Introduction à l'inférence bayésienne avec **JAGS** et `rjags`

Marie Laure Delignette-Muller

15 septembre 2021

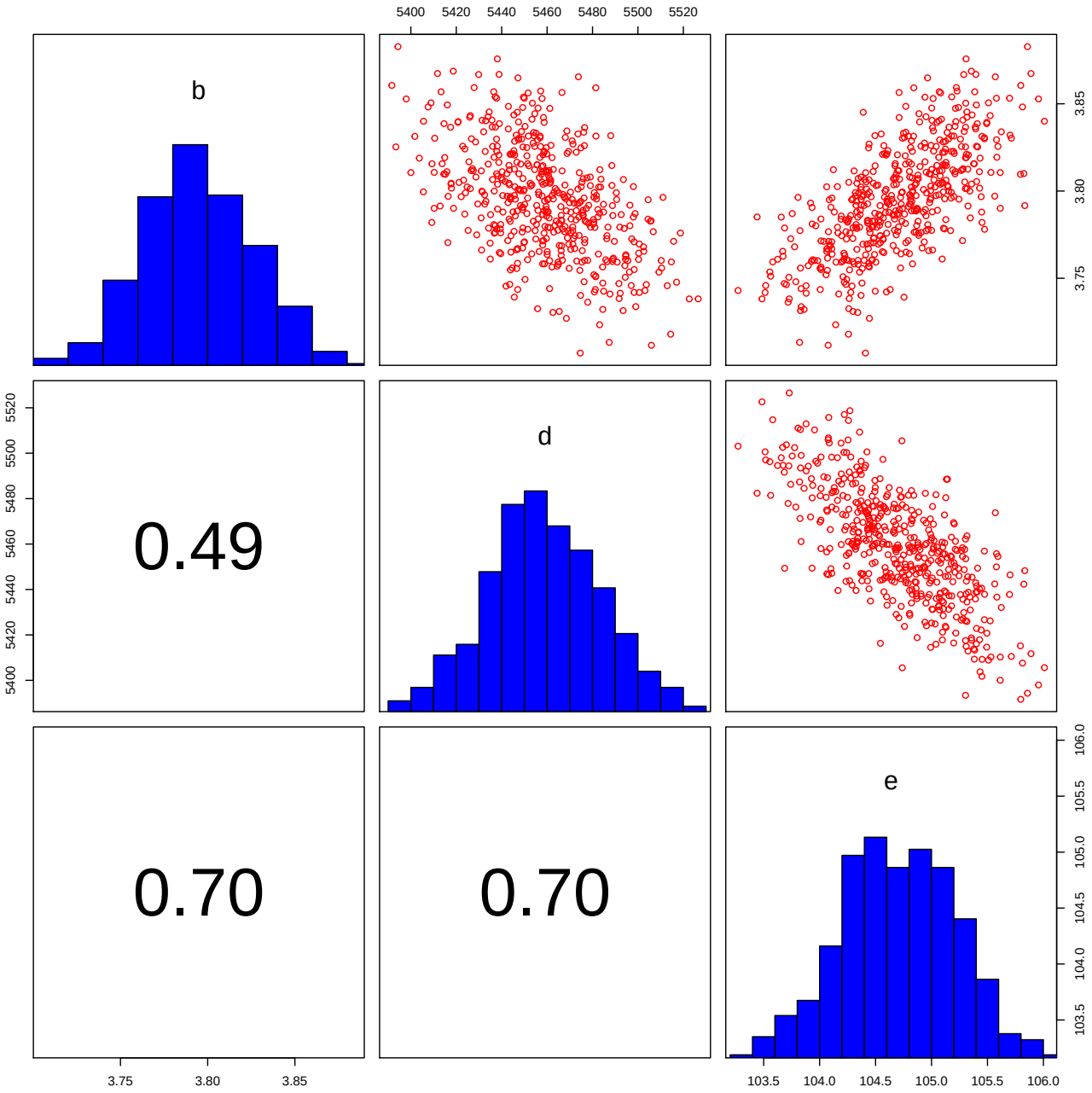


Table des matières

1 Introduction **3**
- 1.1 Objectif général du document . . . 3
- 1.2 Installation de **JAGS** et `rjags` . . . 3
- 1.3 présentation du document et de l'exemple traité . . . 3
- 1.4 Construction du graphe acyclique dirigé . . . 3
- 1.5 Définition des distributions *a priori* des noeuds parents . . . 4
- 1.6 Codage du modèle dans le langage de **JAGS** . . . 4

2 Simulations MCMC **5**
- 2.1 Codification des données . . . 5
- 2.2 Initialisation des chaînes de Markov . . . 5
- 2.3 Simulation des chaînes de Markov . . . 6
- 2.4 Vérification de la convergence des chaînes de Markov . . . 6
- 2.5 Autocorrélation des MCMC . . . 8
- 2.6 Parallélisation des chaînes MCMC . . . 11

3 Utilisation des MCMC **12**
- 3.1 Caractérisation des distributions *a posteriori* marginales . . . 12
- 3.2 Caractérisation de la distribution *a posteriori* jointe . . . 13
- 3.3 Comparaison des lois *a priori* et des lois *a posteriori* . . . 16
- 3.4 Utilisation des distributions *a posteriori* pour estimer une fonction des paramètres . . . 17

4 Validation du modèle **18**
- 4.1 Test de robustesse des résultats aux choix effectués . . . 18
- 4.2 Critère d'ajustement et comparaison de modèles . . . 18
- 4.3 Utilisation des MCMC pour la validation interne ou externe . . . 19

5 Annexe : rudiments pour le codage d'un modèle avec JAGS **22**
- 5.1 Quelques fonctions utilisables . . . 22
- 5.2 Quelques distributions disponibles . . . 23
- 5.3 Troncature et censure . . . 23

1 Introduction

1.1 Objectif général du document

Ce document a pour but de vous initier à l'utilisation du logiciel **JAGS** à partir de **R** via la librairie `rjags` pour estimer les paramètres d'un modèle par inférence bayésienne. Il ne s'agit en aucun cas d'un guide complet d'utilisation, et la consultation des documentations relatives au logiciel **JAGS** (http://sourceforge.net/projects/mcmc-jags/files/Manuals/), à la librairie `rjags` (http://cran.r-project.org/web/packages/rjags/index.html) ainsi que la documentation de **WinBUGS** (http://www.mrc-bsu.cam.ac.uk/wp-content/uploads/manual14.pdf) seront très utiles à l'utilisateur de **JAGS**.

1.2 Installation de JAGS et `rjags`

L'utilisation de **JAGS** à partir de **R** via le paquet `rjags` nécessite l'installation de l'application **JAGS** d'une part (http://sourceforge.net/projects/mcmc-jags/) et du paquet **R** `rjags` (http://cran.r-project.org/web/packages/rjags/index.html) d'autre part.

1.3 présentation du document et de l'exemple traité

La suite du document tente de décrire une démarche structurée d'inférence bayésienne, en se basant sur un exemple simple décrit ci-dessous.

Dans cet exemple on souhaite modéliser l'effet d'un substance toxique susceptible de polluer les eaux douces sur la reproduction d'invertébrés aquatiques. Dans ce but on teste *in vitro* l'effet de diverses concentrations de cette substance sur le nombre total d'oeufs collectés par bécher au cours du bioessai, chaque bécher contenant un nombre défini d'organismes parents (ici cinq). Le plan d'expérience comprend six béchers par concentration testée. On modélisera la moyenne λ du nombre total d'oeufs collectés par bécher en fonction de la concentration en polluant (*conc*) par le modèle log-logistique suivant (partie déterministe du modèle) :

$$\lambda = \frac{d}{1 + (\frac{conc}{e})^b} \tag{1}$$

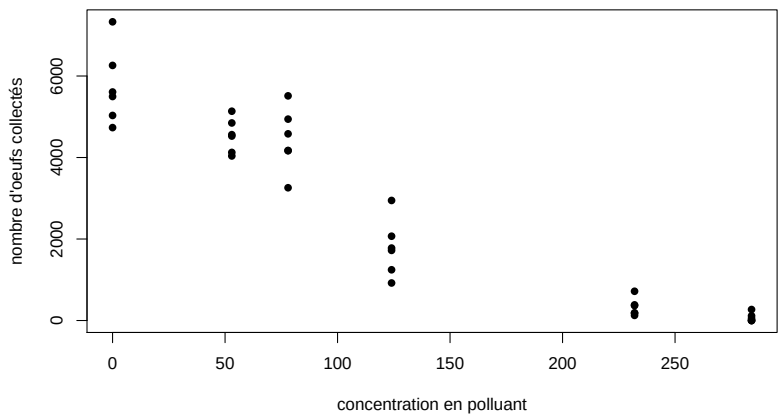
En ce qui concerne la partie stochastique du modèle, comme il est assez classique de le faire sur des données de ce type, on modélisera le nombre d'oeufs cumulé (*Ncumul*) réellement observé par une loi de Poisson, en supposant qu'il n'y a pas de surdispersion induite par une variabilité inter-bécher.

Les données observées ont été codées dans le jeu de données `d` et sont visualisées ci-dessous.

```
str(d)
```

```
## 'data.frame': 36 obs. of  2 variables:
##  $ conc  : int  0 53 78 124 232 284 0 53 78 124 ...
##  $ Ncumul: int  7330 4526 5513 921 187 54 6260 4038 4166 1721 ...
```

```
plot(Ncumul ~ conc, data = d, pch = 16, xlab = "concentration en polluant",
     ylab = "nombre d'oeufs collectés")
```



1.4 Construction du graphe acyclique dirigé

La construction du graphe acyclique dirigé intégrant toutes les variables et paramètres en tant que noeuds du graphe et tous les liens (stochastiques en traits pleins et déterministes en traits pointillés) permet de vérifier que le modèle est complet et correct et que l'on peut le coder facilement. Les noeuds sont représentés par des ellipses, mis à

part ceux correspondant à des covariables (lorsqu'il y en a) que l'on représente souvent par des rectangles. Voici une représentation du graphe acyclique correspondant à notre exemple, avec n le nombre total d'observations de $Ncumul$, correspondant au nombre total de béchers :

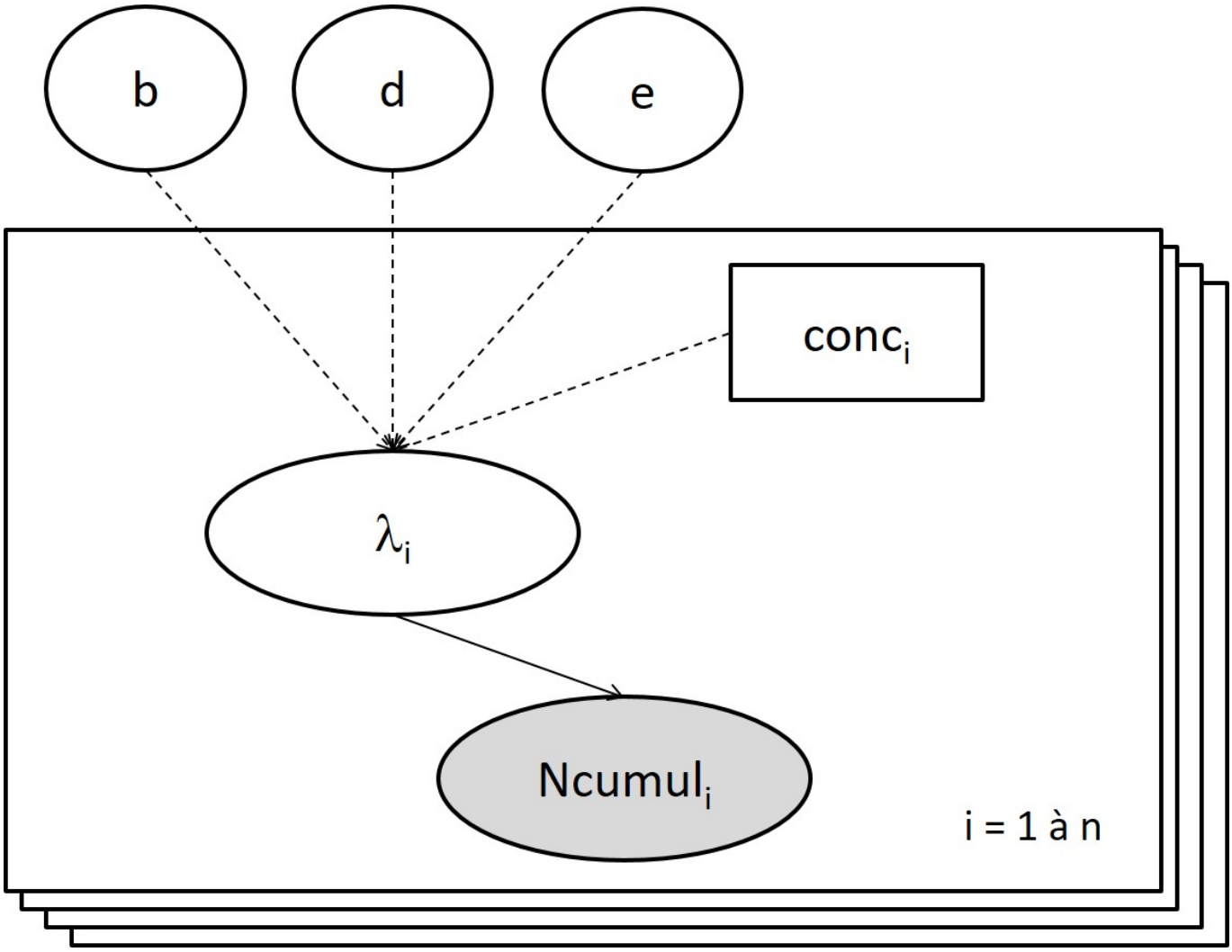


FIGURE 1 – Graphe acyclique dirigé

1.5 Définition des distributions *a priori* des noeuds parents

Pour compléter la construction du modèle, il convient de caractériser les distributions *a priori* des noeuds parents (sans lien entrant) appelés aussi ancêtres du graphe acyclique dirigé. Cette étape n'est pas la plus facile. Elle nécessite de faire le bilan des connaissances *a priori* sur chaque noeud entrant et de choisir une loi *a priori* adaptée pour caractériser cette connaissance. Nous ne rentrerons pas dans le détail de cette caractérisation dans le cadre de ce document. Il faut savoir que c'est sans doute u des points les plus délicats dans la mise en oeuvre d'une inférence bayésienne, qui fait l'objet de maintes discussions scientifiques au sein de la communauté des utilisateurs de ce type d'approche. Lorsqu'un seul choix de la loi *a priori* ne s'impose pas de façon évidente, il sera important de tester la robustesse de l'estimation des paramètres au choix de cette loi, dans une étape ultérieure d'analyse de sensibilité.

Dans cet exemple, on considérera qu'il est raisonnable de définir comme lois *a priori* sur les différents paramètres à estimer :

— une loi uniforme entre 2000 et 10000 sur d,
— une loi uniforme entre -2 et 2 sur $log_{10}(b)$,
— et une loi uniforme entre 1 et 3 sur $log_{10}(e)$.

1.6 Codage du modèle dans le langage de JAGS

Une fois tout ce travail réalisé, le modèle n'a plus qu'à être codé dans un langage de type **BUGS** (ici celui utilisé par **JAGS**) en suivant les règles de syntaxes décrites dans le manuel correspondant (cf. lien internet donné en introduction). Attention, même si les langages de type **BUGS** ressemblent au langage **R**, il ne s'agit pas d'une programmation **R**, mais d'une déclaration du modèle avec définition des noeuds, des liens et des lois *a priori*, et les possibilités d'écriture sont beaucoup plus restreintes qu'avec le langage **R**. En ce qui concerne le langage utilisé dans **JAGS**, vous trouverez en annexe (5) quelques rudiments pour démarrer.

Le modèle peut être décrit dans un fichier texte ou codé directement dans le script **R** sous forme d'une chaîne de caractères entre guillemets, comme ci-dessous la chaîne nommée `codemodel`, que l'on pourra relire ensuite comme le contenu d'un fichier texte à l'aide de la fonction `textConnection()` appliquée à la chaîne de caractères (cf. exemple plus loin).

```
modelPoisson <-
"model
{
  # Definition des liens
  for (i in 1:n)
  {
    lambda[i] <- d / (1 + (conc[i]/e)^b)
    Ncumul[i] ~ dpois(lambda[i])
  }

  # Definition des distributions a priori
  log10b ~ dunif(-2,2)
  d ~ dunif(2000, 10000)
  log10e ~ dunif(1, 3)

  b <- pow(10,log10b)
  e <- pow(10,log10e)
}"
```

ATTENTION, il semble que l'utilisation de caractères spéciaux dans les noms des noeuds, y compris les underscore, puisse provoquer des erreurs de compilation difficiles à identifier

2 Simulations MCMC

2.1 Codification des données

Les données correspondant au modèle décrit précédemment doivent être codées sous forme d'une liste, en utilisant les mêmes noms que ceux utilisés dans le modèle, afin que **JAGS** puisse établir correctement le lien entre les deux. Dans cette liste de données doivent figurer :

— les valeurs observées des sorties du modèle (ici le vecteur *Ncumul*)
— les valeurs des covariables (ici le vecteur *conc*)
— les dimensions des vecteurs (ici n le nombre de concentrations testées)

Voici leur codage pour l'exemple étudié :

```
data4jags <- list(n = length(d$conc),
                  conc = d$conc,
                  Ncumul = d$Ncumul)
```

2.2 Initialisation des chaînes de Markov

Il est tout d'abord recommandé de définir des valeurs initiales pour chaque noeud entrant du DAG et chaque chaîne de Markov, dans une liste du même type que celle définissant les données, nommée `ini` dans l'exemple ci-dessous. Ceci est important pour une utilisation optimale du critère de convergence de Gelman et Rubin (cf. partie 2.4) qui suppose des chaînes partant de valeurs initiales surdispersées par rapport aux distributions *a posteriori*. Ceci peut être fait de la façon suivante :

```
# Definition des valeurs initiales des noeuds entrants pour chaque chaîne
ini1 <- list(log10b = -1, d = 2000, log10e = 2)
ini2 <- list(log10b = 1, d = 8000, log10e = 1)
ini3 <- list(log10b = 0, d = 4000, log10e = 3)
ini <- list(ini1, ini2, ini3)
```

Si on ne fixe pas de valeurs initiales **JAGS** pourra généralement en définir de façon automatique à partir des distributions *a priori* mais pour chaque paramètre il prendra la même pour toutes les chaînes. Ceci devrait changer dans la prochaine version de JAGS (version 5) qui devrait proposer automatiquement des valeurs initiales différentes (il s'agit d'une procédure délicate à mettre en oeuvre : tirer simplement des valeurs initiales aléatoirement dans les lois *a priori* sans exclure des valeurs trop extrêmes causerait beaucoup d'erreurs numériques).

Il arrive dans certains cas que le modèle ne puisse pas tourner (problèmes numériques) pour des valeurs initiales qu'on aurait choisi trop extrêmes. Dans ce cas on peut commencer par lancer les simulations sans fixer de valeurs initiales et les fixer prudemment dans un deuxième temps.

2.3 Simulation des chaînes de Markov

Il est ensuite nécessaire de charger la librairie `rjags` pour utiliser la fonction `jags.model` qui définit un objet représentant un modèle bayésien avec ses données et un nombre de chaînes à simuler fixé. Il est nécessaire d'indiquer comme arguments de cette fonction :

- `file`, le nom du fichier texte dans lequel on a codé le modèle ou, si l'on a codé le modèle directement sous forme d'une chaîne de caractères dans le script **R**, la fonction `textConnection()` appliquée au nom de la chaîne de caractères, comme dans l'exemple ci-dessous.
- `data`, le nom du jeu de données défini auparavant dans le script **R** comme indiqué dans la partie 2.1.
- `inits` la liste des valeurs initiales pour les noeuds entrants définie comme indiqué dans la partie 2.2.
- `n.chains` le nombre de chaînes de Markov, souvent fixé à 3.

```
require(rjags)
model <- jags.model(file = textConnection(modelPoisson), data = data4jags,
                    inits = ini, n.chains = 3)
```

La fonction `jags.model` va vérifier l'intégrité du code du modèle codé et lancer `n.adapt` (fixé par défaut à 1000) premières itérations servant à adapter l'algorithme utilisé pour générer ensuite les MCMC de façon la plus efficace possible. Ces premières itérations ne sont donc pas encore des MCMC.

Une phase de chauffe (dite en anglais "burn-in") est ensuite nécessaire pour amorcer les premières simulations. Les simulations de cette phase de chauffe ne seront pas prises en compte pour estimer les distributions *a posteriori* des paramètres.

La phase de chauffe peut nécessiter un nombre plus ou moins grand d'itérations, suivant la vitesse de convergence de l'algorithme sur un modèle bayésien donné. On pourra partir par défaut de phase de chauffe de 5000 itérations et l'augmenter si nécessaire. La fonction à utiliser pour générer ces premières itérations est `update`.

```
update(model, 5000)
```

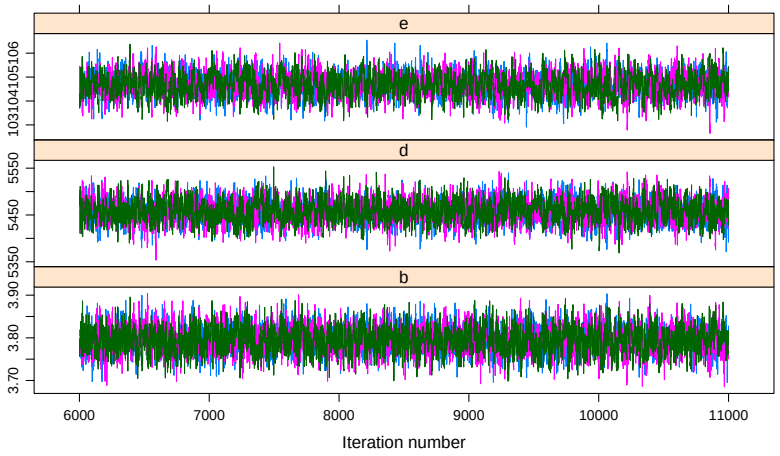
Passé la phase de chauffe, on peut utiliser la fonction `coda.samples` pour générer `niter` autres itérations et garder les valeurs simulées des noeuds qui nous intéressent (les noms sont spécifiés comme ci-dessous dans la fonction), en vue de caractériser la distribution *a posteriori* de ces noeuds.

```
mcmc <- coda.samples(model, c("b", "d", "e"), n.iter = 5000)
```

2.4 Vérification de la convergence des chaînes de Markov

Avant toute interprétation des simulations obtenues en terme de distributions *a posteriori*, il convient de vérifier la convergence des chaînes de Markov. Elles doivent toutes converger vers la même limite en distribution (recouvrement des chaînes à vérifier). Ceci peut déjà être vérifié visuellement en observant les valeurs simulées pour chaque noeud d'intérêt en fonction du nombre d'itérations, à l'aide de la fonction `plot` ou avec la fonction `xyplot` qui utilise le package `lattice`

```
# Tracé basique des chaînes non réalisé ici
# plot(mcmc, trace = TRUE, density = FALSE)
# Tracé des chaînes avec xyplot{lattice}
require(lattice)
xyplot(mcmc)
```



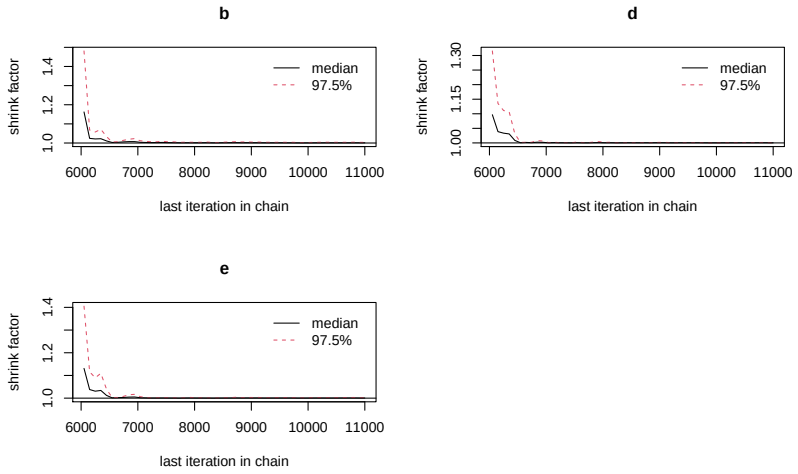
JAGS propose par ailleurs le calcul de plusieurs critères de convergence. Le critère de Gelman et Rubin, appelé indice de rédcution de la variance, est basé sur une comparaison de variance et revient un peu comme en analyse de variance, à se demander s'il y a un effet chaîne sur les valeurs simulées. Le critère est calculé, pour chaque paramètre du modèle, comme la racine carrée du rapport entre la variance de sa distribution *a posteriori* marginale (estimée par une combinaison linéaire de la variance inter et intra chaînes) et la variance intra-chaîne. Il est proche de 1 lorsque la variance inter-chaîne est négligeable par rapport à la variance intra-chaîne (ce que l'on souhaite). La fonction `gelman.diag` permet de calculer ce critère :

```
gelman.diag(mcmc)
```

```
## Potential scale reduction factors:
##
##   Point est. Upper C.I.
## b          1          1
## d          1          1
## e          1          1
##
## Multivariate psrf
##
## 1
```

Une fonction graphique permet aussi de voir comment ce critère évolue quand on augmente la taille de la fenêtre de calcul, c'est-à-dire le nombre d'itérations utilisées pour le calculer. La première valeur du graphe est par défaut calculé avec les 50 premières itérations. La fonction `gelman.plot` permet de réaliser cette représentation graphique. Ce graphe permet de voir si on converge dès les premières itérations et de vérifier que le nombre d'itérations est suffisant pour que le critère de Gelman et Rubin converge, et qu'il converge vers 1.

```
gelman.plot(mcmc)
```



Un autre critère de convergence est proposé dans **JAGS**, le critère de Geweke. Il est basé sur le test de l'égalité des moyennes des valeurs simulées en début de chaîne et en fin de chaîne. Ce critère est calculé pour chaque chaîne et chaque noeud d'intérêt (sélectionné dans `coda.samples`), en utilisant le premier dixième des itérations pour calculer les moyennes en début de chaîne, et la deuxième moitié des itérations pour calculer les moyennes en fin de chaîne. La valeur du critère est un score qui suit une loi normale centrée réduite lorsqu'il y a convergence. La fonction `geweke.diag` permet de calculer ces scores pour chaque chaîne et chaque noeud. Il convient donc de vérifier que toutes les valeurs calculées se trouvent grosso modo entre -2 et 2.

```
geweke.diag(mcmc)
```

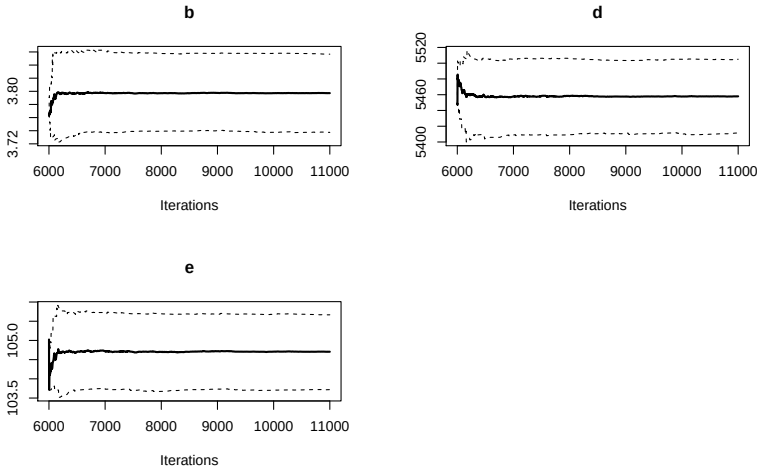
```
## [[1]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##       b       d       e
##  0.1779 -0.0282  0.2825
##
##
## [[2]]
```

```
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##      b      d      e
## -0.619  0.297 -0.506
##
##
## [[3]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##      b      d      e
##  0.524 -0.585  0.643
```

La fonction graphique `geweke.plot` peut aussi être utilisée en complément. Elle trace pour chaque chaîne et chaque noeud, l'évolution du score lorsque l'on enlève des bouts de plus en plus grands de la première moitié de chaque chaîne.

La fonction graphique `cumuplot` peut aussi être intéressante pour juger de la convergence des chaînes. Elle trace, pour chaque chaîne et chaque noeud, l'évolution en fonction du nombre d'itérations de la médiane et des quantiles à 0.025 et 0.975 des valeurs simulées. Il permet de voir si on a utilisé suffisamment d'itérations pour avoir une estimation fiable et stable de ces quantiles.

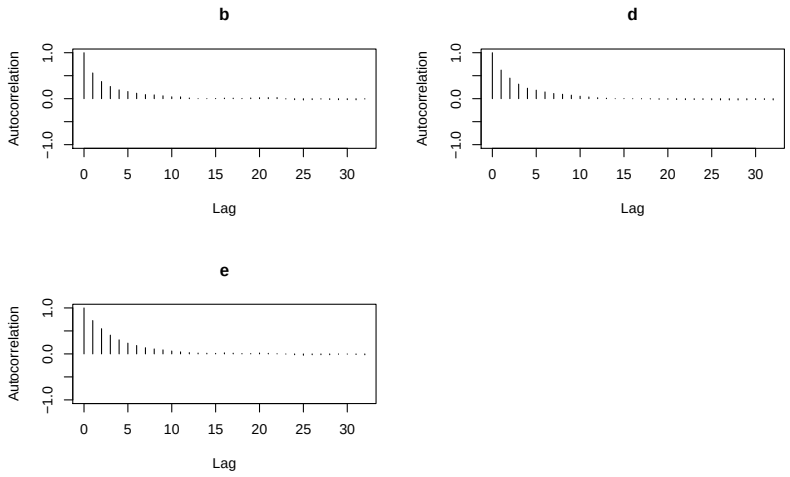
```
cumuplot(mcmc[[1]]) # ici appliqué uniquement à la première chaîne pour limiter l'encombrement
```



2.5 Autocorrélation des MCMC

Les chaînes de Markov, de part leur construction, sont souvent autocorrélées. Il est donc intéressant de détecter visuellement une éventuelle autocorrélation à l'aide des fonctions `autocorr` et `autocorr.plot`. En cas d'autocorrélation importante, il est raisonnable d'augmenter le nombre d'itérations et de ne garder qu'une simulation sur un nombre choisi habilement de façon à ne plus vois d'autocorrélation, en fixant l'argument `thin` à ce nombre choisi dans la fonction `coda.samples` (cf. `?coda.samples`).

```
autocorr.plot(mcmc[[1]]) # ici appliqué uniquement à la première chaîne pour limiter l'encombrement
```



Un autre type de représentation de l'autocorrélation entre les valeurs de paramètres simulées, sur un seul graphe, avec différentes couleurs pour les différentes chaînes, peut être obtenu avec la fonction `acfplot` qui utilise le package `lattice`.

Dans cet exemple l'autocorrélation n'est pas négligeable, et il serait raisonnable de multiplier le nombre d'itérations de quelques unités et de mettre un `thin` associé. La fonction `effectiveSize` donne, par paramètre, une estimation de la taille efficace des chaînes en prenant en compte l'autocorrélation. Par définition, pour une série X de taille N, l'erreur standard de la moyenne est la racine carrée de $\frac{V(X)}{n}$ avec n la taille efficace de la série. S'il n'y a pas d'autocorrélation $n = N$. Dans l'exemple ci-dessous la taille efficace est de l'ordre de 2000 pour $3 \times 5000 = 15000$ itérations réalisées sur le paramètre avec le plus d'autocorrélation (e ici). Ce résultat donne une idée de la valeur du `thin` à choisir si on veut éliminer complètement l'autocorrélation, en faisant le rapport du nombre total d'itérations réalisées sur la taille efficace (ici on prendrait un `thin` d'environ 7-8 : $\frac{15000}{2000} = 7.5$).

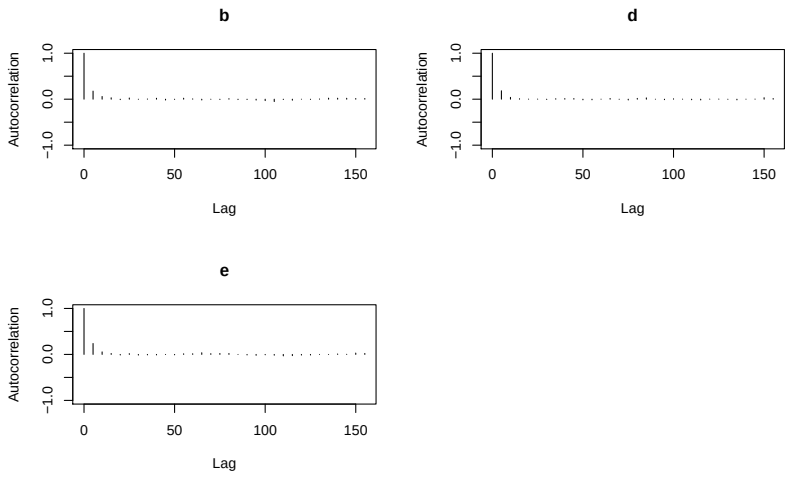
```
# estimation de la taille effective du fait de l'autocorrélation pour l'ensemble
# des 3 chaînes 3*5000 = 15000
effectiveSize(mcmc)

##    b    d    e
## 2738 2546 2131

# même calcul par chaîne
lapply(mcmc, effectiveSize)

## [[1]]
##   b   d   e
## 999 862 723
##
## [[2]]
##   b   d   e
## 873 840 697
##
## [[3]]
##   b   d   e
## 865 844 711
```

La fonction `raftery.diag` prend en compte cette autocorrélation des chaînes pour estimer, pour chaque paramètre, un nombre minimal total d'itérations (`N`) pour estimer correctement la distribution *a posteriori* de ce paramètre en prenant en compte l'autocorrélation. Lorsque le facteur de dépendance dépasse 5 cela indique une forte autocorrélation. Ce calcul vise en fait à obtenir une précision donnée (paramétrable) sur un quantile de la distribution *a posteriori*, par défaut le quantile à 2.5% (cf. `?raftery.diag` pour une description plus détaillée). On se place généralement un peu au-dessus des valeurs indiquées. Dans cet exemple où le facteur de dépendance (`I`, rapport entre `N` et `Nmin`), est toujours inférieur à 4, on serait tranquille en fixant un `thin` à 5 et `n.iter` à 20000.

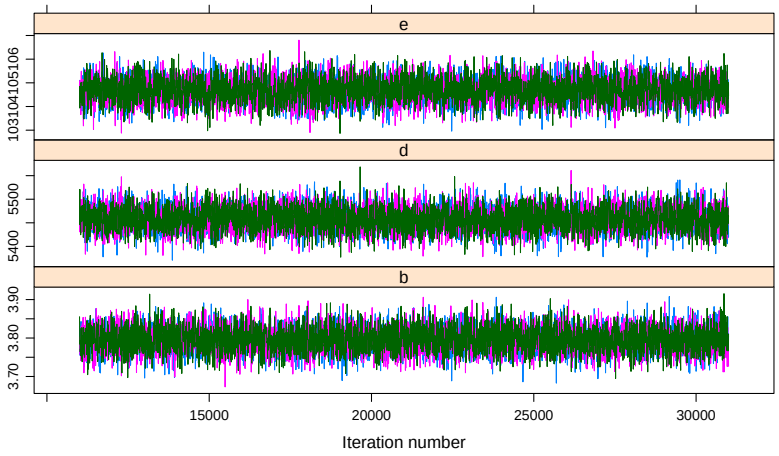
```
# diagnostic de Raftery
raftery.diag(mcmc)

## [[1]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
```

```
##
##    Burn-in  Total Lower bound  Dependence
##    (M)      (N)   (Nmin)       factor (I)
##  b 7        8120  3746         2.17
##  d 8        8602  3746         2.30
##  e 12       14562 3746         3.89
##
##
## [[2]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
##    Burn-in  Total Lower bound  Dependence
##    (M)      (N)   (Nmin)       factor (I)
##  b 8        8945  3746         2.39
##  d 7        7968  3746         2.13
##  e 10       10758 3746         2.87
##
##
## [[3]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
##    Burn-in  Total Lower bound  Dependence
##    (M)      (N)   (Nmin)       factor (I)
##  b 8        10112 3746         2.70
##  d 14       16400 3746         4.38
##  e 8        9770  3746         2.61
```

```
# nouvelles simulations avec n.iter et thin adaptés
mcmc <- coda.samples(model, c("b", "d", "e"), n.iter = 20000, thin = 5)
# nouveau graphe des autocorrélations
autocorr.plot(mcmc[[1]]) # ici appliqué uniquement à la première chaîne pour limiter l'encombrement
```




```
# nouveau graphe des auto-corrélations
xyplot(mcmc)
```



On peut observer sur les figures ci-dessus, qu'il n'y a quasi plus d'autocorrélation dans le graphe d'auto-corrélation et que le recouvrement des trois chaînes est meilleur, avec moins de vagues de chaque chaîne.

Lorsque le nombre d'itérations de l'objet `mcmc` n'est pas de taille suffisante pour pouvoir faire le calcul proposé par Raftery, la fonction `raftery.diag` met un message d'erreur et indique le nombre d'itérations minimal nécessaire pour appliquer la fonction.

2.6 Parallélisation des chaînes MCMC

Les calculs pouvant être longs lors de simulations MCMC, notamment si l'on doit augmenter le nombre d'itérations du fait de l'autocorrélation, il peut être intéressant de paralléliser le calcul des chaînes. Le package `dclone` peut être utilisé à cet effet. L'usage des fonctions de ce package impose de travailler sur un modèle codé dans un fichier texte (on nen peut pas utiliser la fonction textConnection() comme dans l'exemple présenté précédemment, mais il suffit de mettre le code du modèle dans un fichier texte et d'appeler le nom de ce fichier texte qui est dans cet exemple `modelPoisson.txt`).

Dans l'exemple ci-dessous sont comparées les versions utilisant `rjags` et `dclone`. Le calcul parallèle n'est pas ici nettement plus rapide que le calcul standard, mais il le devient vite lorsque les calculs sont nettement plus longs (plusieurs heures ou jours).

```
### Version non parallèle sur un grand nombre d'itérations ####
t1 <- Sys.time()
model <- jags.model(file = "MODEL/modelPoisson.txt",
                    data = data4jags, inits = ini, n.chains = 3)
update(model, 5000)
mcmc <- coda.samples(model, c("b", "d", "e"), n.iter = 200000, thin = 40)
t2 <- Sys.time()
```

```
# temps de calcul
t2 - t1

## Time difference of 31.1 secs
```

```
### Version parallèle ################################
library(dclone)
cl <- makePSOCKcluster(3)
# sous linux ou MAC ce sera plus efficace d utiliser makeForkCluster
# cl <- makeForkCluster(3)

t1 <- Sys.time()
parJagsModel(cl, name = "modelpar", file = "MODEL/modelPoisson.txt",
                    data = data4jags, inits = ini, n.chains = 3)
parUpdate(cl, object = "modelpar", n.iter = 5000)
mcmcpar <- parCodaSamples(cl, model = "modelpar",
                          variable.names = c("b", "d", "e"),
                          n.iter = 200000, thin = 40)
t2 <- Sys.time()
stopCluster(cl)
```

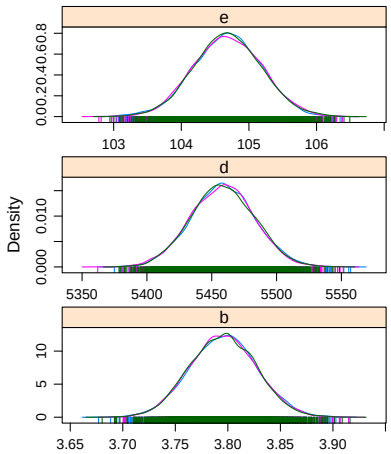
```
# temps de calcul
t2 - t1

## Time difference of 18.5 secs
```

3 Utilisation des MCMC

3.1 Caractérisation des distributions *a posteriori* marginales

Une fois que l'on s'est assuré de la convergence des chaînes de Markov vers la distribution *a posteriori* des paramètres que l'on souhaite estimer, il est temps d'utiliser les fonctions de **JAGS** qui permettent de caractériser les distributions obtenues. La fonction `plot` déjà mentionnée précédemment pour visualiser la trace de toutes les simulations peut aussi être utilisée pour visualiser les densités de probabilité pour chaque noeud, et la fonction `summary` pour calculer les paramètres statistiques décrivant ces distributions. Les fonctions de densité *a posteriori* peuvent aussi être tracées avec différentes couleurs correspondant aux différentes chaînes avec la `densityplot` du package `lattice`.

```
summary(mcmc)

##
## Iterations = 6040:206000
## Thinning interval = 40
## Number of chains = 3
## Sample size per chain = 5000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##     Mean      SD Naive SE Time-series SE
## b    3.8  0.0316 0.000258       0.000251
## d 5458.6 24.5322 0.200305       0.200305
## e  104.7  0.5022 0.004100       0.004090
##
## 2. Quantiles for each variable:
##
##      2.5%     25%    50%     75%   97.5%
## b    3.73    3.77    3.8    3.82    3.86
## d 5411.05 5441.99 5458.5 5475.04 5506.79
## e  103.70  104.35  104.7  105.02  105.67

# Tracé basique des densités non réalisé ici
# plot(mcmc, trace = FALSE, density = TRUE)
# Tracé des densités avec densityplot{lattice}
densityplot(mcmc)
```



les intervalles de crédibilité sur chacun des paramètres peuvent être estimés classiquement à partir des quantiles à 2.5% et 97.5% donnés dans le résumé précédent (`summary(mcmc)`) mais on peut aussi les estimer autrement, comme les intervalles de plus forte densité (High Posterior density intervals) à l'aide de la fonction `HPDinterval` . Le calcul d'un intervalle de plus forte densité à 95% pour un paramètre donne le plus petit intervalle correspondant à un niveau

de 95%. Dans le cas de distributions *a posteriori* marginales dissymétriques, on obtiendra des résultats différents entre ces deux types d'intervalles.

```
HPDinterval(mcmc)
```

```
## [[1]]
##     lower   upper
## b    3.73    3.86
## d 5410.57 5506.97
## e  103.70  105.64
## attr(,"Probability")
## [1] 0.95
##
## [[2]]
##     lower   upper
## b    3.73    3.86
## d 5409.51 5505.33
## e  103.70  105.65
## attr(,"Probability")
## [1] 0.95
##
## [[3]]
##     lower   upper
## b    3.74    3.86
## d 5410.98 5505.84
## e  103.62  105.60
## attr(,"Probability")
## [1] 0.95
```

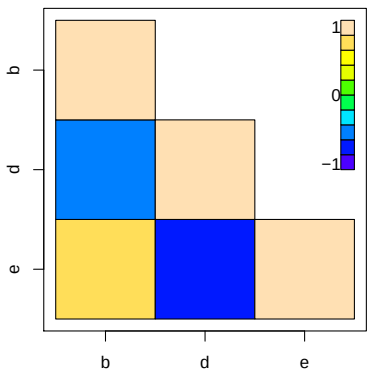
3.2 Caractérisation de la distribution *a posteriori* jointe

Pour mieux apprécier la distribution **jointe** des paramètres, la corrélation entre les paramètres estimés peut être évaluée à l'aide des fonctions `crosscorr` et `crosscorr.plot`.

```
crosscorr(mcmc)
```

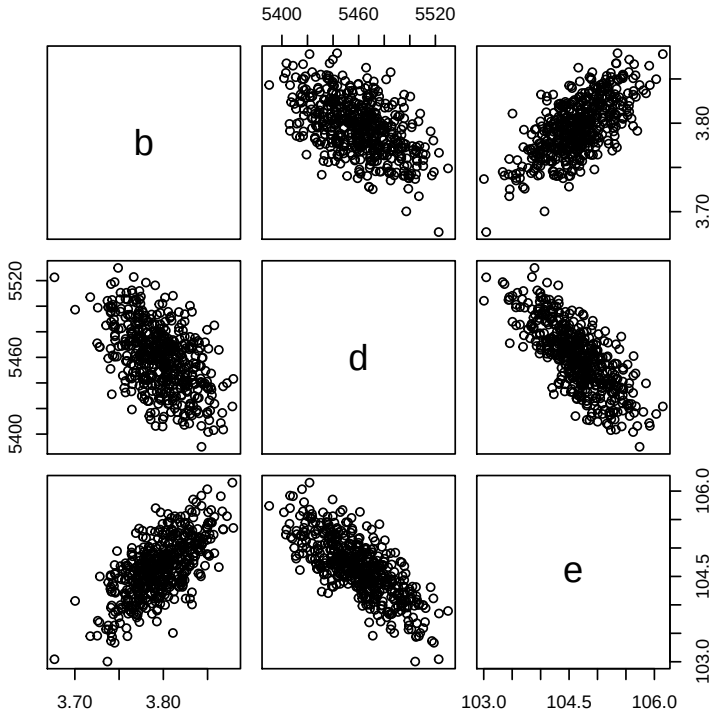
```
##        b      d      e
## b  1.000 -0.488  0.665
## d -0.488  1.000 -0.722
## e  0.665 -0.722  1.000
```

```
crosscorr.plot(mcmc)
```



Par ailleurs, toutes les fonctions graphiques permettant de visualiser un gros jeu de données peuvent aussi être utilisées sur la concaténation des différentes chaînes dans un même `data.frame`. Par exemple la fonction `pairs` peut être utilisée très facilement pour visualiser la distribution *a posteriori* jointe des paramètres en projetant les valeurs simulées d'une chaîne dans les divers plans. On peut éventuellement tirer un échantillon du jeu de simulations MCMC précédent (ce qui est fait dans le code suivant : `mcmctotsample`) afin d'alléger les fichiers graphiques.

```
### Concaténation des valeurs simulées dans les 3 chaînes
### dans un même jeu de données
mcmctot <- as.data.frame(as.matrix(mcmc))
### Tirage au hasard de 500 simulations
mcmctotsample <- mcmctot[sample.int(nrow(mcmctot), size = 500), ]
### Représentations des valeurs correspondantes sur chaque plan de 2 paramètres
pairs(mcmctotsample)
```



La fonction `ipairs` du package `IDPmisc` permet de faire le même type de représentation avec en ajout un code de couleur pour visualiser la densité de points.

```
# code non lancé ici
# require(IDPmisc)
# ipairs(mcmctotsample)
```

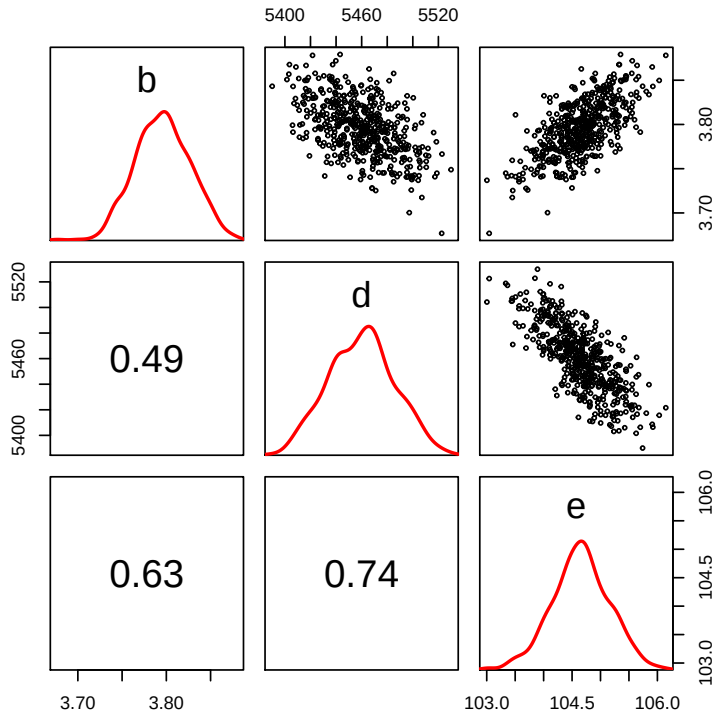
On peut aussi utiliser les différentes possibilités associées à la fonction `pairs` afin de créer une figure représentant la loi *a posteriori* des paramètres à la fois avec des graphes de densité, des nuages de points, l'affichage de valeurs de coefficients de corrélation de rangs entre les paramètres pris deux à deux ou toute autre fonction graphique à spécifier par l'utilisateur. Pour réaliser ce type de représentation, il convient tout d'abord de définir les fonctions de tracé à utiliser sur les graphes de la diagonale (`diag.panel`) ou les graphes du dessus (`upper.panel`) ou les graphes du dessous (`lower.pannel`) (cf. `?pairs`).

```
panel.hist <- function(x, col.hist = "grey", ...)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(usr[1:2], 0, 1.5) )
    h <- hist(x, plot = FALSE)
    breaks <- h$breaks; nB <- length(breaks)
    y <- h$counts; y <- y/max(y)
    rect(breaks[-nB], 0, breaks[-1], y, col=col.hist, ...)
}

panel.dens <- function(x, col.dens = "red", lwd.dens = 2, ...)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(usr[1:2], 0, 1.5) )
        densx <- density(x)
        vx <- densx$x
        vy <- densx$y
```

```
        lines(vx,vy/max(vy),col=col.dens,lwd=lwd.dens, ...)
}

panel.cor <- function(x, y, digits=2, prefix="", cex.cor, ...)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y,method="spearman"))
    txt <- format(c(r, 0.123456789), digits=digits)[1]
    txt <- paste(prefix, txt, sep="")
        if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
   #text(0.5, 0.5, txt, cex = cex.cor * r)
        text(0.5, 0.5, txt, cex = cex.cor * 0.5, ...)
}

panel.xy <- function(x, y, pch.xy = 1, col.xy = "black", cex.xy = 0.5, ...)
{
        points(x,y,pch=pch.xy, col=col.xy,cex=cex.xy, ...)
}
```

Une fois ces fonctions définies, il est très facile de créer les graphes de son choix comme ci-dessous.

```
pairs(mcmctotsample, upper.panel = panel.xy,
      diag.panel = panel.dens,
        lower.panel = panel.cor)
```

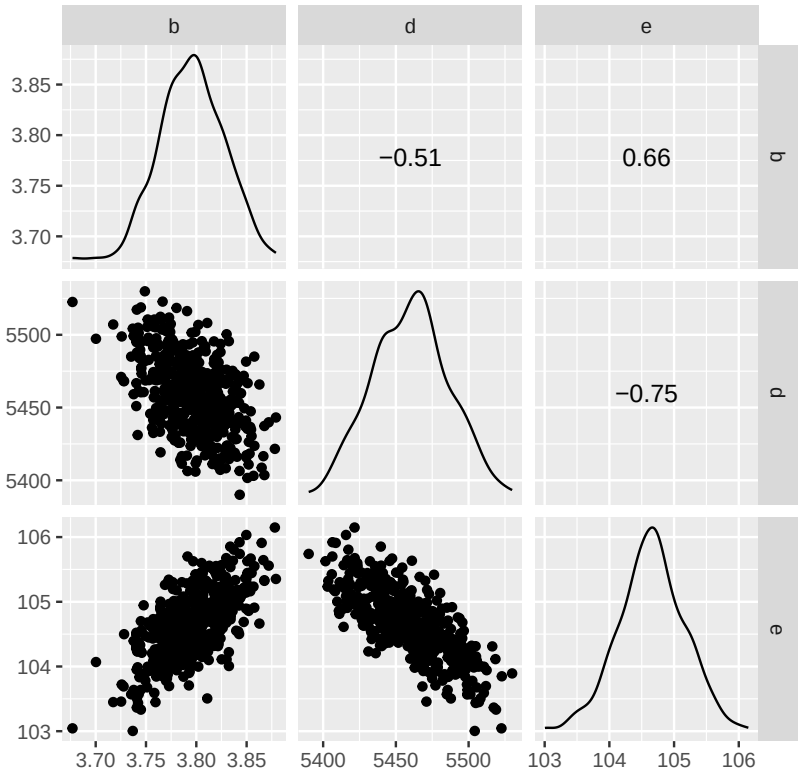


Pour un autre exemple, le code ci-dessous correspond à la réalisation de la figure de la première page de ce document.

```
pdf(file = "figpage1.pdf", width = 10, height = 10)
pairs(mcmctotsample, upper.panel = panel.xy,
        diag.panel = panel.hist,
           lower.panel = panel.cor,
        col.hist = "blue", pch.xy = 1, col.xy = "red", cex.xy = 1)
dev.off()
```

En utilisant les packages `ggplot2` et `GGally` on peut faire le même type de graphe en une ligne de code :

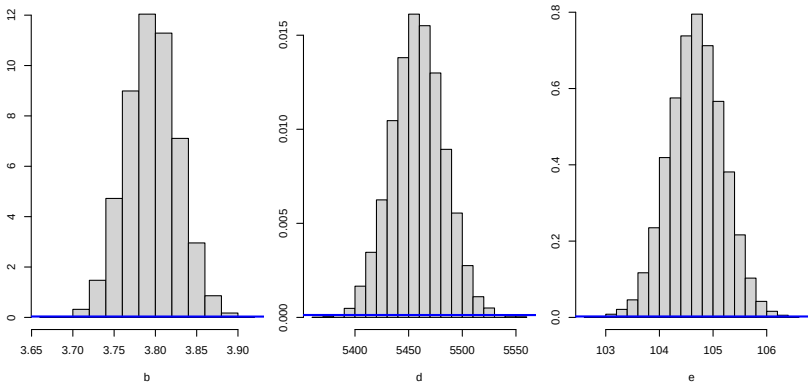
```
require(GGally)
ggscatmat(mcmctotsample)
```



3.3 Comparaison des lois *a priori* et des lois *a posteriori*

A la fois pour visualiser les lois *a priori* et pour juger des impacts respectifs de l'information *a priori* et des données sur les distributions *a posteriori*, il est souvent intéressant de représenter les lois marginales *a priori* et *a posteriori* pour chaque paramètre sur le même graphe. Pour le faire très simplement on peut lancer des simulations sans les données, qui correspondent à des tirages dans les lois *a priori* et comparer visuellement les lois *a priori* et *a posteriori* de la façon suivante :

```
d0 <- list(n = length(d$conc), conc = d$conc)
model0 <- jags.model(file = textConnection(modelPoisson), data = d0, n.chains = 1)

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 0
##    Unobserved stochastic nodes: 39
##    Total graph size: 185
##
## Initializing model

update(model0, 5000)
mcmc0 <- coda.samples(model0, c("b", "d", "e"), n.iter = 5000)
mcmctot0 <- as.data.frame(as.matrix(mcmc0))
par(mfrow = c(1, 3))
par(mar = c(5, 2, 1, 1))
for (i in 1:ncol(mcmctot0))
{
        hist(mcmctot[,i], main = "", xlab = names(mcmctot0)[i], freq = FALSE)
        lines(density(mcmctot0[,i]), col = "blue", lwd = 2)
}
```



Dans cet exemple on voit que les données sont très informatives et permettent de resserrer très nettement les lois *a priori*.

ATTENTION, ce type de représentation est très informatif mais parfois non présentable en l'état, notamment lors de l'utilisation de lois *a priori* uniformes, du fait de l'utilisation de la fonction `density()` qui ne permet pas de donner une bonne représentation d'une loi uniforme. Pour une représentation propre, il conviendra de générer le graphe de densité de chaque loi *a priori* directement à partir de la définition de la loi et non à partir d'un échantillon tiré de cette loi.

3.4 Utilisation des distributions *a posteriori* pour estimer une fonction des paramètres

Les chaînes de Markov simulées peuvent être récupérées, concaténées, et utilisées dans **R** comme un échantillon de la loi *a posteriori* jointe, par exemple pour prendre une décision par rapport à une hypothèse biologique, pour calculer un intervalle de crédibilité sur n'importe quelle fonction des paramètres et juger de la qualité d'ajustement du modèle en comparant les données observées et celles qu'il peut prédire (cf. partie 4).

Dans l'exemple qui nous intéresse, imaginons que nous voulions estimer la concentration efficace à 10% appelée EC_{10}. Elle est définie comme la concentration qui induit une diminution relative de la reproduction de 10%. On peut donc la formuler mathématiquement à partir de la partie déterministe du modèle et de ses paramètres :

$$\frac{d}{1+(\frac{EC_{10}}{e})^b} = d \times \frac{9}{10} \Leftrightarrow EC_{10} = e \times (\frac{1}{9})^{\frac{1}{b}} \tag{2}$$

Si l'on veut estimer la EC_{10} à partir de la loi *a posteriori*, il suffit d'utiliser un échantillon ou l'ensemble des jeux de paramètres des chaînes de Markov simulés, et de calculer, pour chaque jeu de paramètres, la valeur de EC_{10} à partir de la formule ci-dessus, puis de calculer les quantiles de ces valeurs simulées (médiane pour l'estimation ponctuelle et quantiles à 2.5% et 97.5% pour l'intervalle de crédibilité à 95%), comme ci-dessous.

```
### Concaténation des valeurs simulées dans les 3 chaînes
### dans un même jeu de données
mcmctot <- as.data.frame(as.matrix(mcmc))
## Récupération sous forme de vecteurs des MCMC pour les paramètres
### dont nous avons besoin pour calculer la EC10
b <- mcmctot[, "b"]
e <- mcmctot[, "e"]
### Calcul de la EC10 pour chaque jeu de paramètres
EC10 <- e * (1/9) ^(1/b)
### Calcul des quantiles à 2.5, 50 et 97.5 %
quantile(EC10, probs = c(0.025, 0.5, 0.975))

##  2.5%   50% 97.5%
##  57.7  58.7  59.7
```

On peut de la même façon prédire la réponse du modèle pour une valeur donnée de la covariable. Par exemple le code ci-dessous donne la prédiction du nombre d'oeufs collectés pour une expérience similaire qui serait réalisée à une concentration en polluant égale à 60.

```
### Concaténation des valeurs simulées dans les 3 chaînes
### dans un même jeu de données
mcmctot <- as.data.frame(as.matrix(mcmc))
## Récupération sous forme de vecteurs des MCMC pour les paramètres
vb <- mcmctot[, "b"]
vd <- mcmctot[, "d"]
ve <- mcmctot[, "e"]
### Prédiction de la réponse pour une concentration de 60 pour chaque jeu de paramètres
```

```r
conc <- 60
vlambda <- vd / (1 + (conc/ ve) ^ vb)
vNcumul <- rpois(n = length(vlambda), lambda = vlambda)
### Calcul des quantiles à 2.5, 50 et 97.5 %
quantile(vNcumul, probs = c(0.025, 0.5, 0.975))

##  2.5%   50% 97.5%
##  4729  4870  5012
```

4 Validation du modèle

4.1 Test de robustesse des résultats aux choix effectués

Il est important, dans une démarche de modélisation bayésienne, de tester la robustesse des estimations obtenues aux divers choix réalisés au cours de la modélisation, choix de modèle, choix relatifs à la définition des distributions *a priori* ou à l'introduction de tel ou tel jeu de données (si un choix intervient à chacune de ces étapes). Il n'existe pas de méthode systématique pour réaliser cette analyse de sensibilité au choix réalisés. Il s'agit, sur le principe, de réaliser à nouveau l'estimation des paramètres en modifiant les choix réalisés (modification raisonnable des lois *a priori* par exemple) et de quantifier l'impact de ces modifications sur les paramètres estimés.

4.2 Critère d'ajustement et comparaison de modèles

Le critère qui a été le plus classiquement utilisé pour caractériser la qualité de l'ajustement aux données en inférence bayésienne est le DIC (Deviance Information Criterion). Ce n'est pas le seul critère, et son utilisation est criticable, mais c'est le seul fourni par tous les logiciels de type **BUGS**. Il s'agit d'une déviance pénalisée par la complexité du modèle, sorte de généralisation du critère d'Akaïké (AIC : Akaike Information Criterion) adaptée notamment aux modèles mixtes et à l'inférence bayésienne. Rappelons que ces critères d'information (AIC, DIC) ont été construits pour tenter d'approcher une erreur (en terme de déviance) lors de l'utilisation du modèle en prédiction (sur de nouvelles données) et que le terme de pénalisation peut être vu de ce fait comme la correction à apporter pour passer de l'erreur observée en ajustement (optimiste) à l'erreur attendue en prédiction. La fonction `dic.samples` permet de calculer cette déviance avec deux types de pénalisation différentes correspondant à l'argument `type = "pD"` pour la pénalisation classique proposée en 2002 par Spiegelhalter et ses collaborateurs, et à l'argument `type = "popt"` pour la pénalisation proposée en 2008 par Martyn Plummer, l'auteur de **JAGS** (se référer au manuel de référence de **JAGS** pour plus de détails). Voici le code qui permet sur notre exemple de calculer le DIC classique sur 5000 itérations.

```r
dic.samples(model, n.iter = 5000, type = "pD")

## Mean deviance:  5612
## penalty 3.08
## Penalized deviance: 5615
```

Dans le cadre de la comparaison de plusieurs modèles, **JAGS** propose aussi la fonction `diffdic` qui permet de comparer les DIC de deux modèles (cf. `?diffdic`).

Depuis quelques années les auteurs utilisent de plus en plus d'autres critères d'ajustement qui semblent donner une meilleure approximation de l'erreur en prédiction dans le cadre bayésien (cf. Vehtari et al. (2017). Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC in *Statistics and Computing*). Il s'agit du WAIC (Widely applicable information criterion) et du PSIS-loo (approximation du "leave-one-out cross validation" -loo- obtenu par "Pareto smoothed importance sampling" PSIS- d'où son nom). Pour plus de précisions sur les principes et usages de ces méthodes, référez-vous à la documentation du package R `loo` et à l'article précédemment cité. Ces deux critères ne sont pas directement calculés par les outils de type BUGS, mais on peut facilement les calculer à l'aide du package `loo` comme dans l'exemple ci-dessous. La procédure consiste juste :

- à ajouter dans le corps du modèle une ligne définissant la logvraisemblance en chaque point de données observées (sous forme d'un vecteur intitulé ici `loglik`),
- à récupérer la matrice des MCMC sur ce noeud vectoriel à l'aide de `coda.samples()`
- puis à fournir aux fonctions `waic()` ou `loo()` du package `loo` la matrice ainsi récupérée, matrice avec autant de colonnes que le nombre de points de données observées et un nombre de lignes égal au nombre total d'itérations MCMC (soit le nombre d'itérations par chaîne fois le nombre de chaîne).

```r
modelPoisson4loo <-
"model
{
  # Definition des liens
  for (i in 1:n)
```

```
  {
    lambda[i] <- d / (1 + (conc[i]/e)^b)
    Ncumul[i] ~ dpois(lambda[i])
    # Definition de la log-vraisemblance pour alimenter le package loo
    loglik[i] <- log(dpois(Ncumul[i], lambda[i]))
  }

  # Definition des distributions a priori
  log10b ~ dunif(-2,2)
  d ~ dunif(2000, 10000)
  log10e ~ dunif(1, 3)

  b <- pow(10,log10b)
  e <- pow(10,log10e)
}"
```

```
model4loo <- jags.model(file = textConnection(modelPoisson4loo), data = data4jags,
                    inits = ini, n.chains = 3)
update(model4loo, 5000)
mcmc4loo <- coda.samples(model4loo, c("loglik"), n.iter = 5000)
loglik4loo <- as.matrix(mcmc4loo)
```

```
require(loo) # package a installer au prealable
# Calcul du WAIC
(mWAIC <- waic(loglik4loo))

##
## Computed from 15000 by 36 log-likelihood matrix
##
##           Estimate     SE
## elpd_waic  -3096.9  616.0
## p_waic       463.4  117.4
## waic        6193.9 1231.9
##
## 31 (86.1%) p_waic estimates greater than 0.4. We recommend trying loo instead.

# Calcul du PSIS-loo
(mloo <- loo(loglik4loo))

##
## Computed from 15000 by 36 log-likelihood matrix
##
##          Estimate     SE
## elpd_loo  -2998.1  580.9
## p_loo       364.5   79.7
## looic      5996.2 1161.9
## ------
## Monte Carlo SE of elpd_loo is NA.
##
## Pareto k diagnostic values:
##                          Count Pct.    Min. n_eff
## (-Inf, 0.5]   (good)     13    36.1%   633
##  (0.5, 0.7]   (ok)        6    16.7%   208
##    (0.7, 1]   (bad)       6    16.7%   28
##    (1, Inf)   (very bad) 11    30.6%   2
## See help('pareto-k-diagnostic') for details.
```

Dans le cadre de la comparaison de plusieurs modèles, le package `loo` propose aussi la fonction `compare` qui permet de comparer les WAIC ou PSIS-loo de deux modèles (cf. `?compare`).

4.3 Utilisation des MCMC pour la validation interne ou externe

Les chaînes MCMC, que l'on peut facilement récupérer et concaténer (cf. partie 3.4), peuvent être utilisées dans **R** pour simuler la distribution prédictive des données ou de tout autre statistique et construire ainsi des graphes

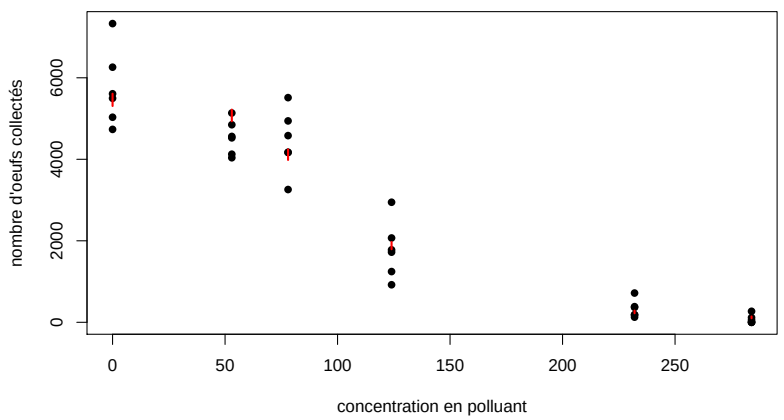
appropriés à chaque cas pour visualiser la qualité d'ajustement du modèle aux données, en représentant les valeurs prédites (issues de la distribution *a posteriori*) par des moyennes, des intervalles de crédibilité ···

Ci-dessous, à titre d'exemple, à l'image de ce qui avait été codé pour prédire la réponse à une concentration donnée, nous avons prédit la réponse aux concentrations testées de façon à confronter visuellement les intervalles de crédibilité des données prédites aux données observées.

```
### Concaténation des valeurs simulées dans les 3 chaînes
### dans un même jeu de données
mcmctot <- as.data.frame(as.matrix(mcmc))
## Récupération sous forme de vecteurs des MCMC pour les paramètres
vb <- mcmctot[, "b"]
vd <- mcmctot[, "d"]
ve <- mcmctot[, "e"]
### Prédiction de la réponse pour les concentrations testées pour chaque jeu de paramètres
vconc <- unique(d$conc) # concentrations testées
l <- length(vconc) # nombre de concentrations testées
## préparation du stockage des quantiles à 2.5, 50 et 97.5 %
## des valeurs simulées à chaque concentration
qinf <- vector(length = l); qmed <- vector(length = l); qsup <- vector(length = l)
## pour chaque concentration simulation et calcul des quantiles
for (i in 1:l)
{
  vlambda <- vd / (1 + (vconc[i]/ve)^vb)
  vNcumul <- rpois(n = length(vlambda), lambda = vlambda)
  qinf[i] <- quantile(vNcumul, probs = 0.025)
  qmed[i] <- quantile(vNcumul, probs = 0.5)
  qsup[i] <- quantile(vNcumul, probs = 0.975)
}
print(cbind(vconc, qinf, qmed, qsup))

##      vconc qinf qmed qsup
## [1,]     0 5308 5459 5610
## [2,]    53 4934 5075 5220
## [3,]    78 3984 4113 4244
## [4,]   124 1793 1881 1971
## [5,]   232  222  254  287
## [6,]   284   99  120  144
```

Pour confronter ces prédictions aux données observées, on peut représenter les données et y ajouter les intervalles de crédibilité à 95% comme ci-dessous (on s'attend à ce que 95% des données soient dans les intervalles de crédibilité).

```
plot(Ncumul ~ conc, data = d, pch = 16, xlab = "concentration en polluant",
     ylab = "nombre d'oeufs collectés")
for (i in 1:l)
{
  segments(vconc[i], qinf[i], vconc[i], qsup[i], col = "red", lwd = 2)
}
```

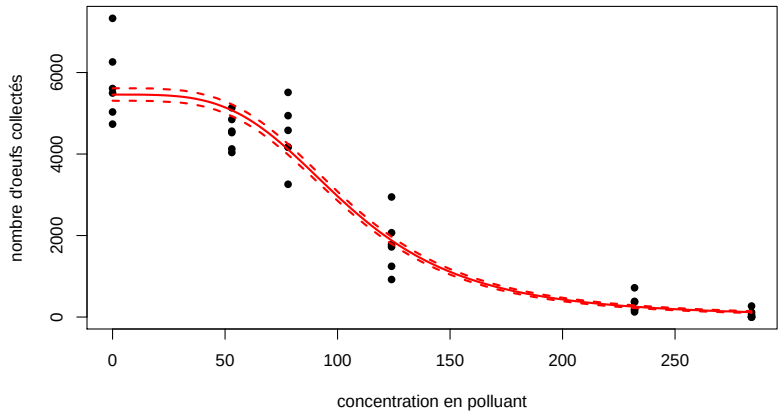


Sur cet exemple on remarque d'emblée que si les intervalles crédibilité sont à peu près au bon endroit (bonne adéquation de la partie déterministe du modèle aux données), ils sont trop petits pour recouvrir 95% des points, du

fait d'une dispersion inter-bécher non décrite par le modèle.

On peut aussi dans cet exemple tracer une bande de confiance et une courbe médiane estimée, en faisant le même calcul non plus sur les seules concentrations testées mais sur une séquence de concentrations définies sur la gamme des concentrations testées.

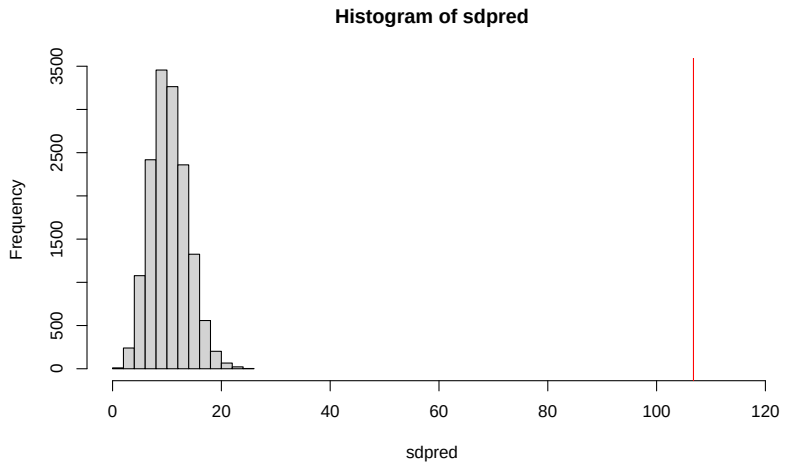
```
### Concaténation des valeurs simulées dans les 3 chaînes
### dans un même jeu de données
mcmctot <- as.data.frame(as.matrix(mcmc))
## Récupération sous forme de vecteurs des MCMC pour les paramètres
vb <- mcmctot[, "b"]
vd <- mcmctot[, "d"]
ve <- mcmctot[, "e"]
### Prédiction de la réponse sur la gamme des concentrations testées pour chaque jeu de paramètres
## définition de la série de concentrations sur la gamme
l <- 100 # nombre de concentrations souhaitées
vconc <- seq(min(d$conc), max(d$conc), length.out = l)
## préparation du stockage des quantiles à 2.5, 50 et 97.5 %
## des valeurs simulées à chaque concentration
qinf <- vector(length = l); qmed <- vector(length = l); qsup <- vector(length = l)
## pour chaque concentration simulation et calcul des quantiles
for (i in 1:l)
{
  vlambda <- vd / (1 + (vconc[i]/ve)^vb)
  vNcumul <- rpois(n = length(vlambda), lambda = vlambda)
  qinf[i] <- quantile(vNcumul, probs = 0.025)
  qmed[i] <- quantile(vNcumul, probs = 0.5)
  qsup[i] <- quantile(vNcumul, probs = 0.975)
}
plot(Ncumul ~ conc, data = d, pch = 16, xlab = "concentration en polluant",
     ylab = "nombre d'oeufs collectés")
lines(vconc, qinf, col = "red", lwd = 2, lty = 2)
lines(vconc, qmed, col = "red", lwd = 2, lty = 1)
lines(vconc, qsup, col = "red", lwd = 2, lty = 2)
```



Dans cet exemple ces représentations suffisent pour mettre en évidence le problème de surdispersion non décrite par le modèle, mais sur d'autres exemples il pourrait être intéressant de prédire une statistique calculable sur les données ou une partie des données et de confonter la distribution *a posteriori* prédictive de cette statistique à sa valeur observée sur les données. Pour donner un exemple de ce genre de calcul, ci-dessous est représentée la distribution *a posteriori* prédictive de l'écart type pour les six béchers à la concentration maximale testée (sous-jeu de données à $conc = 284$) ainsi que sa valeur observée dans le jeu de données.

```
### Concaténation des valeurs simulées dans les 3 chaînes
### dans un même jeu de données
mcmctot <- as.data.frame(as.matrix(mcmc))
## Récupération sous forme de vecteurs des MCMC pour les paramètres
vb <- mcmctot[, "b"]
vd <- mcmctot[, "d"]
ve <- mcmctot[, "e"]
### Valeur observée de l'écart type des réponses à une concentration fixée
```

```r
concfixee <- max(d$conc)
sdobs <- sd(subset(d, conc == concfixee)$Ncumul)
### Pour chaque jeu de paramètres, prédiction de la réponse pour six béchers
### et de la variance l'écart type des six réponses
## préparation du stockage des écarts types prédits
nsimu <- nrow(mcmctot)
sdpred <- vector(length = nsimu)
## pour chaque jeu de paramètres, simulation de six réponses et calcul de leur écart type
for (j in 1:nsimu)
{
  lambda <- vd[j] / (1 + (concfixee / ve[j])^vb[j])
  vNcumul <- rpois(n = 6, lambda = lambda)
  sdpred[j] <- sd(vNcumul)
}
# représentation de la distribution prédictive de cet écart type
hist(sdpred, xlim = c(0, max(sdobs, max(sdpred)) * 1.1 ))
# ajout de la valeur observée
abline(v = sdobs, col = "red")
```



Cette représentation graphique montre clairement que même pour la concentration la plus forte le modèle sous-estime la variabilité entre les points observés, la valeur observée de l'écart type sortant nettement de sa distribution prédictive. Sur cet exemple tous les résultats de validation mettent en évidence une non-prise en compte de la sur-dispersion qui serait à corriger par exemple en choisissant un modèle de type négative-binomiale plutôt que de type Poisson pour la partie stochastique.

Si l'on dispose de nouvelles données n'ayant pas servi à l'inférence bayésienne, et donc permettant une validation externe, il est tout aussi facile d'utiliser les chaînes MCMC pour simuler la distribution prédictive de ces nouvelles données et de construire des critères statistiques ou des graphes appropriés pour juger des résultats en validation externe.

5 Annexe : rudiments pour le codage d'un modèle avec JAGS

*La description qui suit ne vise qu'à donner un aperçu du langage utilisé pour déclarer un modèle dans **JAGS** et vous trouverez des informations beaucoup plus complètes et précises dans le manuel de référence de **JAGS** dont l'adresse internet est donnée en introduction.*

5.1 Quelques fonctions utilisables

— **Fonctions de base**

```
y <- abs(x)
y <- exp(x)
y <- log(x)
y <- pow(x,z) (x à la puissance z que l'on peut aussi écrire x^z dans JAGS)
y <- sqrt(x) (racine carrée de x)
y <- round(x)
y <- min(x1,x2)
y <- equals(x1,x2) (vaut 1 si x1 = x2, 0 sinon)
y <- step(x) (vaut 0 si x < 0, 1 sinon)
y <- ifelse(x, a, b)} (vaut a si condition x vérifiée, b sinon)
```

— **Fonctions vectorielles**

```
y <- mean(v[])
y <- sd(v[])
y <- sum(v[])
Y[,] <- inverse(M[,])
y <- ranked(v[],k) (utile pour calculer minv <- ranked(v[],1) ou maxv <- ranked(v[],n) )
```

— **Fonctions de lien pouvant être utilisées à gauche**

```
logit(y) <-
probit(y) <-
log(y) <-
```

5.2 Quelques distributions disponibles

En ce qui concerne les distributions, voici les noms de quelques distributions classiques et la présentation des fonctions permettant de définir des troncatures et des censures :

— **distributions univariées**

```
x ~ dbern(p) (Bernouilli)
x ~ dbin(p, n) (binomiale)
x ~ negbin(p, r) (négative binomiale caratérisant le nb. d'échecs avant r succès
                  pour une binomiale de probabilité p)
x ~ dpois(lambda) (Poisson)
x ~ dcat(p[]) (catégorie tirée lors d'un tirage unique dans une loi multinomiale de paramètre p[])
x ~ dnorm(mu,tau) (normale, ATTENTION, paramétrée en moyenne et précision : tau = 1/variance)
x ~ dt(mu, tau, k) (Student)
x ~ dlnorm(mu, tau) (lognormale avec mu et tau les moyenne et précision de ln(x))
x ~ dbeta(alpha, beta) (bêta)
x ~ dunif(a, b) (uniforme)
x ~ dexp(rate) (exponentielle)
x ~ dgamma(shape, rate) (avec rate = 1/scale)
```

— **distributions multivariées**

```
x[1:k] ~ dmulti(p[],n) (multinomiale)
x[1:k] ~ ddirch(a[]) (Dirichlet)
x[1:k] ~ dmnorm(mu[],T[,]) (multinormale avec T l'inverse de la matrice de variance-covariance)
```

5.3 Troncature et censure

— **troncature d'une distribution**

Pour spécifier une loi tronquée sur l'intervalle $[L, U]$ on fait suivre la loi de `T(L,U)`
Ex. : loi normale tronquée sur $[0, +\infty]$:

```
x ~ dnorm(mu,tau)T(0,)
```

— **description de données censurées**

Pour décrire des données censurées dans **JAGS**, on peut utiliser `dinterval`. En pratique on déclare deux variables dans les données, l'une décrivant classiquement les données avec la valeur `NA` pour toutes les valeurs censurées, et une deuxième variable indicatrice de la censure. Par exemple, pour une censure à droite (type données de survie), on indiquera la valeur 0 pour une valeur non censurée, dont on sait qu'elle est au-dessous du seuil de censure, et 1 pour une valeur censurée, dont on sait qu'elle est au-dessus du seuil de censure. Cette deuxième variable sera déclarée dans le modèle à l'aide de la distribution `dinterval`. Dans l'exemple ci-dessous, `censure` prend la valeur 0 si $y \leq c$ et 1 sinon.

```
censure ~ dinterval(y, c)
```

Il est aussi possible de déclarer des censures par intervalle à l'aide de cette fonction. Dans l'exemple suivant, en supposant que le vecteur `c[]` est de dimension 2 et contient les valeurs c1 et c2, censure prend la valeur 0 si $y \leq c1$, 1 si $c1 < y \leq c2$ et 2 si $y > c2$.

```
censure ~ dinterval(y, c[])
```