Practical 1. Monte Carlo in BUGS - SOLUTIONS

Lecture 1 PDF version

1. “Coins” example (see lectures)

The first thing we need to do is to run the programme from the script provided in the file `coins-script.odc`. The commands in the script make automatic the process of pointing-and-clicking that we would otherwise need to perform the analysis from `BUGS`. We need to be careful in telling `BUGS` what the “working directory” is. In the script, the default path is set as `c:/bayes-hecourse/1_monte-carlo`, which grants a couple of comments.

Firstly, you may not have saved the script file onto this directory (which in fact may not even exist in your computer!). So you need to make sure you update this to the actual path to the folder in which the file `coins-script.odc` is stored, or else `BUGS` will complain that it can’t find the file. Secondly, the default text uses `Unix` notation, where folders are separated using a “`/`” symbol. Under other operating systems (notably `MS Windows`), this is no longer valid and paths are indicated using a “`\`” symbol. You then need to be careful in providing the correct string. For instance, suppose your file under `MS Windows` is saved in the directory `N:\DesktopSetting\Desktop\STAT0019`, then you would need to copy and paste this string into the model script.

In order to run the script, you need to click anywhere on the file `coins-script.odc` (assuming you have opened it in `BUGS`) and then click on the menu `Model`→ `Script`

NB: we use the notation `Command1` → `Command2` to indicate that you need to click on the menu labelled as `Command1` and then on the sub-menu labelled as `Command2`).

Unlike the older version (`WinBUGS`), `OpenBUGS` does not open the log file automatically. Thus, if there is an error, only a message in the bottom-left corner of the window will appear (but it may not be very noticeable). To open the log file you can click on `Info`→ `Open Log`. If you do this, before running the script, remember to click back on the part of the window occupied by the script file and then click on `Model`→ `Script`. In this case, you will see on the log file messages from `OpenBUGS` informing you of what is happening. For instance, if all works, these messages will be printed.

```
model is syntactically correct
model compiled
initial values generated, model initialized
model is updating
1000 updates took 0 s
```

In addition, two more windows will automatically open, showing the output of the `BUGS`procedure. The first one shows the “Node statistics, i.e. the summary of the simulations performed for the node monitored (you can check that the script instructs `BUGS`to monitor the nodes `Y` and `P8` with the commands

```
samplesSet(Y)
samplesSet(P8)
```

The second window shows the graphs of the posterior densities for the nodes monitored. In this case, you will see the histograms of `P8` and `Y`. The latter represents the predictive distribution of Y, the number of coin tosses showing up “heads”. In line with the summary statistics, this distribution has a mean of approximately 5 and most of the probability mass (in fact, 95%) is between 2 and 8. As for P_8, which represents the probability of observing 8 or more “heads”, the histogram has mostly 0s, to indicate that in all the simulations it is most likely that this event does *not* happen, than it does. Again, this is in line with the summary statistics, showing that the sample mean computed from the simulations

$$\mathrm{E}[P_8] = \frac{1}{S}\sum_{s=1}^{S} P_8$$

is just above 5%. Notice that because of how the node `P8` is defined in the model code (see Lecture notes), it either takes value 0 (if the event is not true) or 1 (otherwise). Thus the mean effectively represents the estimated probability that the event (i.e. at least 8 “heads”) is true.

We can also run the model using the point-and-click facility of `BUGS` (this can be done by simply following all the steps in the practical question). The output will be identical to the one just described.

Finally, we can modify the original model code to encode the assumption that the coin is actually unbalanced (with a probaility of "heads" equal to 0.7). To do so, we simply need to modify the model code as follows.

```
model {
  Y ~ dbin(0.7, 30)
  P15 <- step(15.5 - Y)
}
```

and then re-run the `BUGS`procedure. Of course you can either overwrite the original file and then run the script, or save the new code to another file, update the command `modelCheck` to provide the new file name, or simply use the point-and-click procedure (this latter option does not require you to save the file).

The predictive distribution for Y in this new setting is centered around 20, with most of the mass between 16 and 26, while the probability of observing at most 15 "heads" is estimated to be around 1.7%.

2. "Drug" example (see lectures)

To run the model, you can again use the script provided or simply point-and-click to run the model code file, also provided. As discussed in the class, the model code instructs `BUGS`on the assumptions underlying the model.

```
model{
  theta ~ dbeta(9.2, 13.8)
  y ~ dbin(theta, 20)
  P.crit <- step(y - 14.5)
}
```

The structure of the model is pretty much the same as in the previous example — except that in this case, we are not willing to assume a "fixed" and known value for the probability of the event of interest (e.g. "heads" in the previous example, or that the drug is successful, in the current one). So, instead of assigning it a value (e.g. 0.5 or 0.7), we describe our knowledge using a distribution. Given that we believe the "true" success rate to be between 0.2 and 0.6, we can express this using a Beta distribution with parameters $\alpha = 9.2$ and $\beta = 13.8$. As discussed in class, by the mathematical properties of the Beta distribution, this implies that we believe

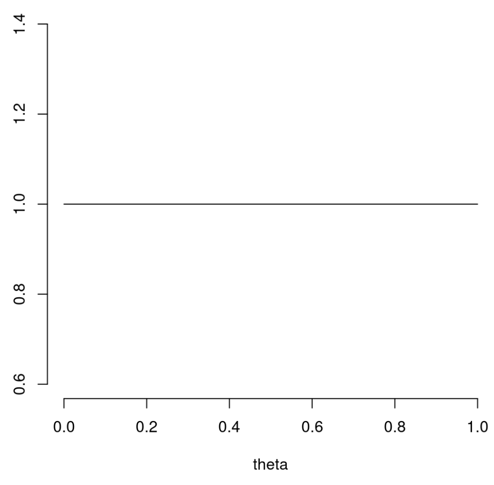
$$\mathrm{E}[\theta] = \frac{\alpha}{\alpha+\beta} = \frac{9.2}{23} = 0.4$$

and

$$\mathrm{Var}[\theta] = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)} = 0.01$$

(which implies $\mathrm{sd}[\theta] = \sqrt{0.001} = 0.1$, which in turn implies that, *approximately*, 95% of the mass for θ is included in $0.4 \pm 2(0.1) = [0.2; 0.6]$, as required).

In the next part of the exercise, you are required to modify the prior distribution for θ to a Uniform(0,1). This effectively means that you are not implying any specific prior knowledge on the "true" success rate, apart from the fact that it can take a value between 0 and 1. Graphically, a Uniform distribution in (0,1) looks like in the following graph.



This prior distribution is "non informative" in the sense that it does not provide much information or clue as to what value of the probability of success is more likely to generate the data. Consequently, it is not surprising that the resulting predictive distribution for Y is spread over the entire range of possible values (with a mean of around 10 and a 95% interval covering 0 to

20 — i.e. the whole range). Basically, if all you're prepared to say before seeing any data is that the true success rate is something between 0 and 1, then the prediction of the number of successes cannot be anything else than anything between 0 (no successes) and 20 (all patients are cured)…

3. Simulating functions of random quantities

One of the main features of Bayesian analysis using simulation methods (such as those underpinning `BUGS`) is that it is possible to obtain, almost as a simple by-product of the estimation procedure, simulations for any function of the main model parameters. We shall see later that this is very helpful when using complex models, e.g. for economic evaluation.

In this case, we need to write a model for a variable $y \sim \text{Normal}(\text{mean} = 0,\ \text{sd} = 1)$. As suggested in the practical exercise, we need to be careful and remember that `BUGS`parameterise the Normal distribution in terms of mean and **precision** (=1/variance). If we are imposing a sd of 1, then this is irrelevant because if $\sigma = 1$, then $\text{variance} = \text{precision} = \frac{1}{\text{variance}} = 1$. The model can be written in `BUGS`language as

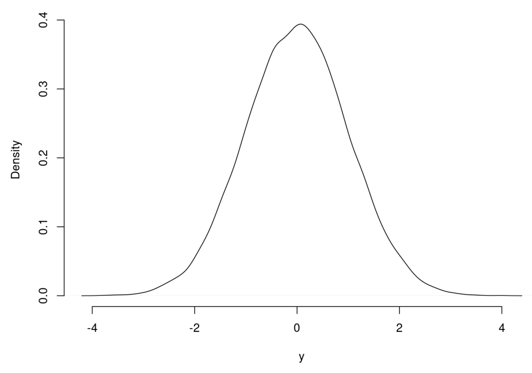
```
model {
  y ~ dnorm(0, 1)
}
```

and run using the same procedure described above (for such a simple model, we can just use the point-and-click approach).

This produces simulations from a Normal variable with 0 mean and unit variance. The summary statistics of this distribution are as follows (your output may vary, due to pseudo-random variation).

```
    mean       sd      2.5%      25%      50%      75%     97.5%
-0.00153  1.00520 -1.95402 -0.67965  0.00295  0.66825  1.98802
```

based on a sample of 10000 simulations. The resulting density looks like the following (you can produce the graph in `BUGS`using the `density` button in the `Sample monitor tool`).



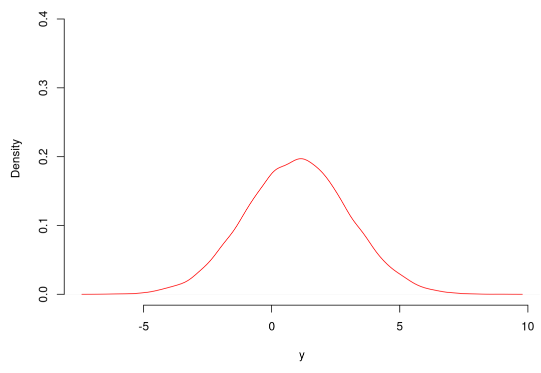
If we want to assume that $y \sim \text{Normal}(\text{mean} = 1,\ \text{sd=}2)$, then this implies that $\text{precision} = 1/2^2 = 0.25$. So we need to modify the model code to

```
model {
  y ~ dnorm(1, 0.25)
}
```

We can run this new model and obtain summaries and graphs for the predictive distribution of y (notice that the term "predictive" here highlights the fact that we have not actually observed y — we might in the future, but not just yet. In fact the most appropriate term here would be "*prior* predictive" distribution, again to highlight the fact that this is only based on our prior knowledge about the parameters, i.e. mean and standard deviation).

The output of the analysis is as follows (again, if you see slight differences, these would be down to simulation error).

```
  mean     sd   2.5%    25%    50%    75%  97.5%
 0.997  2.010 -2.909 -0.359  1.006  2.336  4.975
```

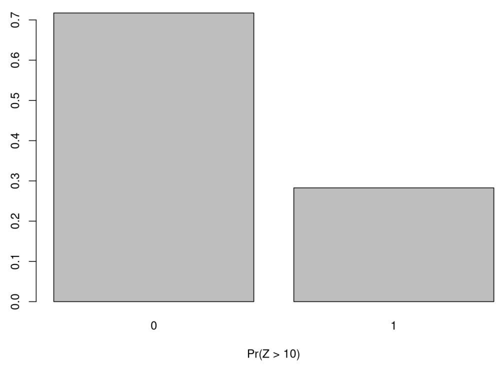


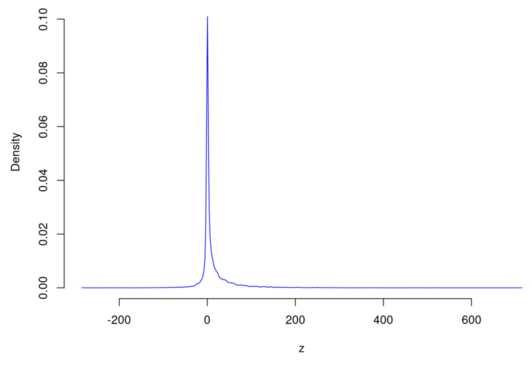
As is reasonable, the second case (with a larger sd) gives estimates that are more spread out (as the variability is higher, by definition).

Now, we are asked to create a new variable $Z = Y^3$ and estimate its distribution. We also need to compute the probability that $Z > 10$. This is extremely easy in `BUGS` — the model code needs to be modified simply to the following.

```
model {
  y ~ dnorm(1, 0.25)
  z <- pow(y, 3)
  # Alternatively, we can write
  # z <- y * y * y
  above10 <- step(z - 10)
}
```

```
          mean    sd   2.5%     25%  50%   75%  97.5%
y        0.997  2.01  -2.91 -0.3593 1.01  2.34   4.98
z       13.089 39.75 -24.61 -0.0464 1.02 12.75 123.20
above10  0.283  0.45   0.00  0.0000 0.00  1.00   1.00
```


Pr(Z > 10)


Density
z