without_drug"] <- tpDn
  p_matrix["Asymptomatic_disease", "Asymptomatic_disease", "without_drug"]
      <- 1 - tpProg*cycle - tpDn
  p_matrix["Progressive_disease", "Dead", "without_drug"] <- tpDcm + tpDn
  p_matrix["Progressive_disease", "Progressive_disease", "without_drug"] <-
       1 - tpDcm - tpDn
  p_matrix["Dead", "Dead", "without_drug"] <- 1

  # Matrix containing transition probabilities for with_drug
  p_matrix["Asymptomatic_disease", "Progressive_disease", "with_drug"] <-
      tpProg*(1 - effect)*cycle
  p_matrix["Asymptomatic_disease", "Dead", "with_drug"] <- tpDn
  p_matrix["Asymptomatic_disease", "Asymptomatic_disease", "with_drug"] <-
    1 - tpProg*(1 - effect)*cycle - tpDn
  p_matrix["Progressive_disease", "Dead", "with_drug"] <- tpDcm + tpDn
  p_matrix["Progressive_disease", "Progressive_disease", "with_drug"] <- 1
      - tpDcm - tpDn
  p_matrix["Dead", "Dead", "with_drug"] <- 1

  return(p_matrix)
}
```

3.7 Running the model

The model is run in the same way as the first example.

- Include a new line `p_matrix <- p_matrix_cycle(p_matrix, j - 1)` inside the loops used to run the model. Where should it go?

The cost, QALY, LE, LY and QALE at each cycle are calculated as the `p_matrix` is updated at each iteration of the loop.

3.8 Plot results

- Calculate the ICER value.

```
plot(x = q_incr/n_pop, y = c_incr/n_pop,
     xlim = c(0, 1500/n_pop),
     ylim = c(0, 12e6/n_pop),
     pch = 16, cex = 1.5,
     xlab = "QALY difference",
     ylab = "Cost difference (Âč)",
     frame.plot = FALSE)
```

- Draw a willingness-to-pay threshold at Âč30,000.
- Are the results different to previously? How?

3.9 Age-dependent probability matrix

Now extend the `p_matrix_cycle()` function to include a dependence on age as well as cycle. The additional code looks like the following:

```
p_matrix_cycle <- function(p_matrix, age, cycle,
                           tpProg = 0.01,
                           tpDcm = 0.15,
                           effect = 0.5) {

 # time-dependent age lookup table
  tpDn_lookup <-
    c("(34,44]" = 0.0017,
      "(44,54]" = 0.0044,
      "(54,64]" = 0.0138,
      "(64,74]" = 0.0379,
      "(74,84]" = 0.0912,
      "(84,100]" = 0.1958)

  # discretize age in to age groups
  age_grp <- cut(age, breaks = c(34,44,54,64,74,84,100))

  # map an age group to a probability
  tpDn <- tpDn_lookup[age_grp]

  #############
  # insert here
  # same as previous transition probs
  #############
}
```

- Use the age-dependent probability matrix function in the code for simulating the Markov model and compute the cost-effectiveness outputs. Are they different? How?

4

A Markov model probability sensitivity analysis (PSA)

The formulation in the previous section can be extended to include uncertainty about one or more of the parameters. Briggs (2000) describe this for the current model by repeating the analytical solution of the model employing different values for the underlying parameters sampled from specified ranges and distributions. Sensitivity analyses can be performed one at a time (one-way) or for multiple parameters simultaneously (multi-way). This section presents a multi-way probabilistic sensitivity analysis.

Performing a PSA analysis can be done by inputting random draws from the unit costs and QALY distributions as inputs to the existing model function. In R, there are numerous ways of implementing a PSA. Following from the R code presented in the previous sections, we can wrap this model code in a function, e.g. called `ce_markov()`, which we can then repeatedly call with different parameter values. To this we will need to pass the starting conditions: population (`start_pop`), age (`init_age`) and number of cycles (`n_cycle`) (in our case, if not defined then age and number of cycles are assigned default values). We will also need the probability transition matrix (`p_matrix`), state cost and QALY matrices (`state_c_matrix`, `state_q_matrix`, `trans_c_matrix`).

In extension to the first analysis, the unit values have distributions rather than point values. To sample from a base R distribution the function name syntax is a short form version of the distribution name preceded by and `r` (for random or realisation). For example, to sample from a normal distribution then call `rnorm()`. We could sample all of the random numbers before running the model which would allow us to save them to use again and improve run time because this would only be performed once outside of the main loop. Alternatively, we can sample the random variables at runtime, within the Markov model function. This is arguably neater and if we wish to replicate a particular run then we can set the random seed beforehand with `set.seed()`. We will demonstrate how to implement a simple version when sampling at runtime.

```
ce_markov <- function(start_pop,
                      p_matrix,
                      state_c_matrix,
                      trans_c_matrix,
                      state_q_matrix,
                      n_cycles = 46,
                      init_age = 55,
                      s_names = NULL,
                      t_names = NULL) {

  n_states <- length(start_pop)
  n_treat <- dim(p_matrix)[3]

  pop <- array(data = NA,
               dim = c(n_states, n_cycles, n_treat),
               dimnames = list(state = s_names,
                               cycle = NULL,
                               treatment = t_names))
  trans <- array(data = NA,
                 dim = c(n_states, n_cycles, n_treat),
                 dimnames = list(state = s_names,
                                 cycle = NULL,
                                 treatment = t_names))

  for (i in 1:n_states) {
    pop[i, cycle = 1, ] <- start_pop[i]
  }
```

```
  cycle_empty_array <-
    array(NA,
          dim = c(n_treat, n_cycles),
          dimnames = list(treatment = t_names,
                          cycle = NULL))

  cycle_state_costs <- cycle_trans_costs <- cycle_empty_array
  cycle_costs <- cycle_QALYs <- cycle_empty_array
  LE <- LYs <- cycle_empty_array    # life-expectancy; life-years
  cycle_QALE <- cycle_empty_array   # qaly-adjusted life-years

  total_costs <- setNames(rep(NA, n_treat), t_names)
  total_QALYs <- setNames(rep(NA, n_treat), t_names)

  for (i in 1:n_treat) {

    age <- init_age

    for (j in 2:n_cycles) {

      # difference from point estimate case
      # pass in functions for random sample
      # rather than fixed values
      p_matrix <- p_matrix_cycle(p_matrix, age, j - 1,
                                 tpProg = tpProg(),
                                 tpDcm = tpDcm(),
                                 effect = effect())

      # Matrix multiplication
      pop[, cycle = j, treatment = i] <-
        pop[, cycle = j - 1, treatment = i] %*% p_matrix[, , treatment = i]

      trans[, cycle = j, treatment = i] <-
        pop[, cycle = j - 1, treatment = i] %*% (trans_c_matrix * p_matrix
            [, , treatment = i])

      age <- age + 1
    }

    cycle_state_costs[i, ] <-
      (state_c_matrix[treatment = i, ] %*% pop[, , treatment = i]) * 1/(1 +
          cDr)^(1:n_cycles - 1)

    cycle_trans_costs[i, ] <-
      (c(1,1,1) %*% trans[, , treatment = i]) * 1/(1 + cDr)^(1:n_cycles -
          2)

    cycle_costs[i, ] <- cycle_state_costs[i, ] + cycle_trans_costs[i, ]

    LE[i, ] <- c(1,1,0) %*% pop[, , treatment = i]

    LYs[i, ] <- LE[i, ] * 1/(1 + oDr)^(1:n_cycles - 1)

    cycle_QALE[i, ] <-
     state_q_matrix[treatment = i, ] %*%  pop[, , treatment = i]

    cycle_QALYs[i, ] <- cycle_QALE[i, ] * 1/(1 + oDr)^(1:n_cycles - 1)

    total_costs[i] <- sum(cycle_costs[treatment = i, -1])
    total_QALYs[i] <- sum(cycle_QALYs[treatment = i, -1])
  }

  list(pop = pop,
       cycle_costs = cycle_costs,
       cycle_QALYs = cycle_QALYs,
       total_costs = total_costs,
       total_QALYs = total_QALYs)
}
```

Because we will want to sample more than once, rather than just once at the start of the simulation, we can wrap the random sampling statements in a function so that they are called newly every time the Markov model is run. So, using the same names as we used for the point values in the previous analysis

```
# replace point values with functions to random sample

cAsymp <- function() rnorm(1, 500, 127.55)
cDeath <- function() rnorm(1, 1000, 255.11)
cDrug  <- function() rnorm(1, 1000, 102.04)
cProg  <- function() rnorm(1, 3000, 510.21)
effect <- function() rnorm(1, 0.5, 0.051)
tpDcm  <- function() rbeta(1, 29, 167)
tpProg <- function() rbeta(1, 15, 1506)
uAsymp <- function() rbeta(1, 69, 4)
```

- What is `uProg`, taken from the spreadsheet? What distribution does it have?

Similarly, rather than using fixed `state_c_matrix`, `trans_c_matrix` and `state_q_matrix`, if we define these as functions, we can sample newly their component values each time they are called. In practice, the code looks the same as previously but now the unit values are function calls so are followed by open and closed brackets.

```
# Define cost and QALYs as functions

state_c_matrix <- function() {
  matrix(c(cAsymp(), cProg(), 0,             # without drug
           cAsymp() + cDrug(), cProg(), 0), # with drug
           byrow = TRUE,
           nrow = n_treatments,
           dimnames = list(t_names,
                           s_names))
}
```

- Do the same modification for `state_q_matrix` and `trans_c_matrix`.

4.1 Run PSA analysis

To finally obtain the PSA output, loop over `ce_markov()` remembering to record the cost and QALYs outputs each time.

```
n_trials <- 500

costs <- matrix(NA, nrow = n_trials, ncol = n_treatments,
                dimnames = list(NULL, t_names))
qalys <- matrix(NA, nrow = n_trials, ncol = n_treatments,
                dimnames = list(NULL, t_names))

for (i in 1:n_trials) {
  ce_res <- ce_markov(start_pop = c(n_pop, 0, 0),
                      p_matrix,
                      state_c_matrix(),
                      trans_c_matrix(),
                      state_q_matrix())

  costs[i, ] <- ce_res$total_costs
  qalys[i, ] <- ce_res$total_QALYs
}
```

4.2 Plot results

```
# incremental costs and QALYs of with_drug vs to without_drug
c_incr_psa <- costs[, "with_drug"] - costs[, "without_drug"]
q_incr_psa <- qalys[, "with_drug"] - qalys[, "without_drug"]
```

- Plot the cost-effectiveness plane for the PSA output. You can base this on the simple case above. Indicate the ICER. How does this compare with the Excel spreadsheet?
- Use the `BCEA` package to create the standard cost-effectiveness plots.

5

Monte Carlo in BUGS

These are the first exercises to familiarise ourselves with using OpenBUGS. We will create more generic models before progressing to health economics models in particular. To begin, we will use the OpenBUGS GUI but later on we will run everything from inside of R.

5.1 "Coins" example

- Start OpenBUGS
- Load the file coins.odc from the appropriate folder — for example, this could look something like C:\bayes-hecourse\1_monte-carlo. This program will simulate throws of 10 balanced coins and record which give 8 or more heads.
- First run the program from a script. Load the file coins-script.odc and check that the path to the working directory is appropriate. Check the script makes sense. With this script window open, click on Info ⇒ Open Log. This opens a new window (containing the log of your BUGS session) — if you do not then you will not be able to see the results. Then Model ⇒ Script.
- Now try running using the interactive interface.
  - Open up the Model ⇒ Specification window.
  - Making sure that coins.odc is open, click on check model.
  - Then compile and gen inits.
  - Open up the Model ⇒ Update window and generate 1000 iterations.
  - Then open up the Inference ⇒ Samples window, type Y in the node window and then click set. This sets the monitor. Repeat for P8.
  - Type * in the node window to indicate all monitored quantities.
  - Click trace to generate traces of the simulated values.
  - Then do another 1000 updates.
  - stats then gives summary statistics.
- Find the probability that a clinical trial with 30 subjects, each with probability 0.7 of response, will show 15 responses or fewer. *(Hint: you could use the notation* step(15.5 - Y)*, or* 1 - step(Y - 15.5)*).*

5.2 Drug example

- Open the file drug-MC.odc and carry out a BUGS run for this model, obtaining the results shown in the lectures. You should be able to run it using the previous instructions (question 1) and the short list given in the lectures. Otherwise full details are given in *Running a model in* OpenBUGS of the "Hints on using OpenBUGS" chapter of this handout. If stuck, a script drug-MC-script.odc is available.
- Edit the model code to specify a Uniform(0, 1) prior on the response rate theta, and re-run the analysis. *(Note: the syntax for the uniform prior in* BUGS *is* dunif(a,b) *where* a *and* b *are the lower and upper bounds. The values of* a *and* b *can either be specified in the data file, or directly in the* BUGS *code (e.g.* a <- 1*), or just replace* a *and* b *by their values in the* dunif *statement).*
- Plot the predictive distribution for the number of successes.
- What is now the predictive probability that 15 or more patients will experience a positive response out of 20 new patients affected?

5.3 Simulating functions of random quantities

- Write a model for a variable with a normal distribution with mean 0 and standard deviation 1 (remember BUGS parameterises the normal in terms of precision = 1/Variance).
- Simulate 10000 values and plot their density.
- Simulate 10000 values of a variable Y with a normal distribution with mean 1 and standard deviation 2.
- For the same Y, create a new variable $Z = Y^3$, simulate 10000 values of Z, and find the mean and variance of Z, and the probability that Z is greater than 10. Are these results surprising?
- Plot the density of Z.

6

Markov Chain Monte Carlo

6.1 Understanding Gibbs sampling

The file `GibbsSampling.xls` is a spreadsheet which you can open using `MS Excel` (or any similar spreadsheet software, such as `LibreOffice Calc` or `OpenOffice Calc`) and contains a very simple example of Gibbs Sampling at work. The first worksheet sets up a simple bivariate Normal model:

$$\boldsymbol{y} = (y_1, y_2) \sim \text{Normal}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

with $\boldsymbol{\mu} = (\mu_1, \mu_2)$ and $\boldsymbol{\Sigma} = \begin{pmatrix} \sigma_1^2 & \sigma_1\sigma_2\rho \\ \sigma_1\sigma_2\rho & \sigma_2^2 \end{pmatrix}$. Given this set up, standard Normal theory says that the *full conditionals* are

- $y_2 \mid y_1 \sim \text{Normal}\left(\mu_2 + \frac{\sigma_2}{\sigma_1}\rho\left(y_1 - \mu_1\right), \left(1-\rho^2\right)\sigma_2^2\right)$, and
- $y_1 \mid y_2 \sim \text{Normal}\left(\mu_1 + \frac{\sigma_1}{\sigma_2}\rho\left(y_2 - \mu_2\right), \left(1-\rho^2\right)\sigma_2^1\right)$.

In order to familiarise yourself with the MCMC process:

a. Inspect cells `J4` and `K4`; the former sets the initial value of y_1 (which is stored in cell `B16`), while the latter simulates from the conditional distribution of y_2 given the current value for y_1. (NB: the notation `NORM.INV(rand(),mean,standard_dev)` instructs Excel to simulate a random draw from a Normal distribution with parameters `mean` and `standard_dev`).
b. Move to the second spreadsheet (named `10iters`). This shows the (y_1, y_2) plane with the first 10 simulated values; compare the graph with those in the spreadsheets `100iters`, `500iters` and `1000iters` (showing 100, 500 and 1000 iterations of the Gibbs sampler, respectively). Assess convergence on the basis of the graphs. Are 100 iterations enough?
c. Move to the spreadsheet named `Tr_y1`: this shows the traceplot for y_1 over 1000 simulations. Would you assess that convergence has been satisfactorly reached for this variable? Repeat this for the worksheet `Tr_y2`, which shows the traceplot for y_2.
d. Go back to the first spreadsheet and modify the initial values for y_1, y_2. Go to cell `B16` and type the Excel command `=D16` and to cell `B17` and type the command `=D17` (these will copy over two random initial values drawn from a Normal distribution with mean 0 and standard deviation 10). Inspect the other spreadsheets; how is convergence affected for (y_1, y_2)?
e. Go back to the first spreadsheet and modify the value for the correlation coefficient in cell `E8`, by typing `0.99`. Inspect the other spreadsheets; how is convergence affected for (y_1, y_2)?

6.2 MCMC in `OpenBUGS`

Consider again the drug example discussed in the lecture. The file `drug-MCMC.odc` contains the `BUGS` code to describe the model as well as data and initial values for a simple analysis. From `OpenBUGS`, open the file and, using the commands and actions discussed in the first practical, execute the following analyses.

a. – Click on `Model` and then `Specification` to open the `Specification Tool`.
   – `check` the model (which is contained in the code prefixed by the comment "`### Model description`", which describes in general the distributional assumptions);
   – `load` the data (contained in the code prefixed by the comment "`### DATA in separate list (better in most cases)`");
   – `compile` the model;

- load the initial values (contained in the code prefixed by the comment ### INITIAL VALUES).
- Click on Inference and then Samples to open the Sample Monitor Tool. Choose the nodes you think should be monitored and then click on Model and then Update to set up the MCMC run, selecting a suitable number of replications.
- Finally, produce summary statistics and graphical description of the results.

b. You can replicate the above analysis by using the code prefixed by the comment

```
### Data supplied with model description
### (only feasible for very few data points)
```

In this case, you do not need to load the data, as they are already given in the model code and thus are automatically loaded when the model is compiled.

c. Try and monitor the node y. Why do you think BUGS does not let you do this?

7

Cost-effectiveness analysis in R using MCMC and BCEA

7.1 MCMC in R/BUGS

Consider Laplace's analysis of birth data, which included $y = 241\,945$ females out of $n = 493\,527$ babies born in Paris between 1745 and 1770. Assume the following modelling assumptions:

$$y \sim \text{Binomial}(\theta, n)$$
$$\theta \sim \text{Uniform}(0, 1).$$

a. Write BUGS code to encode the modelling assumptions above and save it to a `.txt` file (you can choose whatever name and location you want. We assume you've chosen `ModelLaplace.txt`, which you have saved in the current directory).
b. Open R, which will be used to pre-process the data, call BUGS in background to perform the MCMC estimation and then post-process the results. At the terminal, type the following commands

```
> y <- 241945
> n <- 493527
> data <- list(y=y,n=n)
```

This defines the variables `y` and `n` into the R workspace and create a `list` (named `data`), which contains them.
(Notice that in R you can use the notation "<-" to indicate assignment to a variable, as well as the more straightforward notation "=". Thus the commands `a <- b` *and* `c = b` *create two variables* `a` *and* `c` *which both take the value associated with the variable* `b` *— irrespective of the sign used to define the assignment operator)*.
Now, set up some utility variables, such as the location of the BUGS model file, the parameters to be monitored and the initial values, by typing

```
> filein <- "PATH_TO_FILE/NAME_OF_FILE.txt"
> params <- "theta"
> inits_det <- list(list(theta=.1),list(theta=.9))
> inits_ran <- function(){list(theta=runif(1))}
```

The R workspace now contains a variable `filein` (a text string with the path to your model file, e.g. `"C:/bayes-hecourse/4_bcea/ModelLaplace.txt"`); a variable `params` (another text string containing the name of the variable you want to monitor); the two variables `inits_det` and `inits_ran` can be used to instruct BUGS which initial values to use. The former is a `list` containing two initial values for the node `theta` (they are arbitrarily set to 0.1 and 0.9, so that two chains can be run, starting from different points). The latter is an R `function`, which creates a list with a variable `theta`, which is assigned a random draw from a Uniform(0, 1) distribution.
c. Call BUGS from within R by using the following commands

```
> model <- bugs(data=data,inits=inits_det,parameters.to.save=params,model.
    file=filein,n.chains=2,n.iter=10000,n.burnin=4500,n.thin=1,DIC=TRUE)
```

— notice that you can either use the syntax `inits=inits_det` or `inits=inits_ran`. BUGS is called in background and executes the MCMC analysis. When it has completed (which should be extremely quick in this simple case), you will regain use of the R terminal.
The R object `model` contains many variables; typing

```
> names(model)
```

will print a list of them. Each can be accessed using the R "$" notation. For example, typing

```
> model$n.iter
```

will print the number of iterations used by BUGS.

d. Check the results by using the command

```
> print(model,digits=3,intervals=c(0.025,0.975))
```

This instructs R to give a tabular output reporting summary statistics (specifically reporting the 2.5% and 97.5% percentiles) from the posterior distribution(s) of the node(s) monitored, using 3 significant digits.
What can you say about convergence by just looking at the resulting tabular display that R will provide?

e. Attach the MCMC simulations to the R workspace by typing

```
> attach.bugs(model)
```

This makes all the elements of the object `model` available in your R session.
Plot a histogram of the posterior distribution for `theta` by typing

```
> hist(theta)
```

What can you say about the underlying probability of a female birth?

7.1.1 MCMC in R/JAGS

If you want to use JAGS instead of BUGS, there are not many difference with respect to the previous code/analysis. You can use these guidelines to perform the analyses in the practicals using JAGS.

The first thing to do is to load the package `R2jags`, which you do by typing in your R terminal the command `library(R2jags)`. You can now replicate steps a. and b. from the previous section to write your model and prepare the data and relevant variables. Notice, however, that while most of the time the BUGS code will apply with no problems to JAGS, there are a few differences that may prevent your model code from working. One such example is the way in which BUGS and JAGS manage truncation and censoring (see, for instance page 205 of BMHE).

The first change is that (unsurprisingly!) you need to call the function `jags`, instead of the function `bugs`. Thus, point c. above becomes

```
> model2 <- jags(data=data,inits=inits_det,parameters.to.save=params,model.
    file=filein,n.chains=2,n.iter=10000,n.burnin=4500,n.thin=1,DIC=TRUE)
```

This time, JAGS is called in background and performs the MCMC estimation. By default, `R2jags` shows a text bar with the progression through the required simulations; a running series of asterisks is printed and the counter is incremented while the iterations are generated.

The second and perhaps most important difference is in the nature of the objects created. If you type in your R terminal the command `names(model2)`, R will show you the names of the elements of the object `model2`

```
[1] "model"              "BUGSoutput"         "parameters.to.save"
[4] "model.file"         "n.iter"             "DIC"
```

You can access each of these elements using the syntax `object$element`, so for example R will respond to the command `model2$n.iter` by printing the value for the number of iterations that have been used (in this case 10000).

If, for instance, you had run the steps described in the previous section and had produced the BUGS object `model`, R reponse to a command `names(model)` would show different elements. In fact, the difference is that JAGS stores *these* elements in the object `model2$BUGSoutput`; thus typing `names(model2$BUGSOutput)` gives the following output

```
[1] "n.chains"         "n.iter"           "n.burnin"         "n.thin"
[5] "n.keep"           "n.sims"           "sims.array"       "sims.list"
[9] "sims.matrix"      "summary"          "mean"             "sd"
[13] "median"           "root.short"       "long.short"       "dimension.short"
[17] "indexes.short"    "last.values"      "program"          "model.file"
[21] "isDIC"            "DICbyR"           "pD"               "DIC"
```

which is basically the same as that produced by a call to BUGS.

The third basic difference is that if you want to make the results of your simulations available to the R workspace, you need to use the command `attach.jags(model2)` (or whatever the name of the object created with the call to the `jags` function).

7.2 Health economic evaluation in R using BCEA

Suppose the interest is in an infectious disease, e.g. influenza, for which a new vaccine has been produced. Under the current management of the disease some individuals treat the infection by taking over the counter (OTC) medications. Some subjects visit their GP and, depending on the severity of the infection, may receive treatment with antiviral drugs, which usually cures the infection. However, in some cases complications may occur. Minor complications will need a second GP visit after which the patients become more likely to receive antiviral treatment. Major complications are represented by pneumonia and can result in hospitalisation and possibly death. In this scenario, the costs generated by the management of the disease are represented by OTC medications, GP visits, the prescription of antiviral drugs, hospital episodes and indirect costs such as time off work. QALYs are used as a measure of clinical benefit. The focus is on the clinical and economic evaluation of the policy that makes the vaccine available to those who wish to use it ($t = 1$) against the null option ($t = 0$) under which the vaccine will remain unavailable.

a. The data "Vaccine" contains the results of a economic model, which are saved inside of the BCEA package. The BUGS model that generated these data is given in `vaccine.txt` but we don't go in to this component of the modelling any further here. A description of how the separate posterior values are combined to given `e` and `c` is given in the BCEA book. From your R terminal, load the vaccine data, create a BCEA object and examine the resulting object (we've called `he`) using the following commands

```
> data("Vaccine", package = "BCEA")
> he <- bcea(e, c, ref = 2, interventions = c("Status␣Quo", "Vaccination"))
> names(he)
```

Load BCEA by typing the command

```
> library(BCEA)
```

and compute the ICER using the elements `delta_e` and `delta_c`, which are included in the object `he` and contain the simulations from the posterior distributions of the variables (Δ_e, Δ_c) (i.e. the effectiveness and cost differential, respectively), obtained by running a suitable Bayesian model. *Hint: the command* `a = b / c` *instructs* R *to compute a variable* `a` *defined as the ratio between the variables* `b` *and* `c`*; similarly, the command* `d = mean(e)` *saves the mean of the values contained in a vector* `e` *to a new variable* `d`.

b. Assuming a fixed value of the willingness-to-pay threshold of $k = 30\,000$, compute the value of the EIB. *Hint: consider the definition of EIB and use the values for* `delta.e` *and* `delta.c`. *In* R*, the commands* `a = b + c` *and* `d = e - f` *define variables* `a` *and* `d` *as the sum and the difference of other variables, previously defined in the workspace.*

c. In the R terminal, execute the command

```
> ceplane.plot(he)
```

to produce the cost-effectiveness plane for the comparison of $t = 1$ vs $t = 0$, using a default value of $k = 25\,000$.
How can you interpret the resulting graph, in terms of economic evaluation?

d. In the R terminal, execute the command

```
> ceplane.plot(he, wtp=10000)
```

to produce the cost-effectiveness plane for the comparison of $t = 1$ vs $t = 0$, using a value of $k = 10\,000$.
How is the economic interpretation of the results modified, in comparison to point c?

e. In the R terminal, execute the command

```
> eib.plot(he, plot.cri=FALSE)
```

to obtain a graph showing the EIB as a function of the willingness-to-pay. (Notice that you need to add the option `plot.cri=FALSE`, which prevents BCEA from drawing a credible interval around the EIB, in this particular case. The reason is that the BCEA object `he` has been tampered with for the purpose of running this exercise. In general, you do not need to specify the option and by default BCEA will also show the credible interval).
How can you interpret this graph in terms of cost-effectiveness analysis?

f. In the R terminal, execute the command

```
> contour(he)
```

to produce the cost-effectiveness plane for the comparison of $t = 1$ vs $t = 0$ together with a contour plot of the bivariate distribution for (Δ_e, Δ_c). The graph also reports the proportion of simulated points in each quadrant.

Comment on the cost-effectiveness results. What is the probability that vaccination ($t = 1$) is dominated by the status quo ($t = 0$)?

8

Bayesian Markov modelling

In these exercises we apply a Bayesian perspective to the Markov models we met in earlier exercises.

8.1 Revision of beta+binomial conjugate Bayesian inference

This question can be omitted if you are confident about the material on conjugate Bayesian inference from lecture 2. However, it may help in understanding later questions on Markov models constructed from Dirichlet+Multinomial models, which are a generalisation of Beta+Binomial models.

a. Write down the likelihood function for $r = 15$ people falling asleep during the lecture hour, given $n = 40$ were awake at the start. (Assume these are independent events and everyone has the same probability, π, of falling asleep).
b. Assuming a Beta(1, 1) prior distribution for π, write down the posterior distribution, given the numbers in each state in the lecture hour.
c. Simulate a vector of 1000 samples from this posterior distribution using Monte Carlo simulation in `R`
d. Assuming a time-homogeneous 2-state Markov chain with sleep as an absorbing state (people don't wake up again), obtain a vector of 1000 samples from the predictive distribution for the number awake after two hours.

8.2 Dirichlet / Multinomial conjugate Bayesian inference

In the asthma example from the lecture, suppose that the model was simplified so that the two "exacerbation" states (Hex and Pex, hospital or primary-care managed exacerbations) are merged and considered as a single state (Ex: exacerbation).

a. Using data from the SFC treatment arm, write down the likelihood function for the transition probabilities out of state STW, given that $\boldsymbol{r} = (210, 60, 1, 1)$ people end up in states (STW, UTW, Ex, and TF) one week after being in state STW.
b. Assume a Dirichlet(1, 1, 1, 1) prior distribution for $\boldsymbol{\pi}_1$, the vector of probabilities for moving from state STW to the other states in one week. Write down the posterior distribution of $\boldsymbol{\pi}_1$, given data.
c. Simulate a vector of 1000 samples from this posterior distribution using Monte Carlo simulation in `R`

8.3 Markov modelling in `R`

In this section, we will construct a Markov model from the asthma data given in the lecture, using the simplified four-state representation (STW, UTW, Ex, and TF).

The required `R` code gets more complex as this question goes on. If you would like to practice constructing this code from scratch, then use the code given in the lecture notes as the basis for the answer. Otherwise, just step through the `R` code given in the solutions to make sure it makes sense. The solutions with explanations are in `Solutions.pdf`, with code that you can copy and run in `Solutions.R`.

a, Suppose that before this study, we have observed people visiting state Ex on about 100 occasions, but only once did somebody stay in that state for a period of more than one week. Also suppose we are unsure about what happens to people in the week following a period in the Ex state. Construct a Dirichlet prior for the transition probabilities out of state Ex based on this information.

b. Suppose we observed the following transition count data

	to state			
from state	STW	UTW	Ex	TF
STW	210	60	1	1
UTW	88	641	4	13
Ex	1	0	0	1

Extend the code from section 8.2 to generate a sample from the posterior distribution of the full transition probability matrix, using the prior from part (a) for the transition probabilities from state Ex, and uniform Dirichlet priors for the transition probabilities out of STW and UTW, and noting that TF is an absorbing state. Thus estimate the posterior mean transition probability matrix.

c. Given these transition probabilities, generate a sample from the posterior distribution of the probabilities of occupying each state for each of cycles 1 to 12 of a Markov model, assuming everyone starts in state STW.

d. Extend the code to draw from the posterior distribution of the (undiscounted) expected costs and health effects over 12 weeks, assuming costs of 7.96, 7.96, and 1000 for STW, UTW and Ex respectively, and a utility of 1 in STW and 0 otherwise.

e. Plot the joint distribution of costs and effects for the SFC group as a scatterplot in the cost-effectiveness plane.

f. If you have time, repeat the above analysis for the FP group. Thus compute and plot the incremental costs and effects, and compare with the cost-effectiveness scatterplot given in the lecture. Use costs of 2.38 for STW, UTW, and 1000 for Ex, and the same utilities as before.

A

Appendix: Hints on using OpenBUGS

A.1 Running a model in OpenBUGS

1. Start OpenBUGS by double clicking on the OpenBUGS icon (or double click on the file OpenBUGS.exe, typically in somewhere like the OpenBUGS\OpenBUGS322 directory in C:\Program Files (x86)).
2. Open the file containing model code as follows:
   - Point to File on the tool bar and click once with *left mouse button* (LMB);
   - Highlight Open option and click once with LMB;
   - Select appropriate directory and double click on file to open. Files for OpenBUGS input are sometimes in .txt plain text format rather than the default .odc "compound document" format — so you may need to select "Files of type" .txt.
3. Select the Model menu as follows:
   - Point to Model on the tool bar and click once with LMB.
   - Highlight Specification option and click once with LMB.
4. Focus the window containing the model code by clicking the LMB once anywhere in the window — the top panel of the window should then become highlighted in blue to indicate that the window is currently in focus.
5. Highlight the word model at the beginning of the code by dragging the mouse over the word whilst holding down the LMB.
6. Check the model syntax by moving the mouse over the check model box in the Specification Tool window and clicking once with the LMB.
   - A message saying model is syntactically correct should appear in the bottom left of the OpenBUGS program window. Any error messages will appear in the same place if there is a syntax error.
7. Open the data, if there are observed data in the model. The data can either be stored in a separate file, in which case open this file (or multiple files), or they may be stored in the same file as the model code.
8. Load the data as follows:
   - Highlight the word list at the beginning of the data file.
   - Click once with the LMB on the load data box in the Specification Tool window.
   - A message saying data loaded should appear in the bottom left of the OpenBUGS program window.
9. Select number of chains (sets of samples to simulate) by typing the number of chains required in the white box in the Specification Tool window.
   - The default is 1, but we will typically use 2 or more. Running more than one chain, starting from different initial values, makes convergence checking easier (see later).
10. Compile the model by clicking once with the LMB on the compile box in the Specification Tool window.
    - A message saying model compiled should appear in the bottom left of the OpenBUGS program window.
11. Open the initial values files. The initial values can either be stored in separate file(s), in which case open these files, or they may be stored in the same file as the model code.
12. Load any initial values as follows:
    - Highlight the word list at the beginning of the first set of initial values.
    - Click once with LMB on the load inits box in the Specification Tool window.
    - A message saying initial values loaded: model contains uninitialized nodes (try running gen inits or loading more files) should appear in the bottom left of the OpenBUGS program window.
    - Repeat process for the second set of initial values, if running two chains.

- A message saying `initial values loaded: model initialized` should now appear in the bottom left of the OpenBUGS program window.
- Sometimes we can get away with not supplying any initial values ourselves, and we can just click `gen inits` to let BUGS generate these automatically. Though this won't generally work if any of the priors are vague — see lecture 3. Also it is handy to start different chains at widely dispersed initial values to assess convergence.

13. Close the Specification Tool window by clicking once with LMB on the X button in top right corner of window.
14. You are now ready to start running the simulation:
    - Before doing so, you may want to set some monitors to store the sampled values for selected parameters (see section **Monitoring parameter values** below).
    - To run the simulation, select the Update option from the Model menu.
    - Type the number of updates (iterations of the simulation) you require in the appropriate white box (default is 1000).
    - Click once on the `update` box:
    - The program will now start simulating values for each parameter in the model.
    - This may take a few seconds — the box marked `iteration` will tell you how many updates have currently been completed. The number of times this value is revised depends on the value you have set for `refresh` (see white box above `iteration`). The default is every 100 iterations. If the model is very fast, you should increase this to, e.g. 1000 or 10000, then the model will run even faster since OpenBUGS is not unnecessarily redrawing the screen hundreds of times in a split second. If the model is very slow, you may like to decrease it to, e.g. 10 or 1 so that OpenBUGS does not appear to "freeze" during sampling.
15. When the updates are finished, the message updates took `***` s will appear in the bottom left of the OpenBUGS program window (where *** is the number of seconds taking to complete the simulation).
16. If you set monitors for any parameters you can now check convergence and view graphical and numerical summaries of the samples (see below).
17. To save any files created during your OpenBUGS run, focus the window containing the information you want to save, and select the Save As option from the File menu.
18. To quit OpenBUGS, select the Exit option from the File menu.

A.2 Monitoring parameter values

In order to check convergence and obtain posterior summaries of the model parameters, you first need to set *monitors* for each parameter of interest. This tells OpenBUGS to store the values sampled for those parameters; otherwise, OpenBUGS automatically discards the simulated values. There are two types of monitors in OpenBUGS:

1. *Sample* monitors
    - Setting a sample monitor tells OpenBUGS to store *every* value it simulates for that parameter.
    - You will need to set sample monitors if you want to view trace plots of the samples to check convergence (see section **Checking convergence** below) and if you want to obtain posterior quantiles, for example, the posterior 95% Bayesian credible interval for that parameter.
    - To set a sample monitor:
        - Select Samples from the Inference menu.
        - Type the name of the parameter to be monitored in the white box marked `node`.
        - Click once with the LMB on the box marked `set`
        - Repeat for each parameter to be monitored.
2. *Summary* monitors
    - Setting a summary monitor tells OpenBUGS to store the running mean and standard deviation for the parameter.
    - The values saved contain less information than saving each individual sample in the simulation, but require much less storage. This is an important consideration when running long simulations (1000's of iterations) and storing values for many variables.
    - To set a summary monitor:
    - Select Summary from the Inference menu.
    - Type the name of the parameter to be monitored in the white box marked `node`.
    - Click once with the LMB on the box marked `set`
    - Repeat for each parameter to be monitored.

Note: you should not set a summary monitor until you are happy that convergence has been reached (see next section), since it is not possible to discard any of the pre-convergence ('burn-in') values from the running mean summary once it is set.

A.3 Checking convergence

Checking convergence requires considerable care. It is very difficult to say conclusively that a chain (simulation) has converged, only to diagnose when it definitely hasn't converged.

The following are practical guidelines for assessing convergence:

- For models with many parameters, it is inpractical to check convergence for every parameter, so just chose a random selection of relevant parameters to monitor.
  - For example, rather than checking convergence for every element of a vector of random effects, just chose a random subset (say, the first 5 or 10).
- Examine trace plots of the sample values versus iteration to look for evidence of when the simulation appears to have stabilised:
  - To obtain 'live' trace plots for a parameter:
    - Select Samples from the Inference menu.
    - Type the name of the parameter in the white box marked node.
    - Click once with the LMB on the box marked `trace`: an empty graphics window will appear on screen.
    - Repeat for each parameter required.
    - Once you start running the simulations (using the Update Tool, trace plots for these parameters will appear 'live' in the graphics windows.
  - To obtain a trace plot showing the full history of the samples *for any parameter for which you have previously set a sample monitor and carried out some updates:*
    - Select Samples from the Inference menu.
    - Type the name of the parameter in the white box marked node (or select name from pull down list).
    - Click once with the LMB on the box marked `history`: a graphics window showing the sample trace will appear.
    - Repeat for each parameter required.
- In Figure A.3(a) the chain converges by about 250 iterations.
  Running the simulation for longer (5000 iterations) and discarding the first 1000 as a "burn-in" produces the history plot in (b). This looks like a "fat hairy caterpillar", the typical appearance of a chain which has converged to the target distribution, and can be treated as a sequence of independent samples from that distribution.
- Figure A.3, panel(c), is the typical appearance of a chain which *has* converged to the target posterior distribution, but is slow to *mix* around that distribution. That is, the successive draws from the distribution are highly *autocorrelated*. In this case we should run the chain for longer (as in panel (d)) to get sufficiently precise summaries of the posterior — see the next section for more advice.
- If you are running more than 1 chain simultaneously, the trace and history plots will show each chain in a different colour. This is shown in Figure A.3, panel(e). We can be reasonably confident that convergence has been achieved when all the chains appear to be overlapping one another.

A.4 How many iterations after convergence?

Once you are happy that convergence has been achieved, you will need to run the simulation for a further number of iterations to obtain samples that can be used for posterior inference. The more samples you save, the more accurate will be your posterior estimates. Therefore the number of samples to run depends on how many significant figures, for example, you need in your results.

To assess the accuracy of the posterior mean for each parameter, you can simply look at the Monte Carlo standard error, which is provided in the summaries given by `stats` (see the next section). This is an estimate of the difference between the mean of the sampled values (which we are using as our estimate of the posterior mean for each parameter) and the true posterior mean.

To assess the accuracy of the whole distribution, you can compare the Monte Carlo standard error $\mathrm{SE}(\hat{\mu})$ to the sample standard deviation $\hat{\sigma}$ (also reported in the summary statistics table given by `stats`). There is some theory which suggests that for the reported 95% posterior quantiles to have about 94.5%

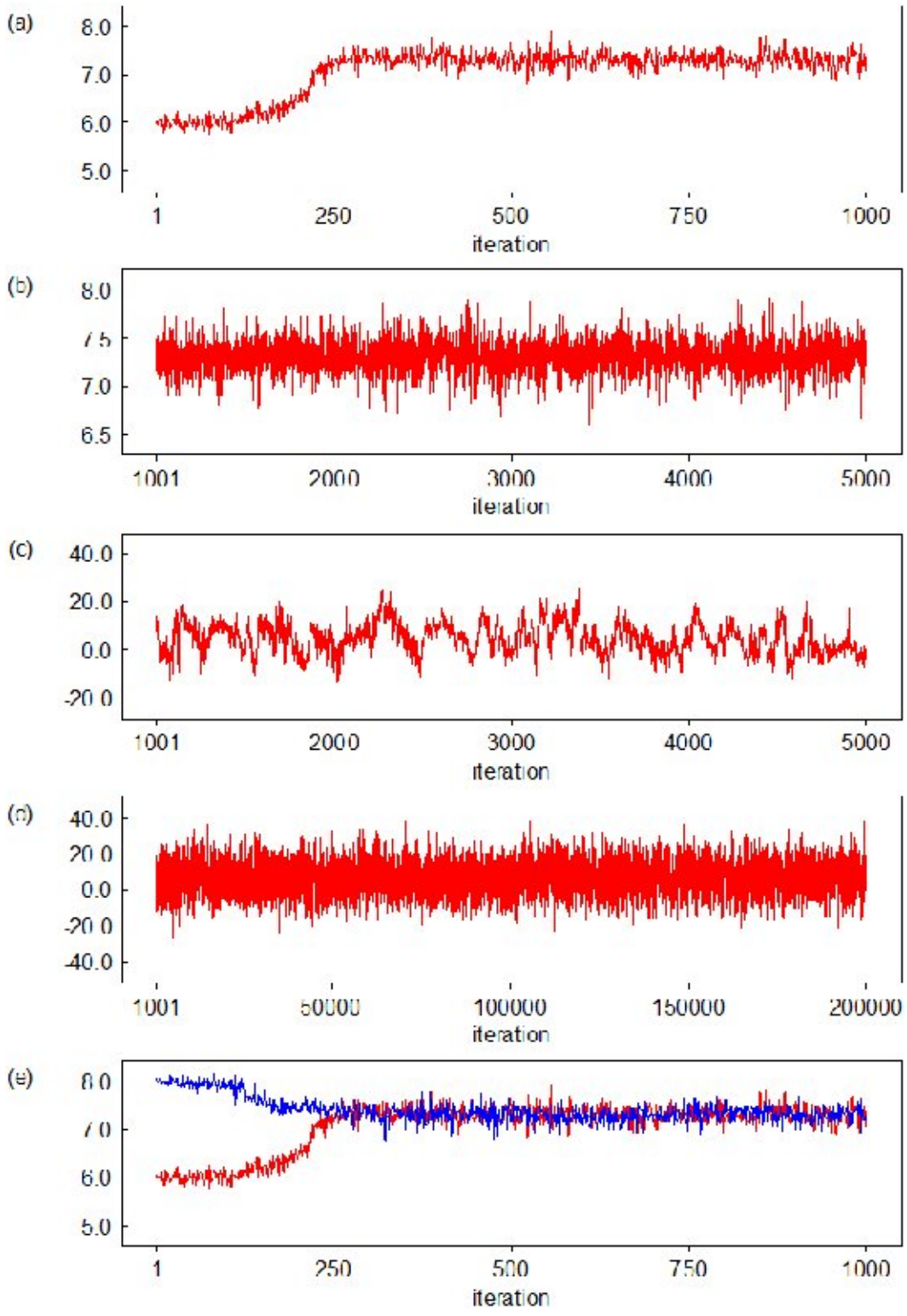


Fig. A.1. Markov Chain Monte Carlo convergence assessment.

to 95.5% true coverage, SE($\hat{\mu}$) should be 1% or less of $\hat{\sigma}$. Equivalently the *effective sample size* of the chain[1] should be greater than about 4000.

[1] The Monte Carlo standard error is higher in chains which are autocorrelated (i.e. look less like independent samples from a posterior, and are slower to explore the posterior distribution, so give less accurate estimates of the posterior mean). The "effective sample size" n_{eff} denotes that a chain of $n > n_{eff}$ autocorrelated samples as much information as n_{eff} independent samples. There are several ways of calculating n_{eff} — three different methods are used in *Bayesian Methods in Health Economics*, *The BUGS Book* and R2OpenBUGS!

Alternatively you can just take an informal approach, and run a sufficient number of samples to ensure that the posterior summaries of interest don't appear to change within the desired number of significant figures.

A.5 Obtaining summary statistics of the posterior distribution

To obtain summaries of the monitored samples, to be used for posterior inference:

- Select Samples from the Inference menu.
- Type the name of the parameter to be summarised in the white box marked node (or select name from the pull down list, or type * to select all monitored parameters).
- Type the iteration number which you want to start your summary from in the white box marked beg: this allows the pre-convergence 'burn-in' samples to be discarded.
- Click once with LMB on box marked stats: a table reporting various summary statistics based on the sampled values of the selected parameter will appear.

A.6 Plotting summaries of the posterior

OpenBUGS includes options for producing various plots of posterior summary statistics. The plot options include:

1. **Kernel density plot:** plots an estimate of the shape of the (univariate) marginal posterior distribution of a parameter see on-line manual for further details (select OpenBUGS User Manual from the Manuals menu in OpenBUGS and then take a look at the subsection on *Density* plots in the *OpenBUGS Graphics* section).
2. **Box plots, caterpillar plots or density strips:** these plots show a side-by-side comparison of the posterior distributions of each element of a vector of parameters summarised either as a point estimate and 95% interval (caterpillar plot), by the mean, interquartile range and 95% interval (box plot), or a representation of the whole distribution using varying shading (density strips). This is often used for random effects parameters. For example, suppose you have a vector of random effects called p in your model:
   - You should have already set a samples monitor on the appropriate vector (p) and carried out a suitable number of updates.
   - Then select Compare from the Inference menu.
   - Type the name of the parameter vector to be plotted (in this case p) in the white box marked node.
   - If you want to discard any pre-convergence burn-in samples before plotting, type the appropriate iteration number in the white box marked beg.
   - Click once with LMB on the button marked either box plot, caterpillar or density strips as required.
3. Model fit: this option produces a 'time series' type plot and is suitable for plotting an ordered sequence of parameter estimates against corresponding values of a known variable, e.g. plotting posterior estimates of the fitted values of a growth curve against time. For example, in the rats model from the OpenBUGS examples Vol I, you could use this option to produce a plot of the vector of 5 fitted values for the weight of each rat (mu[i,]) against age (x), as follows:
   - You should have already set a samples monitor on the appropriate vector (i.e. mu, the mean of the normal distribution assumed for the responses, Y) and carried out an appropriate number of updates.
   - Then select Compare from the Inference menu.
   - Type the name of the (stochastic) parameter vector to be plotted on the vertical axis in the white box marked node (e.g. mu[1,] to produce the plot for rat 1).
   - Type the name of the known (i.e. not stochastic) variable to be plotted on the horizontal axis in the white box marked axis (in this case, x, the name of the vector of ages at which each rat was measured).
   - An optional argument is to type the name of another known (i.e. not stochastic) variable in the white box marked other (for example, Y[1,] — this would plot the observed measurements for rat 1 as well as the fitted values mu[1,]).
   - If you want to discard any pre-convergence burn-in samples before plotting, type the appropriate iteration number in the white box marked beg.

- Click once with LMB on the button marked `model fit`. The resulting plot shows the posterior median (solid red line) and posterior 95% interval (dashed blue line) for the values of node (in this case, the fitted values `mu[1,]` for rat 1) against the values of the variable specified in the axis box (in this case, `x`, the age of the rat at each measurement). The black dots show the values of the variable specified in the other box (in this case, `Y[1,]`, the observed weights for rat 1).

4. There are various options for customising all these plots (e.g. changing the order in which the elements of the vector are plotted, switching the x and y axis, etc.). To access these options, click on the window containing the plot to focus it, then place the mouse somewhere in the plot window and click once with the *right* mouse button. A menu will appear and you should select the Properties option. This will open another menu called Plot Properties which provides options for editing plot margins, axis labels and fonts etc. (these are generic options for all OpenBUGS plots), plus some special options specific only to certain plots (click on the `Special` button at the bottom of the Plot Properties menu). See the on-line manual for further details (select OpenBUGS User Manual from the Manuals menu in OpenBUGS, then go to `OpenBUGS Graphics`, or `The Inference Menu` then `Compare`).

A.7 Some notes on the BUGS language

A.7.1 Basic syntax

- `<-` represents logical dependence, e.g. `m <- a + b*x`
- `~` represents stochastic dependence, e.g. `r ~ dunif(a,b)`
- Can use arrays and loops

```
for (i in 1:n){
  r[i] ~ dbin(p[i],n[i])
  p[i] ~ dunif(0,1)
}
```

- Some functions can appear on left-hand-side of an expression, e.g.

```
logit(p[i])<- a + b*x[i]
log(m[i]) <- c + d*y[i]
```

- `mean(p[])` to take mean of whole array, or `mean(p[m:n])` to take mean of elements `m` to `n`. Also for `sum(p[])`.
- `dnorm(0,1)I(0,)` means the prior will be restricted to the range $(0, \infty)$.

A.7.2 Functions in the BUGS language

- `p <- step(x - 0.7)` $= 1$ if $x \geq 0.7$, 0 otherwise. Hence monitoring `p` and recording its mean will give the probability that $x \geq 0.7$.
- `p <- equals(x, 0.7)` $= 1$ if $x = 0.7$, 0 otherwise.
- `tau <- 1/pow(s,2)` sets $\tau = 1/s^2$.
- `s <- 1/ sqrt(tau)` sets $s = 1/\tau$.
- `p[i,k] <- inprod(pi[], Lambda[i,,k])` sets $p_{ik} = \sum_j \pi_j \Lambda_{ijk}$
- Many other mathematical functions: see Functions under the Help menu in OpenBUGS for full syntax.

A.7.3 Some common Distributions

Expression	Distribution	Usage
`dbin`	binomial	`r ~ dbin(p,n)`
`dnorm`	normal	`x ~ dnorm(mu,tau)`
`dpois`	Poisson	`r ~ dpois(lambda)`
`dunif`	uniform	`x ~ dunif(a,b)`
`dgamma`	gamma	`x ~ dgamma(a,b)`

NB. The normal is parameterised in terms of its mean and *precision* = 1/ variance = $1/\text{sd}^2$.

See Distributions under the Help menu in OpenBUGS for a full list of supported distributions and their parameterisations.

Functions cannot be used as arguments in distributions (you need to create new nodes).

A.7.4 The OpenBUGS data formats

OpenBUGS accepts data files in:

1. Rectangular format

```
n[] r[]
47 0
148 18
...
360 24
END
```

2. R / S-Plus-like 'list' format:

```
list(N=12,n = c(47,148,119,810,211,196,
148,215,207,97,256,360),
r = c(0,18,8,46,8,13,9,31,14,8,29,24))
```

The more flexible 'list' format is recommended, since data often consist of mixtures of scalars and vectors/arrays, or vectors/arrays of different lengths.

A.8 Related software

- **WinBUGS** (1.4.3), the predecessor of OpenBUGS, is stable and still very widely used. It has the same Windows interface as OpenBUGS and works essentially the same way. There are a few more functions, distributions and MCMC sampling methods in OpenBUGS. OpenBUGS runs on Linux (through a plain text interface or through R) but WinBUGS does not.
- **JAGS** (http://mcmc-jags.sourceforge.net) was developed independently of Win/OpenBUGS. It is plain text-based with no graphical user interface, though works natively on Linux, Unix and Mac as well as Windows.
- **Stan** (http://mc-stan.org) is the newest BUGS-like program, and can perform Bayesian analyses of arbitrary structure and complexity just like WinBUGS, OpenBUGS and JAGS. It works using a modelling language superficially similar to BUGS but different in many subtle ways, and the sampling methods it uses under the surface are fundamentally different and often more efficient. It has an active and growing development team and user community.

A.9 Calling OpenBUGS from other software

OpenBUGS and the programs mentioned in the previous section can all be controlled from other software. We will concentrate on the R interfaces, but there are interfaces for many other programs (including Excel, SAS, Matlab and Stata, see http://www.mrc-bsu.cam.ac.uk/bugs).

A.9.1 Calling BUGS in batch mode

R2OpenBUGS is used to run an entire OpenBUGS simulation from R in a single R function call, and return the resulting samples and summaries as R objects. R2WinBUGS and R2jags have a very similar syntax, and are used to control WinBUGS and JAGS respectively in a similar "batch" mode.

A.9.2 Calling BUGS interactively

BRugs controls OpenBUGS from R in a more sophisticated way, by calling an embedded OpenBUGS computation library. Therefore, different steps of the analysis (such as checking model files, loading data, calculating summary statistics) can be performed by different R functions *interactively*. R2OpenBUGS, on the other hand, works by opening the OpenBUGS Windows program in the background, and therefore it needs to runs an entire OpenBUGS session as a single function call.

rjags and rstan control JAGS and Stan in a similar "embedded" manner, and are developed by the authors of JAGS and Stan in parallel with JAGS and Stan, respectively.

See the manuals of these packages for further details.