Bayesian inference with **JAGS** and `rjags`

M. L. Delignette-Muller
VetAgro Sup - LBBE

September 21, 2021

Clostridium example

Modeling of the dose-response curve related to the ingestion of Clostridium perfringens.

- **Deterministic part** of the model, probability that the host gets sick:

  $$p = 1 - (1 - r)^{dose}$$

  with *dose* le number of ingested cells

- **Stochastic part** of the model, number of sick hosts *Nsick* for N exposed hosts :

  $$Nsick \sim Binomial(n = N, p = 1 - (1 - r)^{dose})$$

Formalization of a model using a DAG - Directed Acyclic Graph

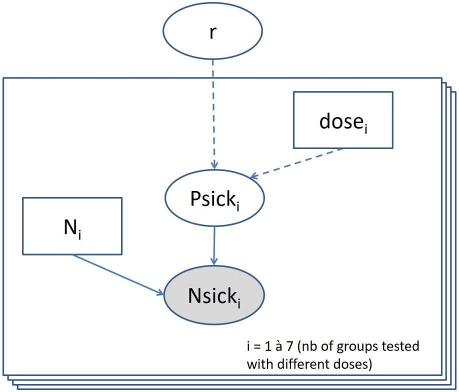
What is a DAG ?

- **a directed graph**
  (all the links are directed)
- **without cycles (loops)**
  (from each node, and following the links, it is impossible to return to this node)
- that we use in Bayesian inference to represent **conditional dependencies** between nodes.
  (you can see a DAG as a mecanistic description of how output data could be used simulated from input data.)

DAG formalism

- **Nodes**
  - covariable (rectangle)
  - variable (ellipse)
    observed variable, latent variable or intermediate variable
    Variables corresponding to output data are sometimes shaded
- **Links**
  - deterministic link (or logical link - dashed arrow - link that could be omitted by writing the model more synthetically)
  - stochastic link (solid line arrow - essential link, that cannot be omitted)

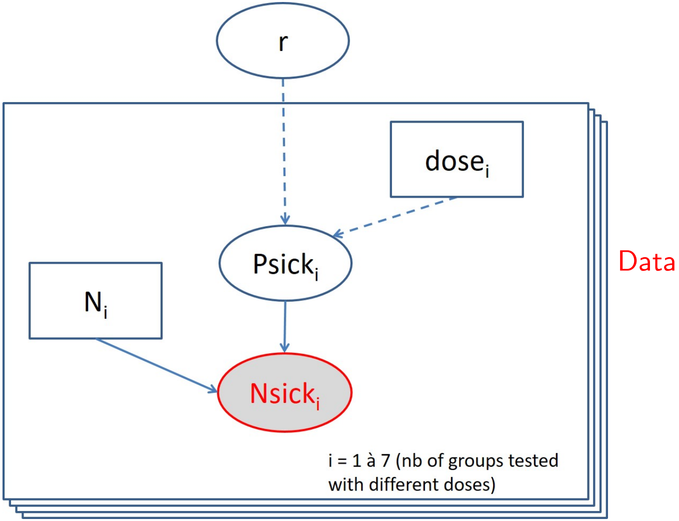
DAG of the model on our example



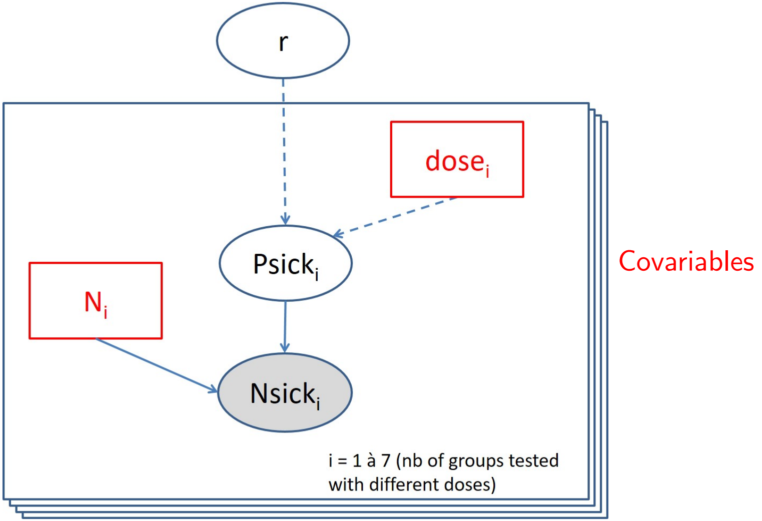
Mathematical definition of links

- Deterministic links
  $Psick_i = 1 - (1 - r)^{dose_i}$
- Stochastic links
  $Nsick_i \sim Binomial(N, Psick_i)$

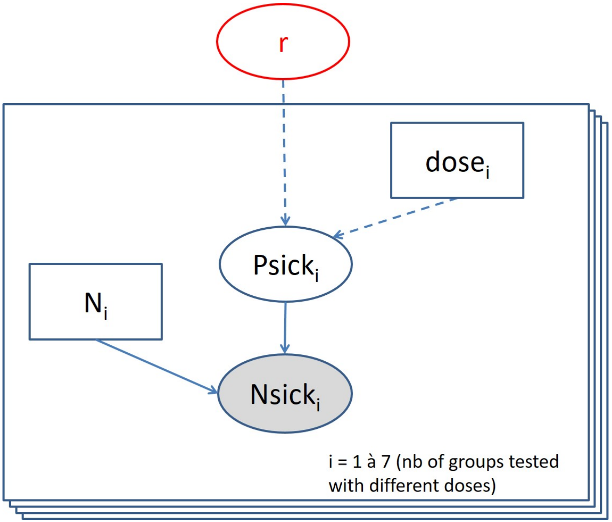
DAG of the model - data (likelihood)



DAG of the model - covariables (explicative variables)



DAG of the model - parameters (to estimate)



Parameters

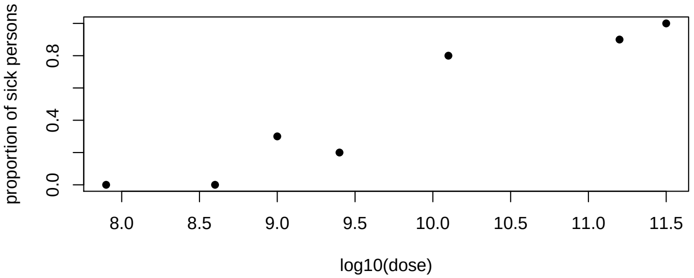
Prior information

In this example, we will assume that from prior information about the unique parameter it is reasonable to define a uniform prior distribution between -15 and -5 on $log_{10}(r)$,

Data related to our example

Number of sick persons *$Nsick_i$* for each group of *N_i* persons exposed at the dose *$dose_i$*

```
> plot(Nsick/N ~ doselog10, data = d, pch = 16,
+ xlab = "log10(dose)", ylab = "proportion of sick persons")
```

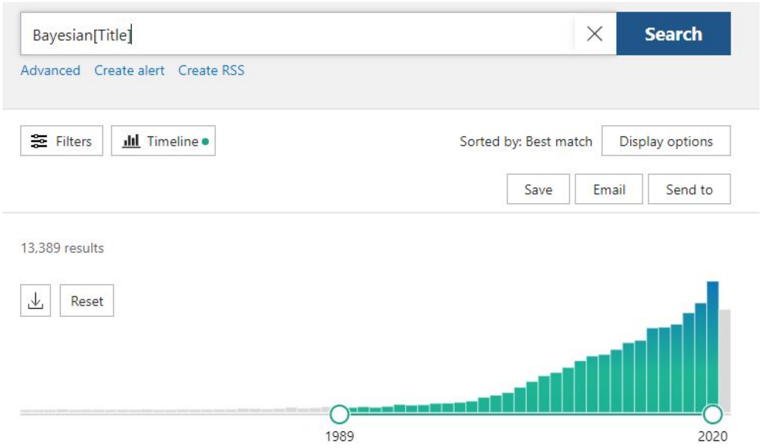


The BUGS project (since 1989)

Bayesian inference **U**sing **G**ibbs **S**ampling
Development and provision of flexible software to implement Bayesian inference on complex models using MCMC.
Some available tools :

- WinBUGS and OpenBUGS
- **JAGS (Just Another Gibbs sampler - Martyn Plummer)**
- Stan and Nimble (new algorithms added to MCMC, that are more efficient for some model families, but may also be inefficient for others)
- RevBayes (for phylogeny)
- several other tools for specific model families

Evolution of the number of PubMed citations with **Bayesian** in the title from the beginning of the project


Bayesian[Title]

13,389 results

Coding of a model in the BUGS language

A declarative language
(the order of the command lines does not matter)
that looks like **R**

- **Declaration of a deterministic node**

```
node <- fonction(some other nodes)
```

- **Declaration of a stochastic node**
  including input nodes,
  i.e. parameters stochastically defined by their prior

```
node ~ distribution(optionally some other nodes)
```

BE CAREFUL: a node on which we have data must always be coded by a stochastic link !

Code of the model in our example

To be written in a text file or in a string as below.

```
> model <-
+ "model
+ {
+   for (i in 1:Ndose)
+   {
+     psick[i] <- 1 - (1 - r)^dose[i]
+     Nsick[i] ~ dbin(psick[i], N[i])
+   }
+   log10r ~ dunif(-15, -5)
+   r <- 10^log10r
+ }
+ "
```

Some properties of the BUGS language that differentiate it from **R**

A node is univariate.
It is necessary to specify the dimensions, the indices, and **explicitely write loops** to define vectors or matrices or multidimensional arrays.
For example, we can write:

```
v[]      v[i]
M[,]     M[i,j]
A[,,,]   A[i,j,k,l]
M[,j]    v[n:m]
x[y[i]]  x[2*j-1]
```

Let us build the code of our model step by step

A loop to define all the observations

```
model
{
   for(i in 1:Ndose)
   {
      Nsick[i] ~ dbin(psick[i], N[i])
   }
}
```

Build of the code - add of intermediate variables

All nodes must be defined in the model except covariables.
The order of lines does not matter.

```
model
{
   for(i in 1:Ndose)
   {
      Nsick[i] ~ dbin(psick[i], N[i])
      psick[i] <- 1 - (1 - r)^dose[i]
   }
}
```

Build of the code - add of priors

Prior distributions of parameters (here just one) must be defined outside the loop.

```
model
{
   for(i in 1:Ndose)
   {
      Nsick[i] ~ dbin(psick[i], N[i])
      psick[i] <- 1 - (1 - r)^dose[i]
   }
  log10r ~ dunif(-15, -5)
  r <- 10^log10r
}
```

Other differences between **BUGS** and **R** languages

BE CAREFUL,
the BUGS language and the R language are different,
and **some differences concern the name of the distributions and their parameterization**.
Refer to the user manual of JAGS or of other languages for a complete and up-to-date list of the functions and distributions.
The JAGS reference manual:
`http://sourceforge.net/projects/mcmc-jags/files/Manuals/`

Coding of data

Coding of data is software-dependent.
Here we will use **JAGS** (MCMC) and **rjags**.
Data must be defined in a data list (here named `data4jags`).

```
> require(rjags)

> data4jags <- list(dose = 10^d$doselog10,
+                   N = d$N,
+                   Nsick = d$Nsick,
+                   Ndose = nrow(d))
```

Pay attention to the consistency between the names used in the model and in the data list

- All the nodes appearing in the model but not defined in the model, so appearing only to the right of an operator, (here *dose* and *N*)
- as well as the max loop indices (here *Ndose*)
- and the output of the model (observed data, here *Nsick*)

must be defined in the data list.

BE CAREFUL to use the same names in the data list and the model code !

Definition of MCMC initial values

Software-dependent coding.
(described here for **JAGS** and **rjags**)
The **definition of initial values** is theoretically required **for each input node and each chain** especially for a correct use of the **Gelman and Rubin statistics** to appreciate the convergence of MCMCs (otherwise, for each parameter, the chains all start from the same value defined by default as a central value of its prior distribution).

Ex.

```
> ini <- list(list(log10r = -12),
+                 list(log10r = -11),
+                 list(log10r = -10))
```

Simulations

- **Build of a model and adaptation**

```
> m <- jags.model(file = textConnection(model),
+                 data = data4jags, inits = ini,
+                 n.chains = 3, n.adapt = 1000)
```

n.adapt (fixed by default to 1000) corresponds to the number of iterations of a phase during which the algorithm is adapted, so during which the simulated values are not yet MCMCs.

- **Burnin phase**

```
> update(m, 3000)
```

- **Monitoring of simulations**

```
> mc <- coda.samples(m, c("r"), n.iter = 1000)
> # generally one starts rather with n.iter around 5000
```

Simulations

- **Build of a model and adaptation**

```
> m <- jags.model(file = textConnection(model),
+                 data = data4jags, inits = ini,
+                 n.chains = 3, n.adapt = 1000)
```

n.adapt (fixed by default to 1000) corresponds to the number of iterations of a phase during which the algorithm is adapted, so during which the simulated values are not yet MCMCs.

- **Burnin phase**

```
> update(m, 3000)
```

- **Monitoring of simulations**

```
> mc <- coda.samples(m, c("r"), n.iter = 1000)
> # generally one starts rather with n.iter around 5000
```

Simulations

- **Build of a model and adaptation**

```
> m <- jags.model(file = textConnection(model),
+                 data = data4jags, inits = ini,
+                 n.chains = 3, n.adapt = 1000)
```

  n.adapt (fixed by default to 1000) corresponds to the number of iterations of a phase during which the algorithm is adapted, so during which the simulated values are not yet MCMCs.

- **Burnin phase**

```
> update(m, 3000)
```

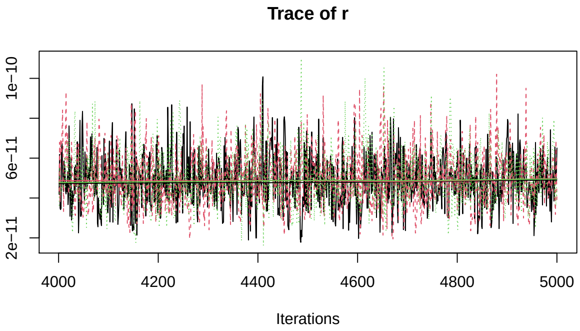
- **Monitoring of simulations**

```
> mc <- coda.samples(m, c("r"), n.iter = 1000)
> # generally one starts rather with n.iter around 5000
```

MCMC trace

All chains should converge to the same limit in term of distribution (stability and overlap/good mixing of the chains).
Here the mixing seems acceptable.

```
> plot(mc, density = FALSE)
```

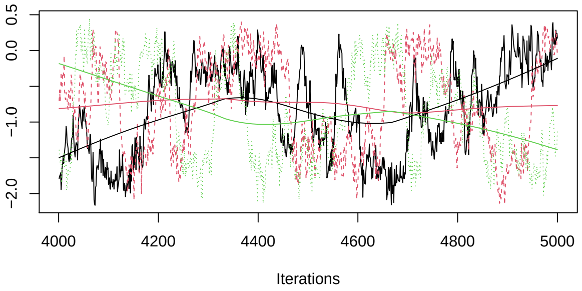


Gelman-Rubin convergence diagnostic

For each parameter, the Gelman-Rubin diagnostic is defined by the square root of the ratio between the variance of its posterior marginal distribution and the intra-chain variance, which we expect to be 1 when convergence is reached.
Gelman gives 1.1 as a maximum acceptable value for all nodes while indicating that one should try to reach 1.00 to get precise final results from MCMCs.

```
> gelman.diag(mc)

Potential scale reduction factors:

  Point est. Upper C.I.
r          1       1.01
```

Example of MCMC chains with a bad overlap



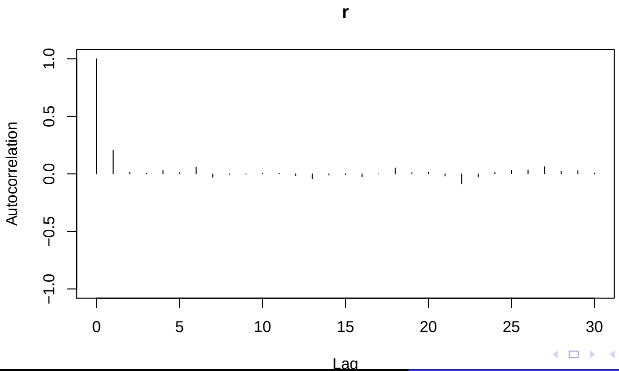
```
> gelman.diag(mc3.3c)

Potential scale reduction factors:

         Point est. Upper C.I.
l10alpha       1.01       1.02
```

Autocorrelation plot

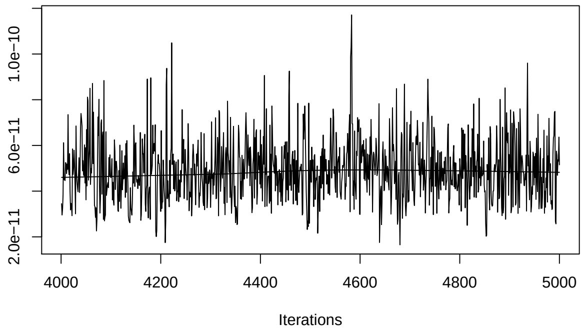
For each chain, plot of the correlation between MCMC iterations as a function of the lag between iterations.

Here the autocorrelation is very low.

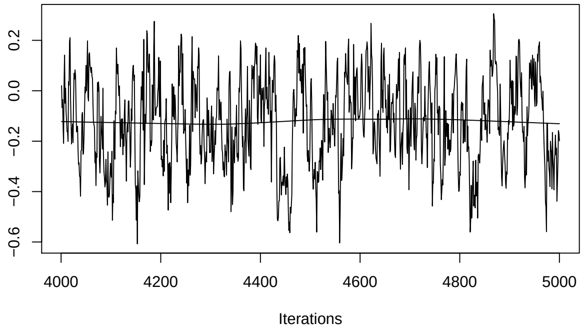
```
> autocorr.plot(mc[[1]])
```



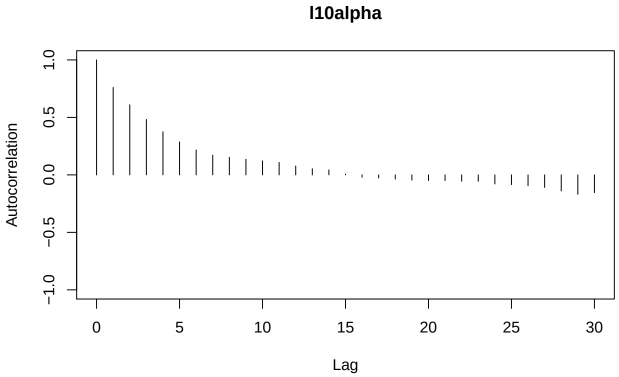
Trace a chain with an acceptable low autorrelation



Trace of a chain with a stronger autocorrelation that would need a thinning

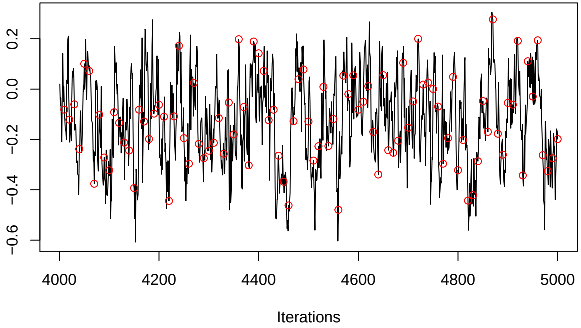


Autocorrelation plot for this chain



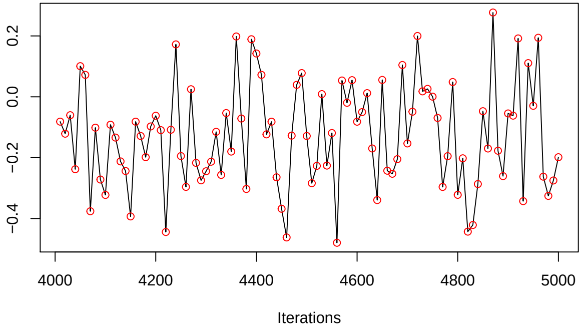
Principle of thinning

With a thin of 10 one stores 1 iteration out of 10.
A thinned chain may contain most of the information when taking up less space in memory.



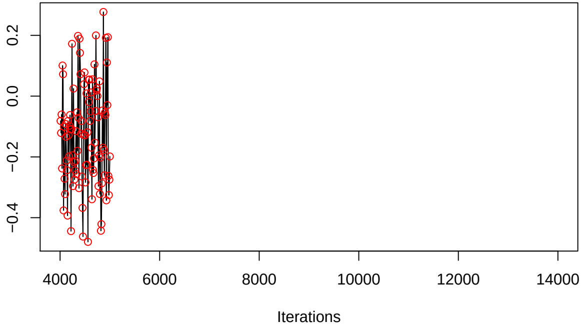
Principle of thinning (2)

After thinning: 100 out of 1000 iterations.



Principle of thinning (3)

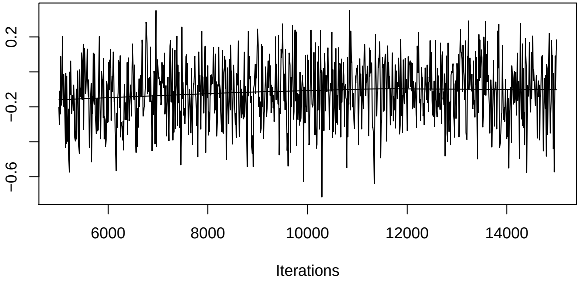
After thinning the number of iterations is low (here only 100).



Principle of thinning (4)

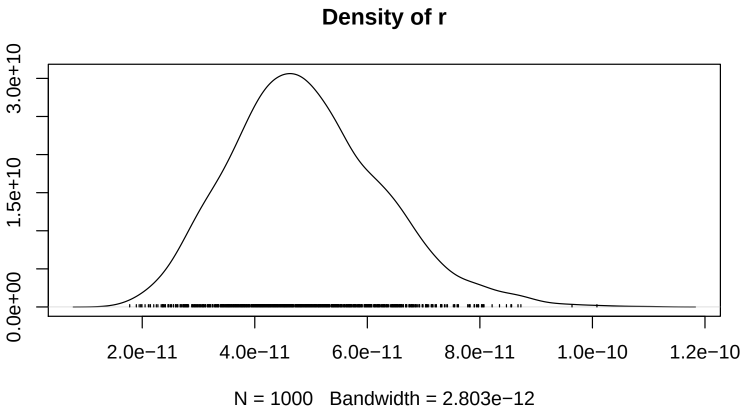
It is thus necessary to increase the initial number of iterations (here $\times 10 \rightarrow$ longer computation).

```
> mc3.1c <- coda.samples(m3.1c, c("l10alpha"), n.iter = 10000, thin = 10)
> plot(mc3.1c, density = FALSE, main = "")
```



Visualisation of the posterior distribution

```
> plot(mc, trace = FALSE)
```



Statistical summary

```
> summary(mc)

Iterations = 4001:5000
Thinning interval = 1
Number of chains = 3
Sample size per chain = 1000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

          Mean             SD       Naive SE Time-series SE
      4.93e-11       1.35e-11       2.47e-13       0.00e+00

2. Quantiles for each variable:

    2.5%      25%      50%      75%    97.5%
2.62e-11 4.00e-11 4.81e-11 5.76e-11 7.95e-11
```

Credibility intervals

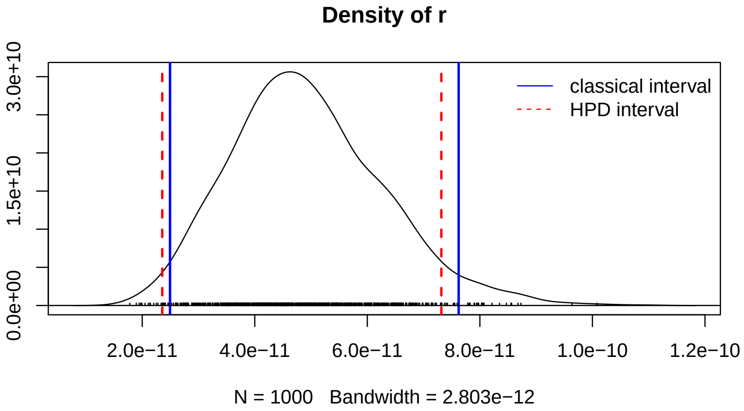
- **Classically based on 2.5% and 97.5% quantiles**

```
> summary(mc)$quantiles

    2.5%      25%      50%      75%    97.5% 
2.62e-11 4.00e-11 4.81e-11 5.76e-11 7.95e-11 
```

- **Less classical High Posterior Density (HPD) intervals**

```
> HPDinterval(mc[[1]], prob = 0.95) # here for the first chain

     lower    upper
r 2.36e-11 7.32e-11
attr(,"Probability")
[1] 0.95
```

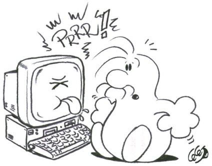
Difference between both intervals for asymmetrical posterior distributions



Conclusion

Now it's your turn to play with JAGS !

To learn the technical aspects, nothing is best than practice !



You have an introductory guide to **JAGS** and `rjags` to help you to start and go further in particular for prediction and model validation.