Data Cloning Workshop, Budapest 2024

Subhash Lele and Peter Solymos

2024-08-28

Preliminaries

In the first part of the course, we will go over some statistical preliminaries and corresponding computational aspects. We will learn:

1. To write down a likelihood function
2. Meaning of the likelihood function
3. Meaning of the Maximum likelihood estimator, difference between a parameter, an estimator and an estimate
4. Good properties of an estimator: Consistency, Asymptotic normality, Unbiasedness, low Mean squared error
5. Frequentist paradigm and quantification of uncertainty
6. How to use Fisher information for an approximate quantification of uncertainty
7. Motivation for the Bayesian paradigm
8. Meaning of the prior distribution
9. Derivation and meaning of the posterior distribution
10. Interpretation of a credible interval and a confidence interval
11. Scope of inference: Should I specify the hypothetical experiment or should I specify the prior distribution? Each one comes with its own scope of inference.

Occupancy studies

Let us start with an occupancy study. Suppose we have a site that is being considered for development. There is a species of interest that might get affected by this development. Hence we need to study what proportion of the area is occupied by the species of interest. If this proportion is not very large, we may go ahead with the development.

Suppose we divide the site in several equal-area cells. Suppose all cells have similar habitats (*identical*). Further we assume that occupancy of one cell does not affect occupancy of other quadrats (*independence*). Let N be the total number of cells.

Let Y_i be the occupancy status of the i-th quadrat. This is unknown and hence is a random variable. It takes values in (0,1), 0 meaning unoccupied and 1 meaning occupied. This is a *Bernoulli* random variable. We denote this by $Y_i \sim Bernoulli(\phi)$. The random variable Y takes value 1 with probability ϕ. This is the probability of occupancy. The value of ϕ is unknown. This is the parameter of the distribution.

Suppose we visit n, a subset, of these cells. These are selected using simple random sampling without replacement. The observations are denoted by $y_1, y_2, y_3, ..., y_n$. We can use these to infer about the unknown parameter ϕ. The main tool for such inductive inference (data to population and *not* hypothesis to prediction) is the *likelihood function.*

The likelihood function Suppose the observed data are (0,1,1,0,0). Then we can compute the probability of observing these data under various values of the parameter ϕ (assuming independent, identically distributed Bernoulli random variables). It can be written as: $L(\phi; y_1, y_2, ..., y_n) = \prod P(y_i; \phi) = \prod \phi^{y_i}(1-\phi)^{1-y_i}$

Notice that this is a function of the parameter ϕ and the data are fixed. The likelihood function or equivalently the log-likelihood function quantifies the *relative* support for different values of the parameters. Hence only the likelihood ratio function is meaningful.

Maximum likelihood estimator A natural approach to estimation (inference) of ϕ is to choose the value that is better supported than any other value in the parameter space $(0, 1)$. This is called the maximum likelihood estimator. We can show that this turns out to be: $\hat{\phi} = \frac{1}{n}\sum y_i$ This is called an *estimate*. This is a fixed quantity because the data are observed and hence not random.

Quantification of uncertainty As scientists, would you stop at reporting this? I suspect not. If this estimate is large, say 0.85, the developer is going to say 'you just got lucky (or, worse, you cheated) with your particular sample'. A natural question to ask then is 'how different this estimate would have been if someone else had conducted the experiment?'. In this case, the 'experiment to be repeated' is fairly uncontroversial. We take another simple random sample without replacement from the study area. However, that is not always the case as we will see when we deal with the regression model.

The sampling distribution is the distribution of the estimates that one would have obtained had one conducted these replicate experiments. It is possible to get an approximation to this sampling distribution in a very general fashion if we use the method of maximum likelihood estimator. In many situations, it can be shown that the sampling distribution is $\hat{\phi} \sim N(\phi, \frac{1}{n}I^{-1}(\phi))$ where $I(\phi) = -\frac{1}{n}\sum \frac{d^2}{d^2\phi} logL(\phi; y)$

This is also called the Hessian matrix or the curvature matrix of the log-likelihood function. The higher the curvature, the less variable are the estimates from one experiment to other. Hence the resultant 'estimate' is considered highly reliable.

95% Confidence interval This is just a set of values that covers the estimates from 95% of the experiments. The experiments are not actually replicated and hence this simply tells us what the various outcomes could be. Our decisions could be based on this variation *as long as we all agree on the experiment that could be replicated*. We are simply covering our bases against the various outcomes and protect ourselves from future challenges. If we use the maximum likelihood estimator, we can obtain this as: $\hat{\phi} - \frac{1.96}{n}\sqrt{I^{-1}(\hat{\phi})}, \hat{\phi} + \frac{1.96}{n}\sqrt{I^{-1}(\hat{\phi})}$ You will notice that as we increase the sample size, the width of this interval converges to zero. That is, as we increase the sample size, the MLE converges to the true parameter value. This is called the 'consistency' of an estimator. This is an essential property of any statistical inferential procedure.

Bayesian paradigm

All the above statements seem logical but fake at the same time! No one repeats the same experiment (although replication consistency is an essential scientific requirement). What if we have time series? We can never replicate a time series. So then should we simply take the estimated value *prima facie*? That also seems incorrect scientifically. So where is the uncertainty in our mind coming from? According to the Bayesian paradigm, it arises because of our 'personal' uncertainty about the parameter values.

Prior distribution: Suppose we have some idea about what values of occupancy are more likely than others *before* any data are collected. This can be quantified as a probability distribution on the parameter space (0,1). This distribution can be anything, unimodal or bimodal or even multimodal! Let us denote this by $\pi(\phi)$. How do we change this *after* we observe the data?

Posterior distribution This is the quantification of uncertainty *after* we observe the data. Usually observing the data decreases our uncertainty, although it is not guaranteed to be the case. The posterior distribution is obtained by: $\pi(\phi|y) = \frac{L(\phi;y)\pi(\phi)}{\int L(\phi;d\phi y)\pi(\phi)}$

Credible interval This is obtained by using the percentiles of the posterior distribution.

Notice a few things here:

1. This involves an integral in the denominator. Depending on how many parameters (unknowns) are in the model, this can be a large dimensional integral. Imagine a regression model with 5 covariates. This integral will be 6 dimensional (add one for the variance).
2. Data are fixed. We do not need to replicate the experiment. The uncertainty is completely in the mind of the researcher.
3. Different researchers might have different prior uncertainties. This will lead to different posterior uncertainties. Hence this is a subjective or personal quantification of uncertainty. It is not transferable from one researcher to another.

An interesting result follows. As we increase the samples size, the Bayesian posterior, for ANY prior, converges to the distribution that looks very much like the frequentist sampling distribution of the MLE. That is, $\pi(\phi|y) \approx N(\hat{\phi}, \frac{1}{n} I^{-1}(\hat{\phi}))$ There are subtle differences that we are going to ignore here. Qualitatively, what this says is that for large sample size:

1. Posterior mean and the MLE are similar
2. Posterior variance is similar to the inverse of the Hessian matrix.

Hence credible interval and confidence intervals will be indistinguishable for large sample size. Effect of the choice of the prior distribution vanishes. How large a sample size should be for this to happen? It depends on the number of parameters in the model and how strong the prior distribution is.

No math please!

Bayesian and ML inference using MCMC and data cloning

We now show how one can compute the posterior distribution for any choice of the prior distribution without analytically calculating the integral in the denominator. We will generate the data under the Bernoulli model. You can change the parameters as you wish when you run the code.

Simulate a simple data set with 30 observations:

```
library(dclone)
```

```
## Loading required package: coda
```

```
## Loading required package: parallel
```

```
## Loading required package: Matrix
```

```
## dclone 2.3-2      2023-07-02
```

```
phi.true = 0.3
n = 30
Y = rbinom(n,1,phi.true)

table(Y)
```

```
## Y
##  0  1
## 20 10
```

Analytical MLE:

```
(MLE.est = sum(Y)/n)
```

```
## [1] 0.3333333
```

Bayesian inference We will use the JAGS program and the dclone R package.

First, we need to define the model function. This is the critical component.

```
Occ.model = function(){
  # Likelihood
  for (i in 1:n){
    Y[i] ~ dbin(phi_occ, 1)
  }
  # Prior
  phi_occ ~ dbeta(1, 1)
}
```

Second, we need to provide the data to the model and generate random numbers from the posterior. We will discuss different options later.

```
dat = list(Y=Y, n=n)
Occ.Bayes = jags.fit(data=dat, params="phi_occ", model=Occ.model)
```

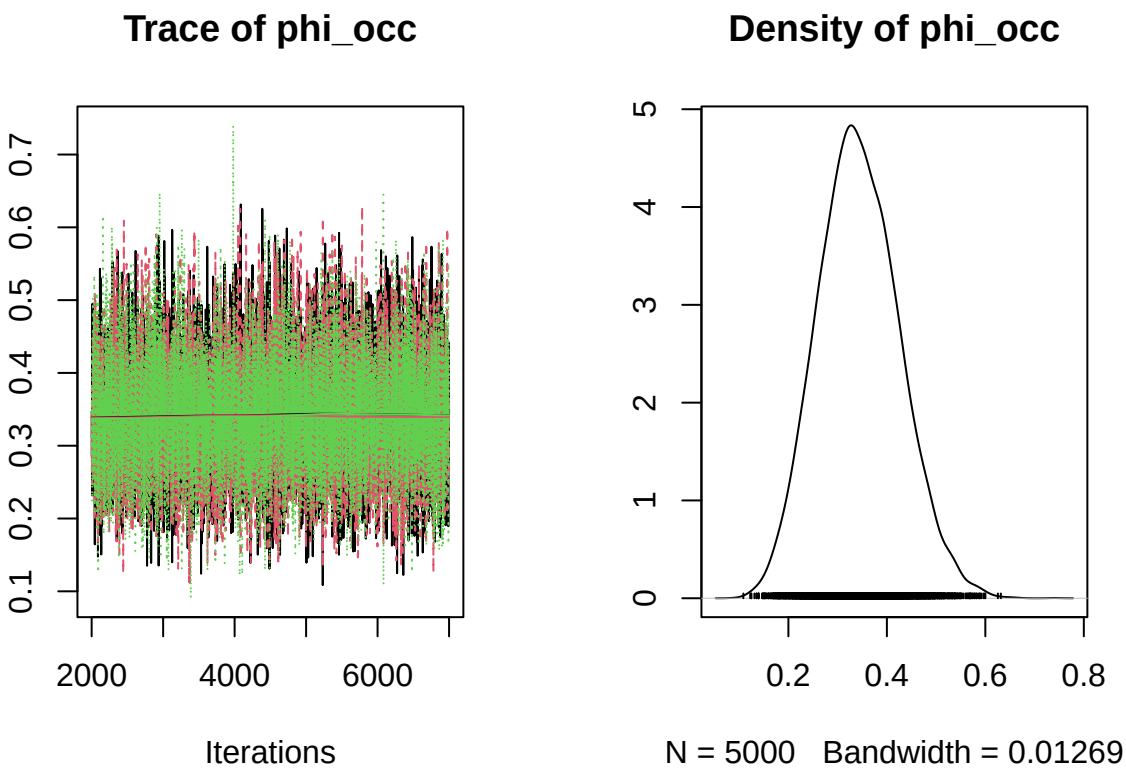
```
## Registered S3 method overwritten by 'R2WinBUGS':
##   method            from
##   as.mcmc.list.bugs dclone
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 1
##    Total graph size: 33
##
## Initializing model
```

```
summary(Occ.Bayes)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
##      0.3431606      0.0819165      0.0006688      0.0008493
##
## 2. Quantiles for each variable:
##
##   2.5%    25%    50%    75%  97.5%
## 0.1917 0.2861 0.3395 0.3973 0.5116
```

```
plot(Occ.Bayes)
```



The summary describes the posterior distribution: its mean, standard deviation, and quantiles.

This was quite easy. Now we use data cloning to compute the MLE and its variance using MCMC.

Data cloning in a nutshell

As you all know, at least in this simple situation, we can write down the likelihood function analytically. We can also use calculus and/or numerical optimization such as the `optim()` function in R to get the location of the maximum and its Hessian matrix. But suppose we do not want to go through all of that and instead want to use the MCMC algorithm. Why? Because it is easy and can be generalized to more complex hierarchical models.

Earlier we noted that as we increase the sample size, the Bayesian posterior converges to the sampling distribution of the MLE. We, obviously, cannot increase the sample size. The data are given to us. Data cloning conducts a computational trick to increase the sample size. We clone the data!

Imagine a sequence of K independent researchers.

- Step 1: First researcher has data $y_1, y_2, ..., y_n$. They use their own prior and obtain the posterior distribution.
- Step 2: Second researcher goes out and gets their own data. It just so happens that they observed the same exact locations as the first researcher. Being a good Bayesian, they use the posterior of the first researcher as their prior (knowledge accumulation).
- Step K: The K-th researcher also obtains the same data but uses the posterior at the (K-1) step as their prior.

What is happening with these sequential posterior distributions?

The posterior distribution is converging to a single point; a degenerate distribution. This is identical to the MLE!

1. As we increase the number of clones, the mean of the posterior distributions converges to the MLE.
2. The variance of the posterior distribution converges to 0.

3. If we scale the posterior distribution with the number of clones (that is, multiply the posterior variance by the number of clones), it is identical to the inverse of the Fisher information matrix.

You can play with the number of clones and see the effect on the posterior distribution using this Shiny app (R code for the app)

We do not need to implement this procedure sequentially. The matrix of these K datasets is of dimension (n,K) with identical columns.

$$\begin{bmatrix} y_1 & y_1 & y_1 & y_1 & y_1 & y_1 & y_1 & y_1 & y_1 & y_1 \\ y_2 & y_2 & y_2 & y_2 & y_2 & y_2 & y_2 & y_2 & y_2 & y_2 \\ y_3 & y_3 & y_3 & y_3 & y_3 & y_3 & y_3 & y_3 & y_3 & y_3 \\ y_4 & y_4 & y_4 & y_4 & y_4 & y_4 & y_4 & y_4 & y_4 & y_4 \\ y_5 & y_5 & y_5 & y_5 & y_5 & y_5 & y_5 & y_5 & y_5 & y_5 \end{bmatrix}$$

We use the Bayesian procedure to analyze these data. The model function used previously can be used with a minor modification to do this.

```
Occ.model.dc = function(){
  # Likelihood
  for(k in 1:ncl){
    for (i in 1:n){
      Y[i,k] ~ dbin(phi_occ, 1)
    }
  }
  # Prior
  phi_occ ~ dbeta(1, 1)
}
```

To match this change in the model, we need to turn the original data into an array.

```
Y = array(Y, dim=c(n, 1))
Y = dcdim(Y)
```

When defining the data, we need to add another index `ncl` for the cloned dimension. It gets multiplied by the number of clones.

```
dat = list(Y=Y, n=n, ncl=1)
# 2 clones of the Y array
dclone(Y, 2)
```

```
##       clone.1 clone.2
##  [1,]       1       1
##  [2,]       0       0
##  [3,]       0       0
##  [4,]       0       0
##  [5,]       1       1
##  [6,]       0       0
##  [7,]       1       1
##  [8,]       1       1
##  [9,]       0       0
## [10,]       0       0
## [11,]       0       0
## [12,]       0       0
## [13,]       0       0
## [14,]       0       0
## [15,]       0       0
## [16,]       1       1
```

```
## [17,]       0       0
## [18,]       1       1
## [19,]       1       1
## [20,]       0       0
## [21,]       1       1
## [22,]       1       1
## [23,]       0       0
## [24,]       0       0
## [25,]       0       0
## [26,]       0       0
## [27,]       0       0
## [28,]       0       0
## [29,]       0       0
## [30,]       1       1
## attr(,"n.clones")
## [1] 2
## attr(,"n.clones")attr(,"method")
## [1] "dim"
## attr(,"n.clones")attr(,"method")attr(,"drop")
## [1] TRUE
```

```r
# 2 clones of the data list - this is not what we want
dclone(dat, 2)
```

```
## $Y
##       clone.1 clone.2
##  [1,]       1       1
##  [2,]       0       0
##  [3,]       0       0
##  [4,]       0       0
##  [5,]       1       1
##  [6,]       0       0
##  [7,]       1       1
##  [8,]       1       1
##  [9,]       0       0
## [10,]       0       0
## [11,]       0       0
## [12,]       0       0
## [13,]       0       0
## [14,]       0       0
## [15,]       0       0
## [16,]       1       1
## [17,]       0       0
## [18,]       1       1
## [19,]       1       1
## [20,]       0       0
## [21,]       1       1
## [22,]       1       1
## [23,]       0       0
## [24,]       0       0
## [25,]       0       0
## [26,]       0       0
## [27,]       0       0
## [28,]       0       0
## [29,]       0       0
```

```
## [30,]       1       1
## attr(,"n.clones")
## [1] 2
## attr(,"n.clones")attr(,"method")
## [1] "dim"
## attr(,"n.clones")attr(,"method")attr(,"drop")
## [1] TRUE
##
## $n
## [1] 30 30
## attr(,"n.clones")
## [1] 2
## attr(,"n.clones")attr(,"method")
## [1] "rep"
##
## $ncl
## [1] 1 1
## attr(,"n.clones")
## [1] 2
## attr(,"n.clones")attr(,"method")
## [1] "rep"
```

Notice that this changes `n` also. We do not want that, we want to keep `n` unchanged.

```
dclone(dat, 2, unchanged="n", multiply="ncl")
```

```
## $Y
##       clone.1 clone.2
##  [1,]       1       1
##  [2,]       0       0
##  [3,]       0       0
##  [4,]       0       0
##  [5,]       1       1
##  [6,]       0       0
##  [7,]       1       1
##  [8,]       1       1
##  [9,]       0       0
## [10,]       0       0
## [11,]       0       0
## [12,]       0       0
## [13,]       0       0
## [14,]       0       0
## [15,]       0       0
## [16,]       1       1
## [17,]       0       0
## [18,]       1       1
## [19,]       1       1
## [20,]       0       0
## [21,]       1       1
## [22,]       1       1
## [23,]       0       0
## [24,]       0       0
## [25,]       0       0
## [26,]       0       0
## [27,]       0       0
```

```
## [28,]       0       0
## [29,]       0       0
## [30,]       1       1
## attr(,"n.clones")
## [1] 2
## attr(,"n.clones")attr(,"method")
## [1] "dim"
## attr(,"n.clones")attr(,"method")attr(,"drop")
## [1] TRUE
##
## $n
## [1] 30
##
## $ncl
## [1] 2
## attr(,"n.clones")
## [1] 2
## attr(,"n.clones")attr(,"method")
## [1] "multi"
```

The `dc.fit` function takes the familiar arguments to determine how to clone the data list.

```
Occ.DC = dc.fit(data=dat, params="phi_occ", model=Occ.model.dc,
  n.clones=c(1, 2, 5), unchanged="n", multiply="ncl")
```

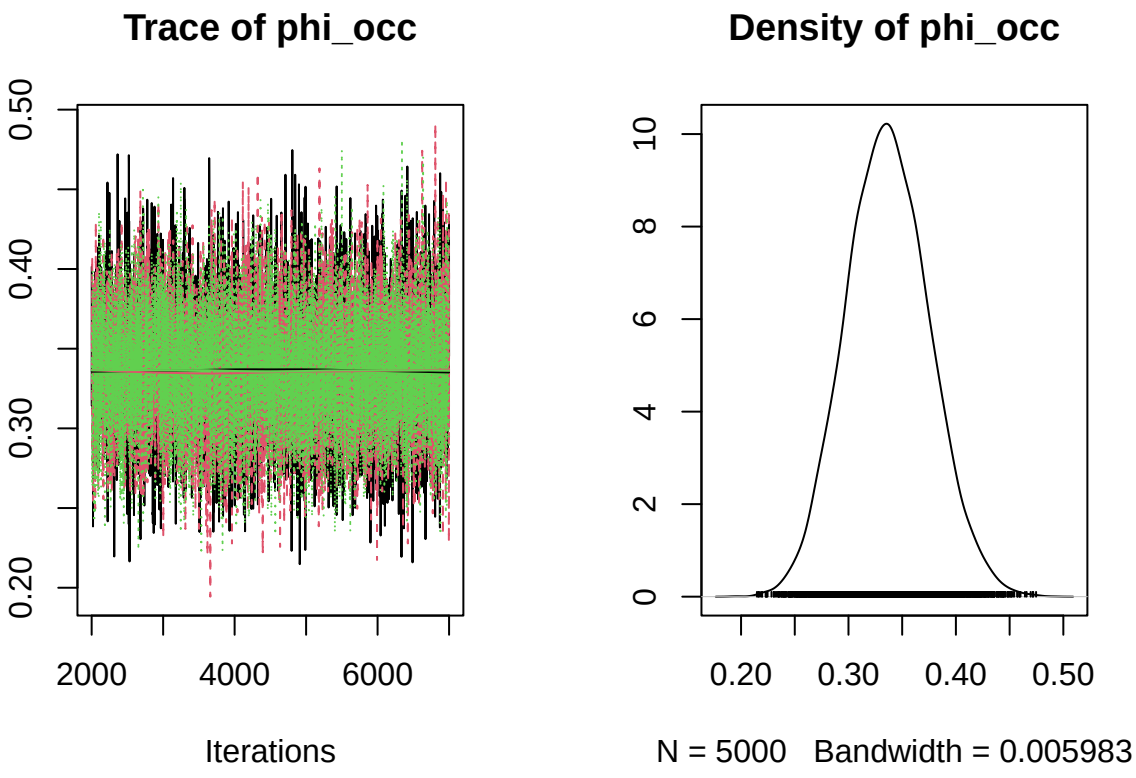
```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 1
##    Total graph size: 34
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 1
##    Total graph size: 64
##
## Initializing model
##
##
## Fitting model with 5 clones
##
## Compiling model graph
```

```
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 150
##    Unobserved stochastic nodes: 1
##    Total graph size: 154
##
## Initializing model
```

```
summary(Occ.DC)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean      SD   DC SD  Naive SE Time-series SE R hat
## phi_occ 0.3366 0.03862 0.08636 0.0003153      0.0004054     1
##
## 2. Quantiles for each variable:
##
##   2.5%    25%    50%    75%  97.5%
## 0.2637 0.3098 0.3359 0.3625 0.4142
```

```
plot(Occ.DC)
```



Notice the `Mean`, `DC SD`, and `R hat` columns in the summary. These refer to the maximum likelihood estimate, the asymptotic standatd error (SD of the posterior times square root of K), and the Gelman-Rubin disgnostic:

```
coef(Occ.DC) # MLE
```

```
##   phi_occ
## 0.3365614
```

```
dcsd(Occ.DC) # SE=SD*sqrt(K)
```

```
##    phi_occ
## 0.08635965
## attr(,"method")
##       Y       n     ncl
##   "dim"      NA "multi"
```

```
gelman.diag(Occ.DC) # R hat
```

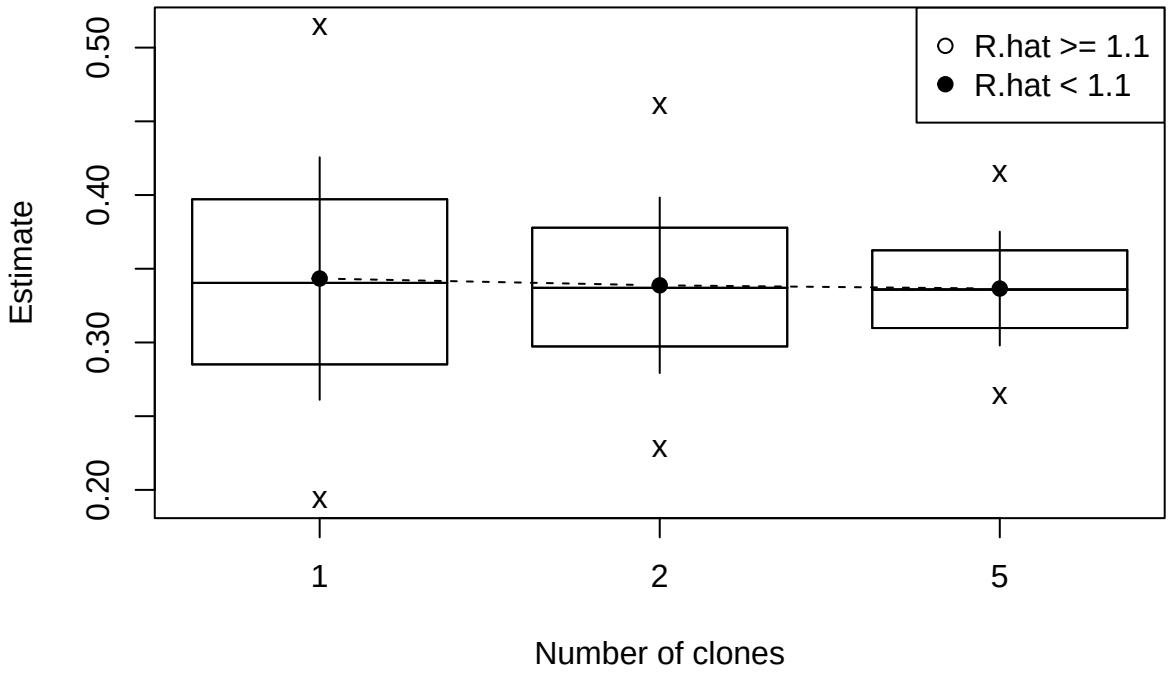
```
## Potential scale reduction factors:
##
##         Point est. Upper C.I.
## phi_occ          1          1
```

Summaries for the clones Summaries of the posterior distributions for the different numbers of clones are saved and we can print these out with the `dctable()` command. We can visualize these with the `plot` function.

```
dctable(Occ.DC)
```

```
## $phi_occ
##   n.clones      mean         sd      2.5%       25%       50%       75%
## 1        1 0.3433713 0.08224185 0.1936630 0.2851346 0.3403916 0.3971257
## 2        2 0.3387504 0.05952716 0.2279913 0.2973051 0.3370137 0.3778473
## 3        5 0.3365614 0.03862121 0.2636994 0.3097723 0.3358995 0.3624977
##       97.5%    r.hat
## 1 0.5144200 1.000054
## 2 0.4608554 1.000360
## 3 0.4142404 1.000332
##
## attr(,"class")
## [1] "dctable"
```

```
dctable(Occ.DC) |> plot()
```

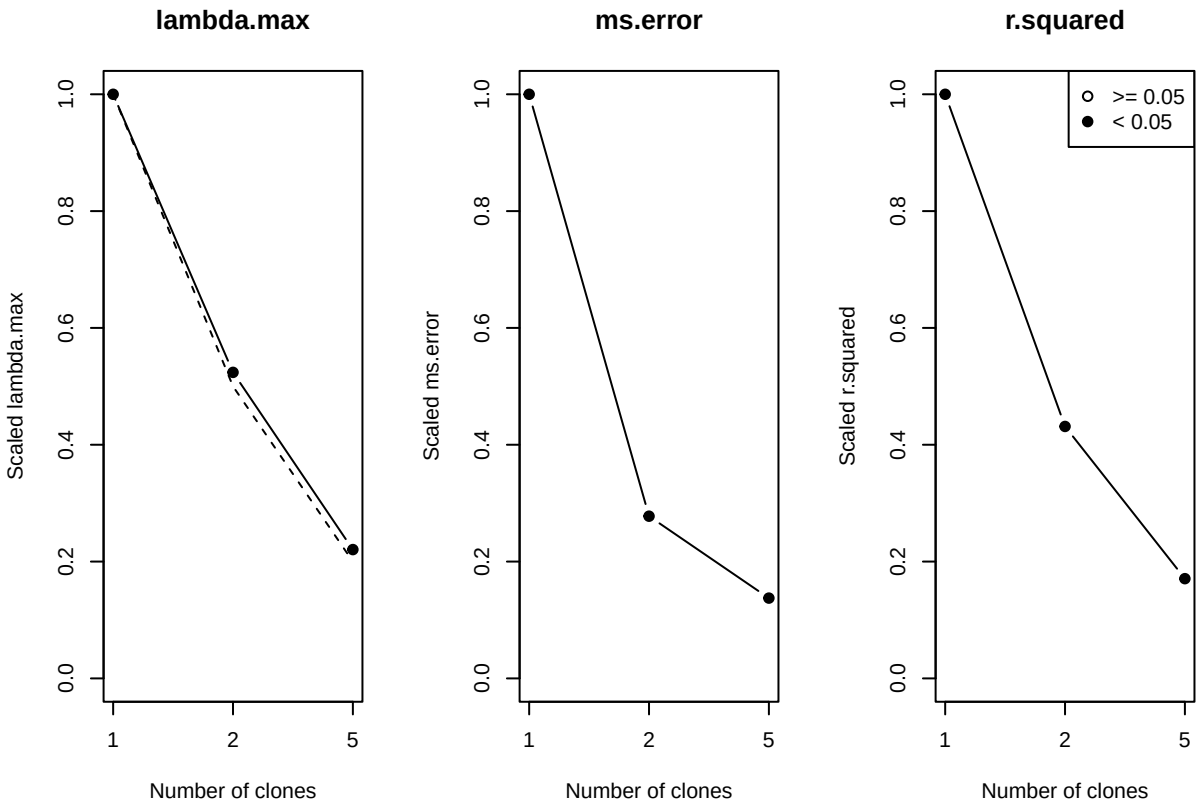


Some data cloning related diagnostics are printed with the `dcdiag()` function. We will discuss these statistics in detail. The most important thing to ckeck is that the solid line follows the scattered line for `lambda.max`, i.e. decreases with the number of clones.

```
dcdiag(Occ.DC)
```

```
##   n.clones  lambda.max     ms.error    r.squared    r.hat
## 1        1 0.006763722 0.0039481528 0.0011996862 1.000054
## 2        2 0.003543483 0.0010962912 0.0005174709 1.000360
## 3        5 0.001491598 0.0005427004 0.0002048375 1.000332
```

```
dcdiag(Occ.DC) |> plot()
```



Regression models

Now we will generalize these models to account for covariates. We will consider Logistic regression but also comment on how to change it to Probit regression easily. Similarly we show how this basic prototype can be modified to do linear and non-linear regression, Poisson regression etc.

```
n = 30 # sample size
X1 = rnorm(n) # a covariate
X = model.matrix(~X1)
beta.true = c(0.5, 1)
link_mu = X %*% beta.true # logit scale
```

```
phi_occ = plogis(link_mu) # prob scale
Y = rbinom(n, 1, phi_occ)
```

Logistic regression model Maximum likelihood estimate using `glm()`:

```
MLE.est = glm(Y ~ X1, family="binomial")
```

Bayesian analysis

```
Occ.model = function(){
  # Likelihood
  for (i in 1:n){
    phi_occ[i] <- ilogit(X[i,] %*% beta)
    Y[i] ~ dbin(phi_occ[i], 1)
  }
  # Prior
  beta[1] ~ dnorm(0, 1)
```

```
  beta[2] ~ dnorm(0, 1)
}
```

Now we need to provide the data to the model and generate random numbers from the posterior. We will discuss different options later.

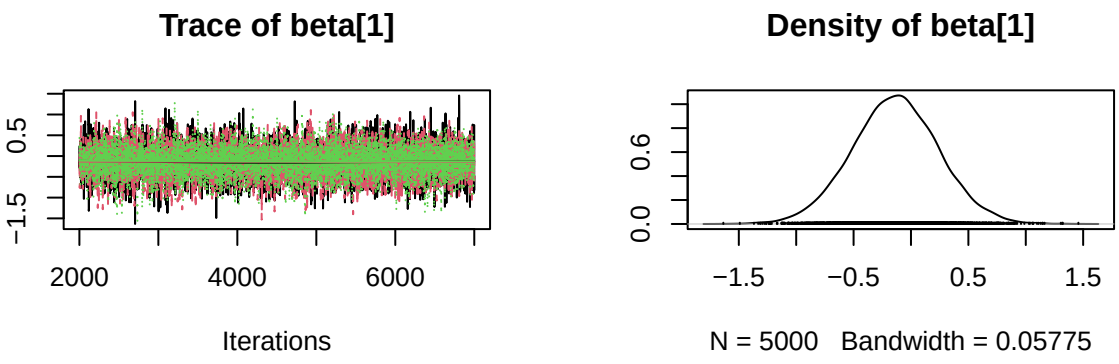
```
dat = list(Y=Y, X=X, n=n)
Occ.Bayes = jags.fit(data=dat, params="beta", model=Occ.model)
```

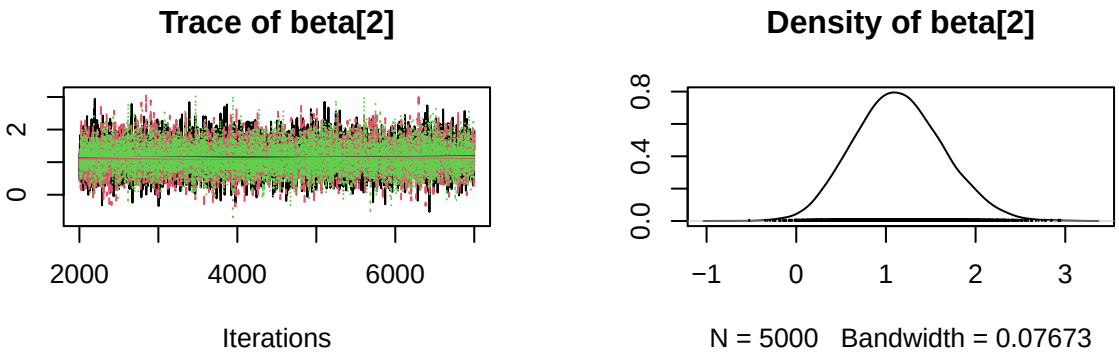
```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 2
##    Total graph size: 186
##
## Initializing model
```

```
summary(Occ.Bayes)
```

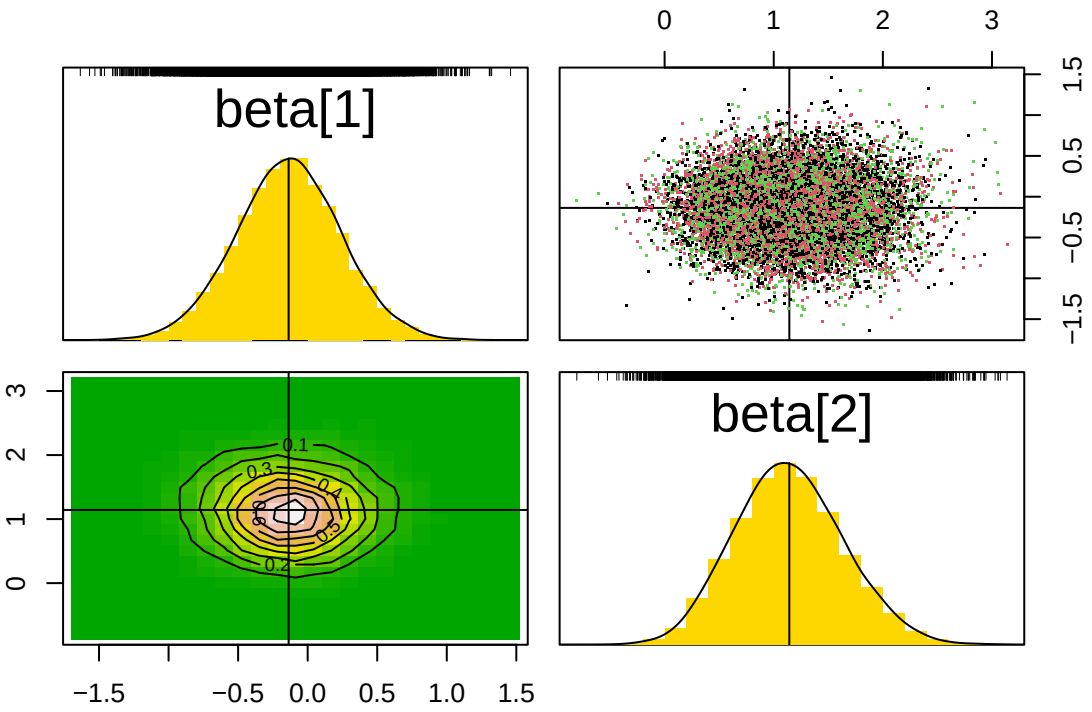
```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##            Mean     SD Naive SE Time-series SE
## beta[1] -0.1369 0.3774 0.003081       0.003893
## beta[2]  1.1428 0.4953 0.004044       0.005062
##
## 2. Quantiles for each variable:
##
##            2.5%     25%     50%    75%  97.5%
## beta[1] -0.8833 -0.3850 -0.1345 0.1145 0.6103
## beta[2]  0.2263  0.8008  1.1281 1.4702 2.1496
```

```
plot(Occ.Bayes)
```






```
pairs(Occ.Bayes)
```



Now we modify this to get the MLE using data cloning.

```
Occ.model_dc = function(){
  # Likelihood
  for (k in 1:ncl){
```

```
    for (i in 1:n){
      phi_occ[i,k] <- ilogit(X[i,,k] %*% beta)
      Y[i,k] ~ dbin(phi_occ[i,k],1)
    }
  }
  # Prior
  beta[1] ~ dnorm(0, 1)
  beta[2] ~ dnorm(0, 1)
}
```

Now we need to provide the data to the model and generate random numbers from the posterior. We will discuss different options later.

```
Y = array(Y, dim=c(n, 1))
X = array(X, dim=c(dim(X), 1))
# clone the objects
Y = dcdim(Y)
X = dcdim(X)
```

Data cloning with `dc.fit()`:

```
dat = list(Y=Y, X=X, n=n, ncl=1)
Occ.DC = dc.fit(data=dat, params="beta", model=Occ.model_dc,
  n.clones=c(1, 2, 5), unchanged="n", multiply="ncl")
```

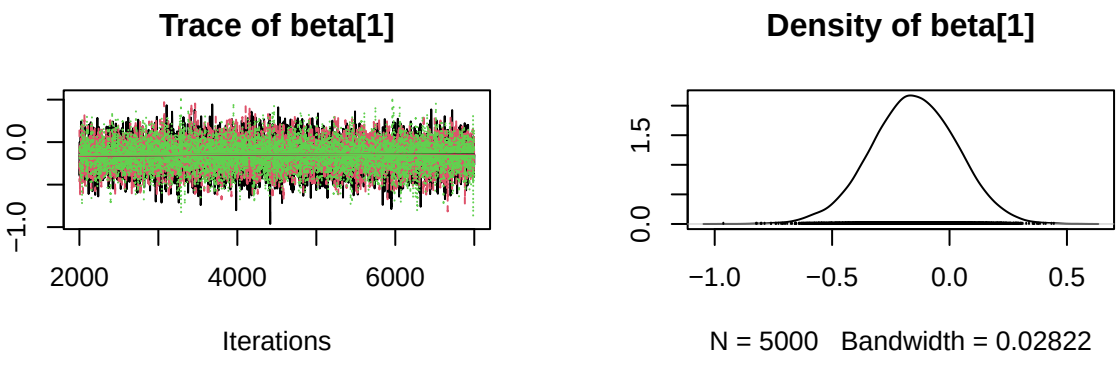
```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 2
##    Total graph size: 187
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 2
##    Total graph size: 307
##
## Initializing model
##
##
## Fitting model with 5 clones
##
## Compiling model graph
```

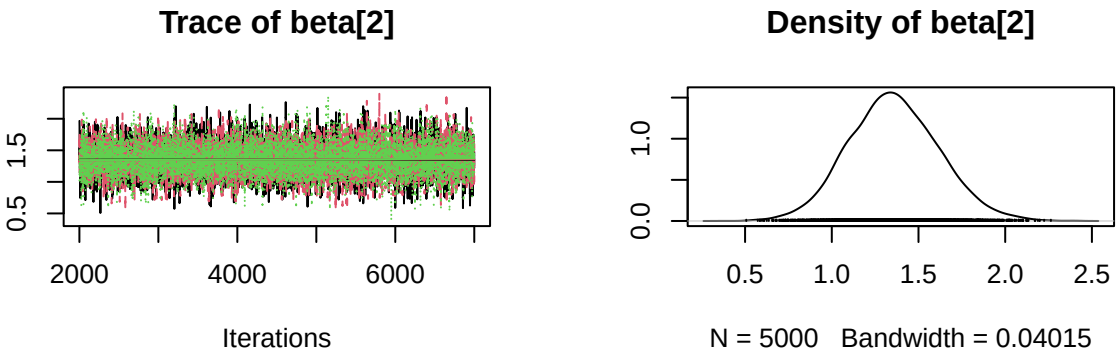
```
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 150
##    Unobserved stochastic nodes: 2
##    Total graph size: 667
##
## Initializing model
```

```
summary(Occ.DC)
```

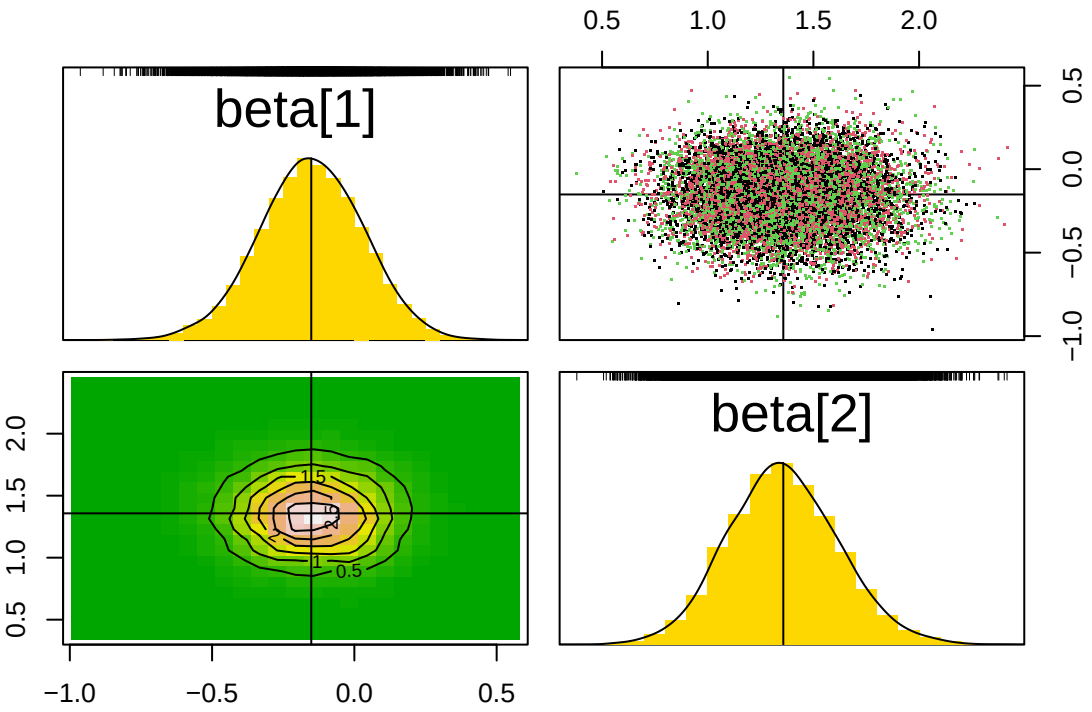
```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##             Mean     SD  DC SD Naive SE Time-series SE  R hat
## beta[1] -0.1515 0.1822 0.4073 0.001487       0.001927 1.0004
## beta[2]  1.3575 0.2600 0.5813 0.002123       0.002734 0.9999
##
## 2. Quantiles for each variable:
##
##             2.5%     25%     50%      75%  97.5%
## beta[1] -0.5162 -0.2722 -0.1512 -0.02728 0.2004
## beta[2]  0.8639  1.1813  1.3505  1.52865 1.8855
```

```
plot(Occ.DC)
```






```
pairs(Occ.DC)
```



These are the familiar functions for the asymptotic ML inference (already accounted for the number of clones):

```
coef(Occ.DC) # MLE
```

```
##    beta[1]    beta[2]
## -0.1515147  1.3574579
```

```
dcsd(Occ.DC) # SE
```

```
##   beta[1]   beta[2]
## 0.4073368 0.5812738
```

```
vcov(Occ.DC) # asymptotic VCV
```

```
##              beta[1]      beta[2]
## beta[1]  0.165923267 -0.005751071
## beta[2] -0.005751071  0.337879228
```

```
confint(Occ.DC, level=0.95) # 95% CI
```

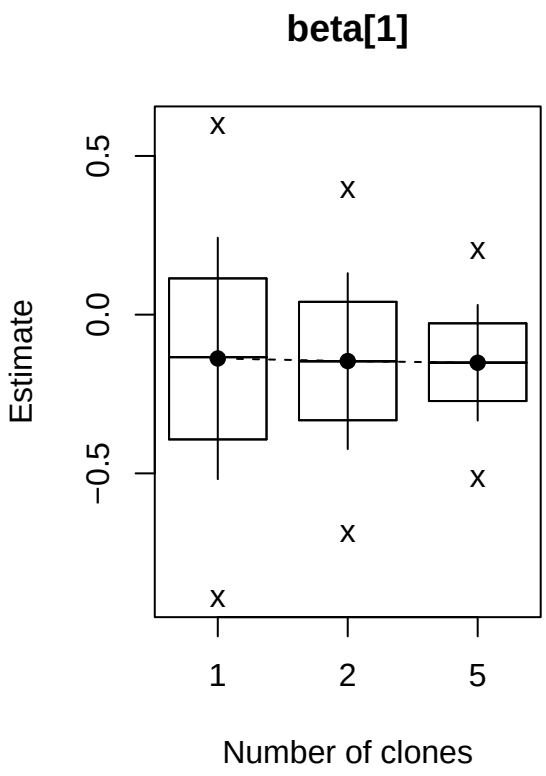
```
##              2.5 %    97.5 %
## beta[1] -0.9498802 0.6468507
## beta[2]  0.2181822 2.4967336
```

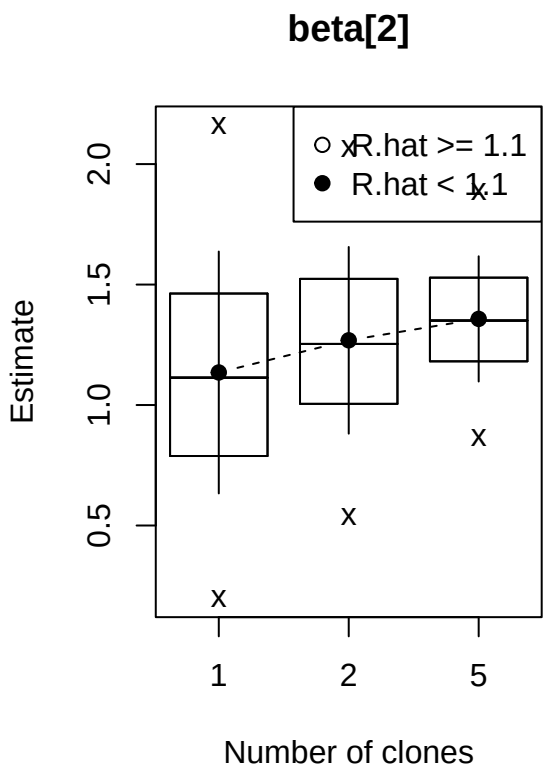
Let's check the posterior summaries and the DC diagnostics:

```
dctable(Occ.DC)
```

```
## $`beta[1]`
##   n.clones       mean        sd       2.5%        25%        50%         75%
## 1        1 -0.1380019 0.3803022 -0.8933828 -0.3930772 -0.1338343  0.11431202
## 2        2 -0.1462633 0.2770016 -0.6889135 -0.3326610 -0.1467354  0.04047121
## 3        5 -0.1515147 0.1821666 -0.5161824 -0.2721661 -0.1511719 -0.02728267
##       97.5%    r.hat
## 1 0.5963905 1.000862
## 2 0.3954406 1.000286
## 3 0.2004307 1.000356
##
## $`beta[2]`
##   n.clones     mean        sd      2.5%       25%      50%      75%    97.5%
## 1        1 1.135142 0.5019435 0.1981546 0.7883859 1.113452 1.462805 2.160731
## 2        2 1.268619 0.3873851 0.5386114 1.0048909 1.253505 1.523686 2.063604
## 3        5 1.357458 0.2599535 0.8639387 1.1813366 1.350491 1.528647 1.885507
##       r.hat
## 1 1.0000614
## 2 1.0004209
## 3 0.9999269
##
## attr(,"class")
## [1] "dctable"
```

```
dctable(Occ.DC) |> plot()
```

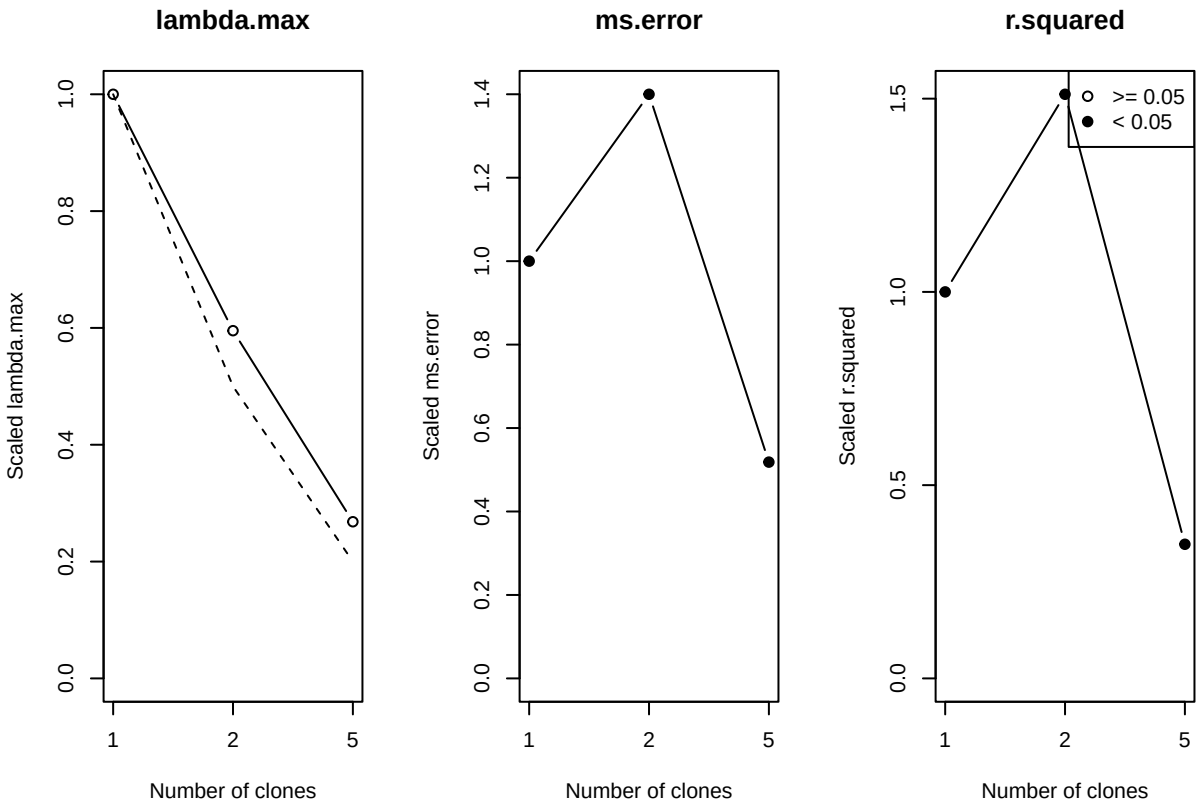





```
dcdiag(Occ.DC)
```

```
##   n.clones lambda.max    ms.error    r.squared     r.hat
## 1        1 0.25214933 0.015746958 0.0028628004 1.0003312
## 2        2 0.15009519 0.022044967 0.0043266123 1.0002885
## 3        5 0.06761427 0.008162313 0.0009934283 0.9999034
```

```
dcdiag(Occ.DC) |> plot()
```



We hope you can see the pattern in how we are changing the prototype model function and the data function. If we want to do a Normal linear regression and Poisson regression we can modify the regression program above easily.

Linear regression The following section issustrates Gaussian linear regression.

```
n = 30
X1 = rnorm(n)
X = model.matrix(~X1)
beta.true = c(0.5, 1)
link_mu = X %*% beta.true

# Linear regression model
mu = link_mu
sigma.e = 1
Y = rnorm(n,mean=mu,sd=sigma.e)

# MLE
MLE.est = glm(Y ~ X1, family="gaussian")

# Bayesian analysis
Normal.model = function(){
  # Likelihood
  for (i in 1:n){
    mu[i] <- X[i,] %*% beta
    Y[i] ~ dnorm(mu[i],prec.e)
  }
  # Prior
  beta[1] ~ dnorm(0, 1)
```

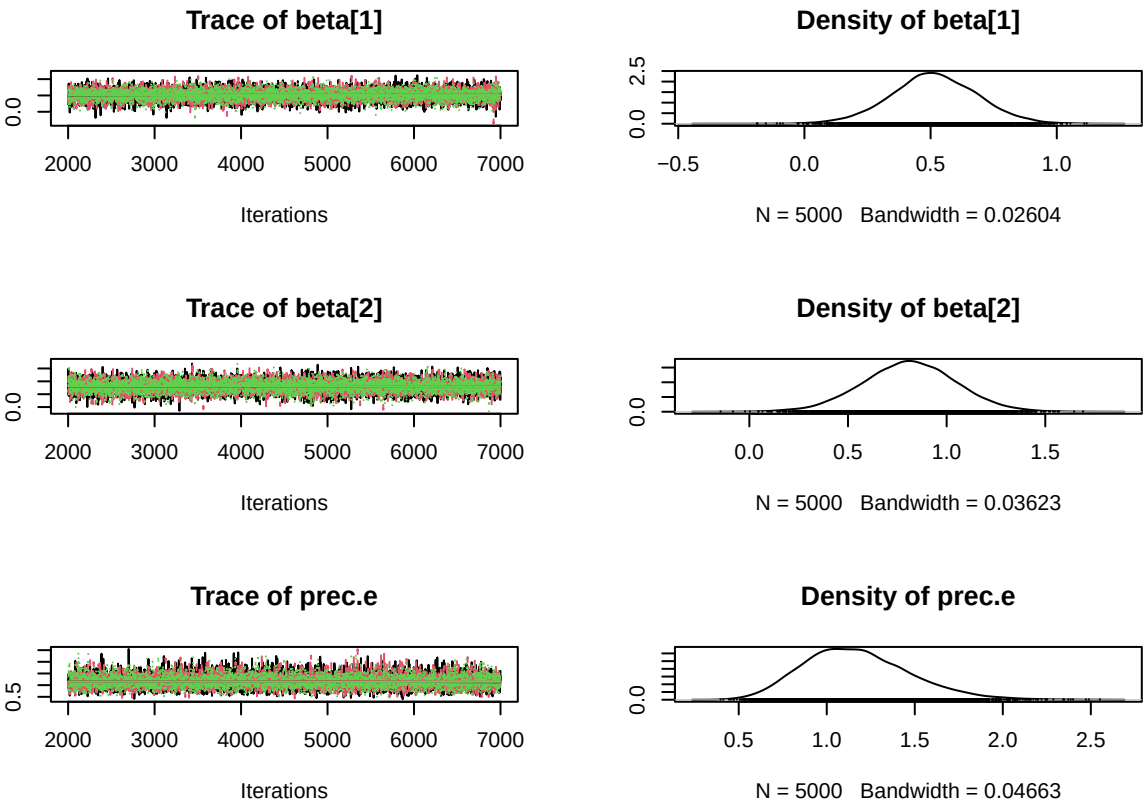
```
  beta[2] ~ dnorm(0, 1)
  prec.e ~ dlnorm(0, 1)
}
dat = list(Y=Y, X=X, n=n)
Normal.Bayes = jags.fit(data=dat, params=c("beta","prec.e"), model=Normal.model)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 3
##    Total graph size: 157
##
## Initializing model
```

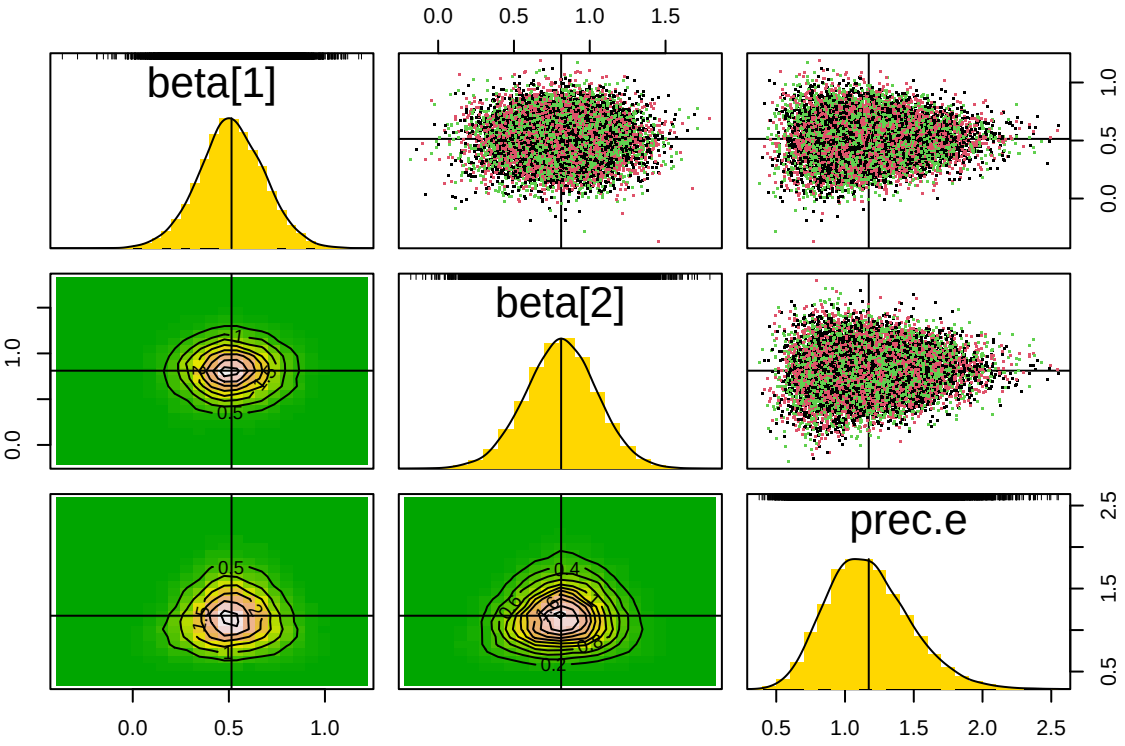
```
summary(Normal.Bayes)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## beta[1] 0.5140 0.1714 0.001399       0.001399
## beta[2] 0.8113 0.2373 0.001937       0.001937
## prec.e  1.1727 0.3010 0.002458       0.003435
##
## 2. Quantiles for each variable:
##
##           2.5%    25%    50%    75%  97.5%
## beta[1] 0.1735 0.4030 0.5126 0.6283 0.8539
## beta[2] 0.3412 0.6555 0.8139 0.9688 1.2728
## prec.e  0.6608 0.9568 1.1483 1.3619 1.8279
```

```
plot(Normal.Bayes)
```




```
pairs(Normal.Bayes)
```




```
# MLE using data cloning.
Normal.model_dc = function(){
  # Likelihood
```

```
  for (k in 1:ncl){
    for (i in 1:n){
      mu[i,k] <- X[i,,k] %*% beta
      Y[i,k] ~ dnorm(mu[i,k],prec.e)
    }
  }
  # Prior
  beta[1] ~ dnorm(0, 1)
  beta[2] ~ dnorm(0, 1)
  prec.e ~ dlnorm(0, 1)
}
Y = array(Y, dim=c(n, 1))
X = array(X, dim=c(dim(X), 1))
Y = dcdim(Y)
X = dcdim(X)
dat = list(Y=Y, X=X, n=n, ncl=1)
Normal.DC = dc.fit(data=dat, params=c("beta","prec.e"), model=Normal.model_dc,
  n.clones=c(1, 2, 5), unchanged="n", multiply="ncl")
```

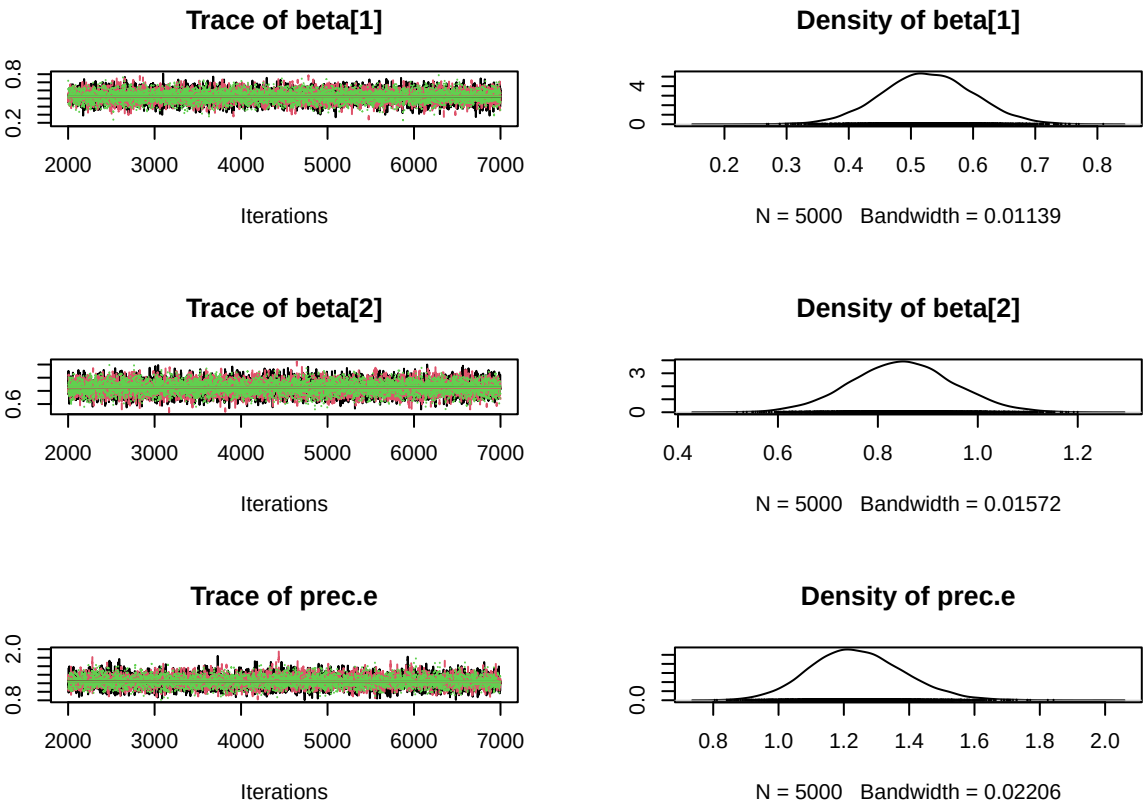
```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 3
##    Total graph size: 158
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 3
##    Total graph size: 278
##
## Initializing model
##
##
## Fitting model with 5 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 150
##    Unobserved stochastic nodes: 3
```

```
##    Total graph size: 638
##
## Initializing model
```

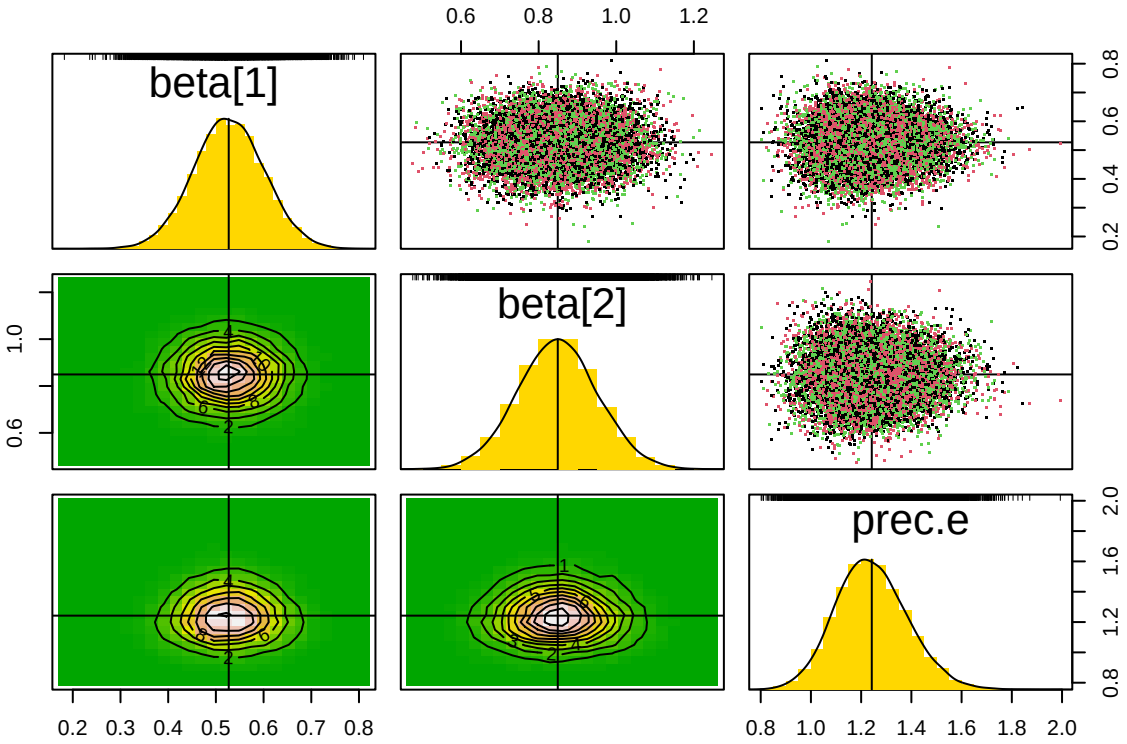
```
summary(Normal.DC)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean      SD  DC SD  Naive SE Time-series SE  R hat
## beta[1] 0.5269 0.07358 0.1645 0.0006007      0.0006008 0.9999
## beta[2] 0.8493 0.10237 0.2289 0.0008359      0.0008358 1.0001
## prec.e  1.2423 0.14239 0.3184 0.0011626      0.0014825 1.0004
##
## 2. Quantiles for each variable:
##
##           2.5%    25%    50%    75%  97.5%
## beta[1] 0.3826 0.4777 0.5261 0.5762 0.6722
## beta[2] 0.6481 0.7804 0.8489 0.9164 1.0517
## prec.e  0.9784 1.1430 1.2349 1.3342 1.5352
```

```
plot(Normal.DC)
```



```
pairs(Normal.DC)
```



Poisson log-link regression We will now modify the code to conduct count data regression using the Poisson distribution and log-link.

```
n = 30
X1 = rnorm(n)
X = model.matrix(~X1)
beta.true = c(0.5, 1)
link_mu = X %*% beta.true

# Log-linear regression model
mu = exp(link_mu)
Y = rpois(n, mu)

# MLE
MLE.est = glm(Y ~ X1, family="poisson")

# Bayesian analysis
Poisson.model = function(){
  # Likelihood
  for (i in 1:n){
    mu[i] <- exp(X[i,] %*% beta)
    Y[i] ~ dpois(mu[i])
  }
  # Prior
  beta[1] ~ dnorm(0, 1)
  beta[2] ~ dnorm(0, 1)
}
dat = list(Y=Y, X=X, n=n)
```

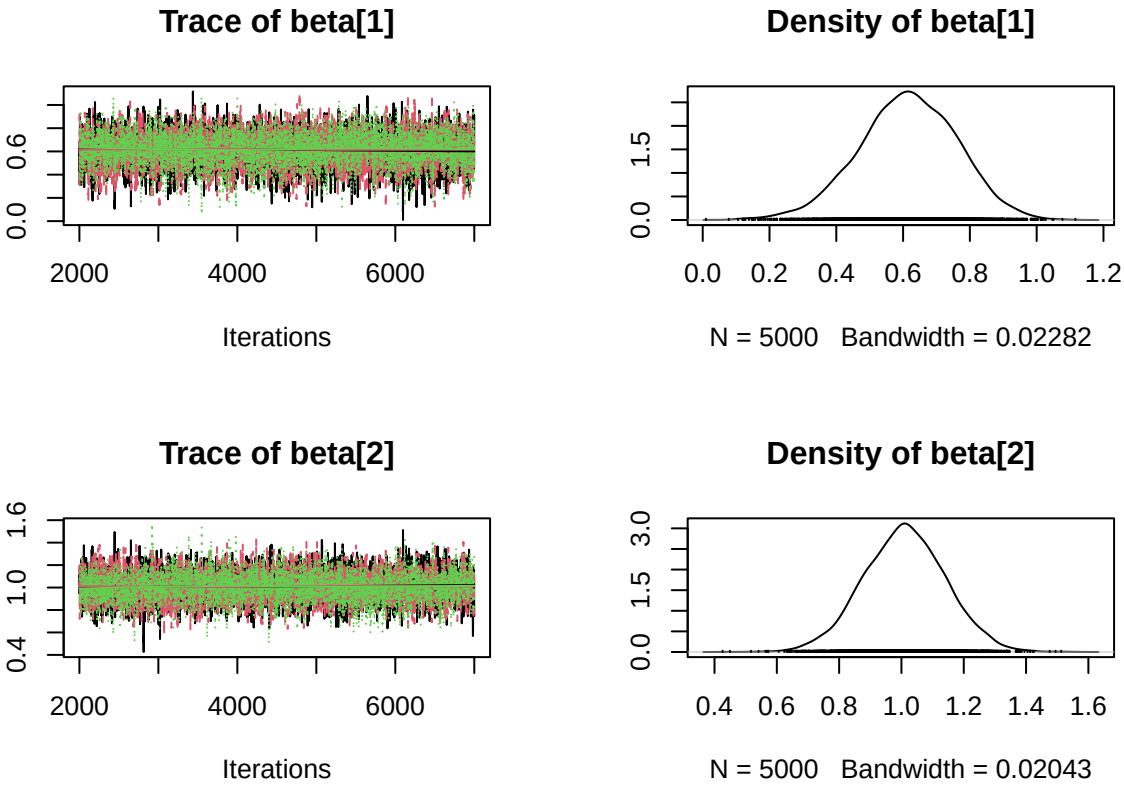
```
Poisson.Bayes = jags.fit(data=dat, params="beta", model=Poisson.model)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 2
##    Total graph size: 186
##
## Initializing model
```

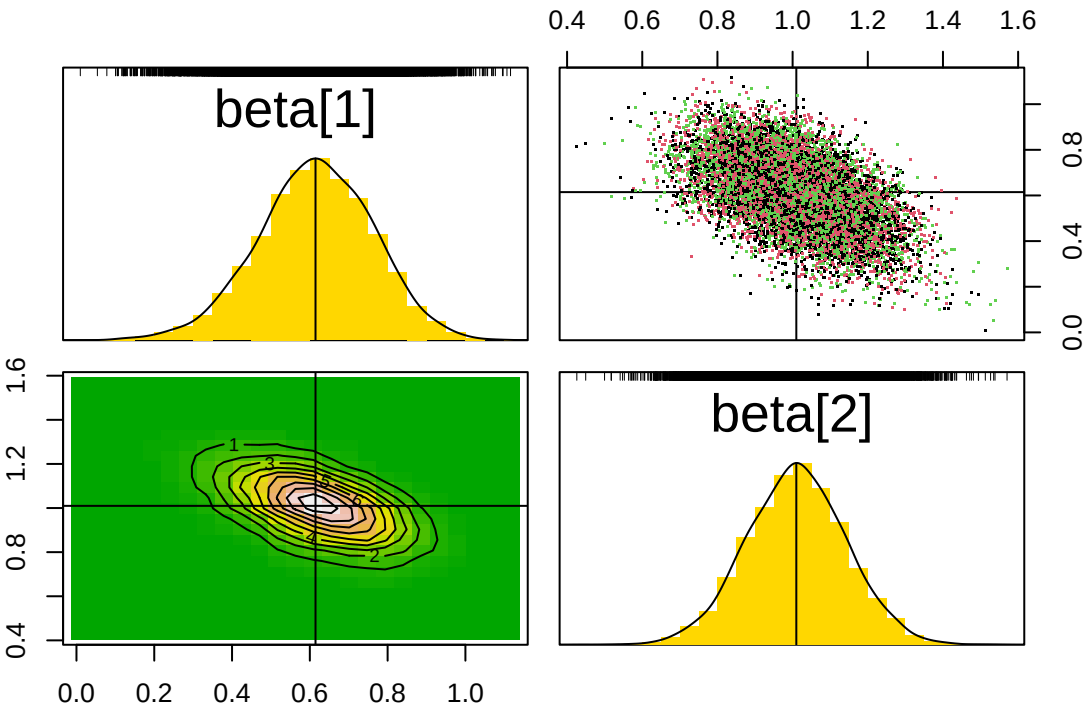
```
summary(Poisson.Bayes)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## beta[1] 0.6146 0.1473 0.001203       0.002101
## beta[2] 1.0099 0.1320 0.001078       0.001857
##
## 2. Quantiles for each variable:
##
##           2.5%    25%    50%    75%  97.5%
## beta[1] 0.3139 0.5198 0.6171 0.7176 0.8934
## beta[2] 0.7483 0.9211 1.0104 1.0978 1.2703
```

```
plot(Poisson.Bayes)
```




```
pairs(Poisson.Bayes)
```




```
# MLE using data cloning
Poisson.model_dc = function(){
  # Likelihood
  for (k in 1:ncl){
    for (i in 1:n){
```

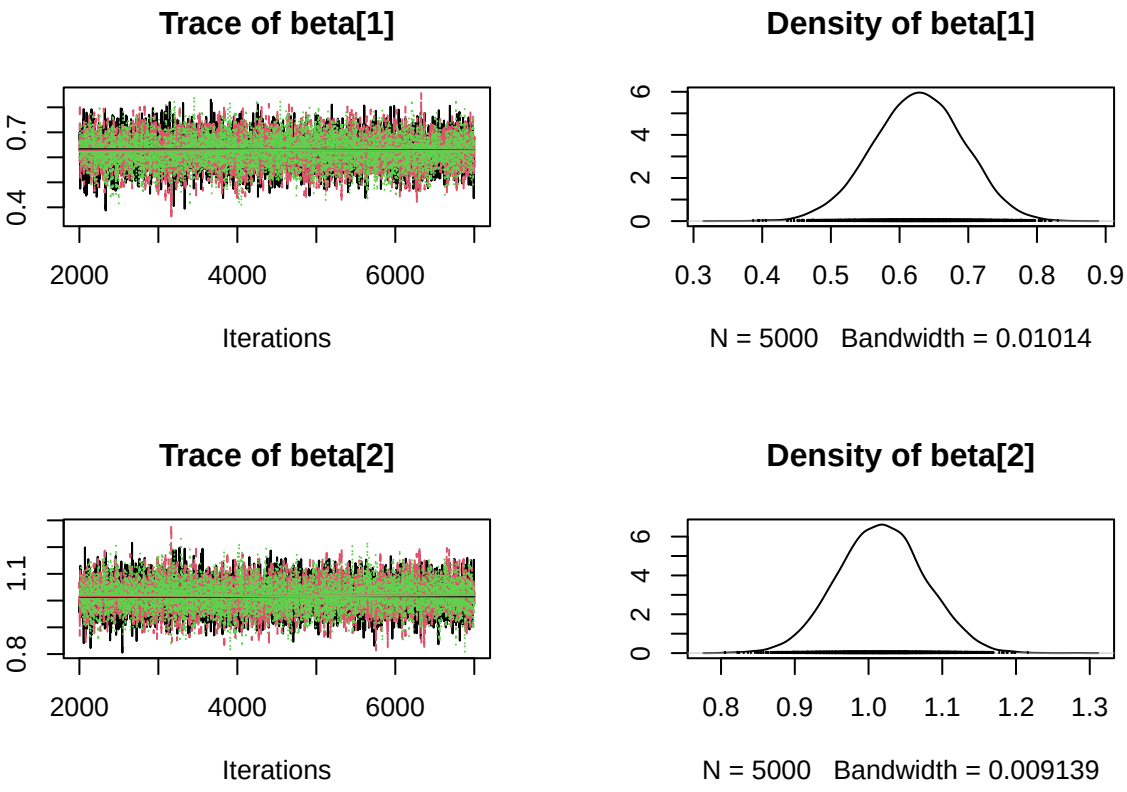
```
      mu[i,k] <- exp(X[i,,k] %*% beta)
      Y[i,k] ~ dpois(mu[i,k])
    }
  }
  # Prior
  beta[1] ~ dnorm(0, 1)
  beta[2] ~ dnorm(0, 1)
}
Y = array(Y, dim=c(n, 1))
X = array(X, dim=c(dim(X), 1))
Y = dcdim(Y)
X = dcdim(X)
dat = list(Y=Y, X=X, n=n, ncl=1)
Poisson.DC = dc.fit(data=dat, params="beta", model=Poisson.model_dc,
  n.clones=c(1, 2, 5), unchanged="n", multiply="ncl")
```

```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 2
##    Total graph size: 187
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 2
##    Total graph size: 307
##
## Initializing model
##
##
## Fitting model with 5 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 150
##    Unobserved stochastic nodes: 2
##    Total graph size: 667
##
## Initializing model
```

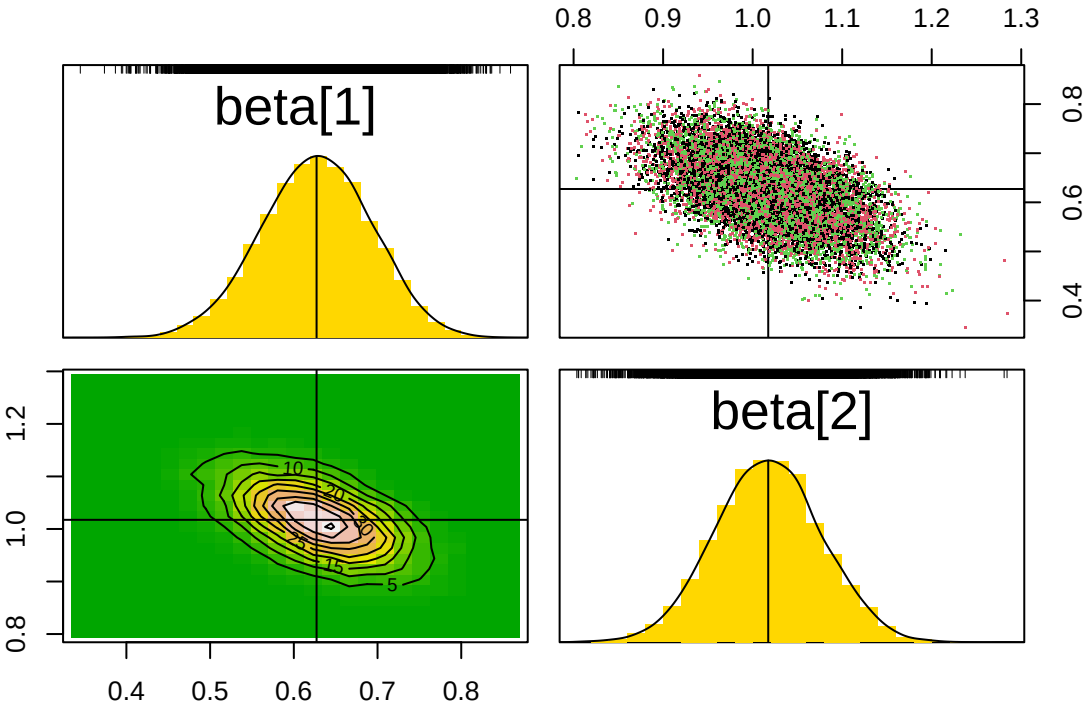
```
summary(Poisson.DC)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean      SD  DC SD  Naive SE Time-series SE R hat
## beta[1] 0.6272 0.06546 0.1464 0.0005345      0.0009067     1
## beta[2] 1.0174 0.05910 0.1321 0.0004825      0.0008153     1
##
## 2. Quantiles for each variable:
##
##           2.5%    25%   50%    75%  97.5%
## beta[1] 0.4966 0.5833 0.628 0.6721 0.7522
## beta[2] 0.9018 0.9774 1.017 1.0564 1.1328
```

```
plot(Poisson.DC)
```




```
pairs(Poisson.DC)
```



Why use MCMC based Bayesian and data cloning?

1. Writing the model function is much more intuitive than writing the likelihood function, prior, etc.
2. Do not need to do numerical integration or numerical optimization.
3. Data cloning overcomes multimodality of the likelihood function. Entire prior distribution essentially works as a set of starting values. In the usual optimization, starting values can be quite important when the function is not well behaved. By using data cloning, except for the global maximum, all the local maxima tend to 0.
4. Asymptotic variance of the MLE is simple to obtain. It is also more stable than computing the inverse of the second derivative of the log-likelihood function numerically.

Mixed Models

In the previous part, we reviewed the basic statistical concepts behind the likelihood inference and the Bayesian inference. We looked at how to write a JAGS model function for some linear and generalized linear regression models and use it in the package 'dclone' to get the Bayesian credible intervals as well as the frequentist confidence intervals based on the asymptotic normal distribution. We also discussed some of the reasons to use the MCMC approach to conducting statistical inference, either Bayesian or frequentist.

We will now generalize the models to make them relevant to some complex practical situations. These are some of the situations where the analytical approaches to Bayesian and likelihood inference are difficult to impossible to implement. The question of estimability of the parameters becomes much more relevant but difficult to diagnose. The method of data cloning is particularly useful for diagnosing estimability of the parameters. You will notice that, although the models are much more complex, the coding component does not increase in complexity. We will also discuss prediction of missing data.

Detection error in occupancy studies (latent variables)

Let us revisit the occupancy model again. In practice, the assumption that you observe the occupancy status correctly is somewhat suspect. For example, if we are looking for a bird species, if the bird never sings or gives some sort of a cue, it is extremely difficult to know if they are there. Hence, even if the species is present, we may make an error and note that it is not present. This is called 'detection error'. How can we model this?

Let W_i denote the true status of the i-th cell. So in our previous notation, now $P(W_i = 1) = \phi$. The observed value, generally denoted by Y_i could be 1 or 0 depending on the true status. If we assume that the species are never misidentified, then we can write $P(Y_i = 1|W_i = 1) = p$ and $P(Y_i = 0|W_i = 1) = 1 - p$. Moreover, $P(Y_i = 0|W_i = 0) = 1$. In principle, we can also have misidentification. For example, a coyote could be mistaken for a wolf. But we will not discuss it here. It is a fairly simple extension of this model. Probability of detection is p and probability of occupancy is ϕ. How can we infer about these given the data?

Notice that we only observe Y_i's and not the W_i's. *The unobserved variable W_i is called a latent variable.*

To write down the likelihood function, we need to compute the distribution of the observed data Y_i. We can see that $P(Y_i = 1) = p\phi$ and $P(Y_i = 0) = (1 - p)\phi$. We can write down the likelihood based on this. However, we are going to write this as a hierarchical model.

Let us modify the previous code to see how this inference proceeds.

```
library(dclone)

phi.true = 0.3 # occupancy
p.true = 0.7   # detectability
n = 30         # sample size
W = rbinom(n, 1 ,phi.true)   # true status
Y = rbinom(n, 1, W * p.true) # detections
```

Bayesian inference using JAGS and dclone Step 1: WE need to define the model function. This is the critical component.

```
Occ.model = function(){
  # Likelihood: Latent variables are random variables.
  for (i in 1:n){
    W[i] ~ dbin(phi_occ, 1)
    Y[i] ~ dbin(W[i] * p_det, 1)
  }
  # Priors
  phi_occ ~ dbeta(1, 1)
  p_det ~ dbeta(1, 1)
}
```

Now we need to provide the data to the model and generate random numbers from the posterior. We will discuss different options later.

```
dat = list(Y=Y, n=n)
```

Following command will not work. But try it by removing the comments hash to see what happens.

```
Occ.Bayes = jags.fit(data=dat, params=c("phi_occ","p_det"), model=Occ.model)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 32
##    Total graph size: 94
##
## Initializing model
## Deleting model
```

```
## Error in rjags::jags.model(model, data, n.chains = n.chains, n.adapt = n.adapt, : Error in node Y[1]
```

```
## Node inconsistent with parents
```

This did not quite work. When there are latent variables, many times, we have to start the process at the appropriate initial values.

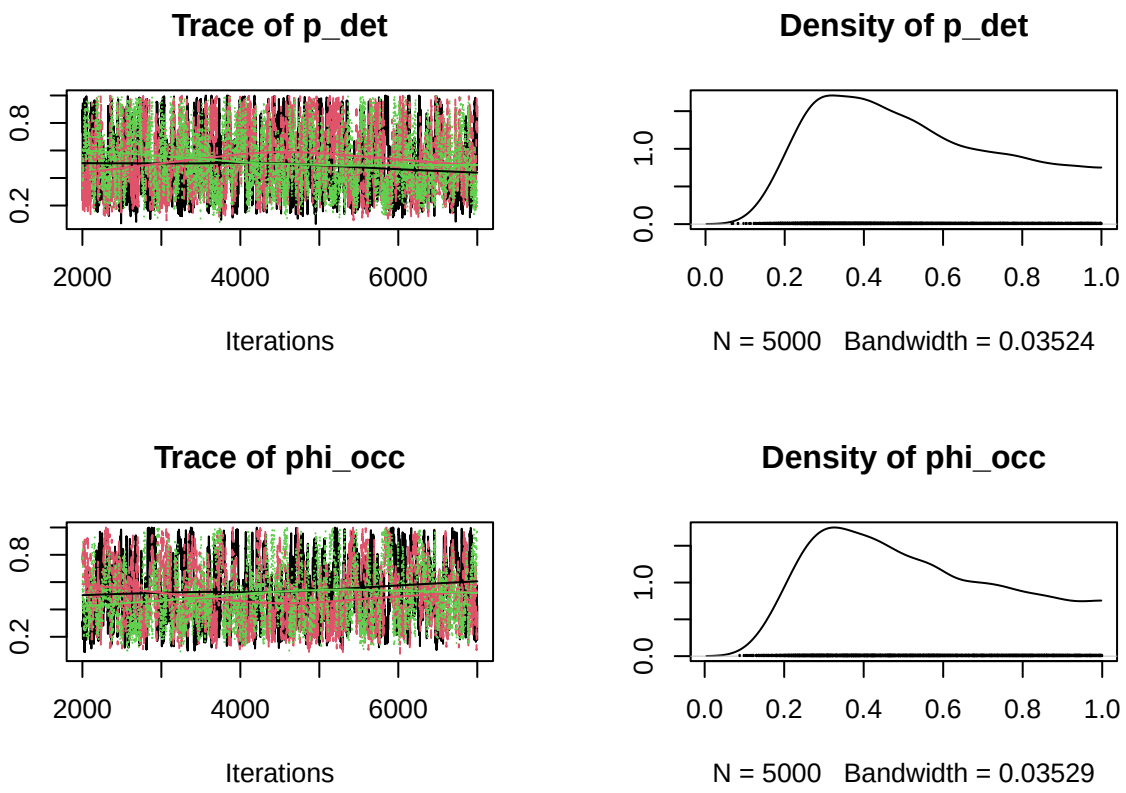
```
ini = list(W=rep(1, n))
Occ.Bayes = jags.fit(data=dat, params=c("phi_occ","p_det"), model = Occ.model,
    inits=ini)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 32
##    Total graph size: 94
##
## Initializing model
```

```
summary(Occ.Bayes)
```

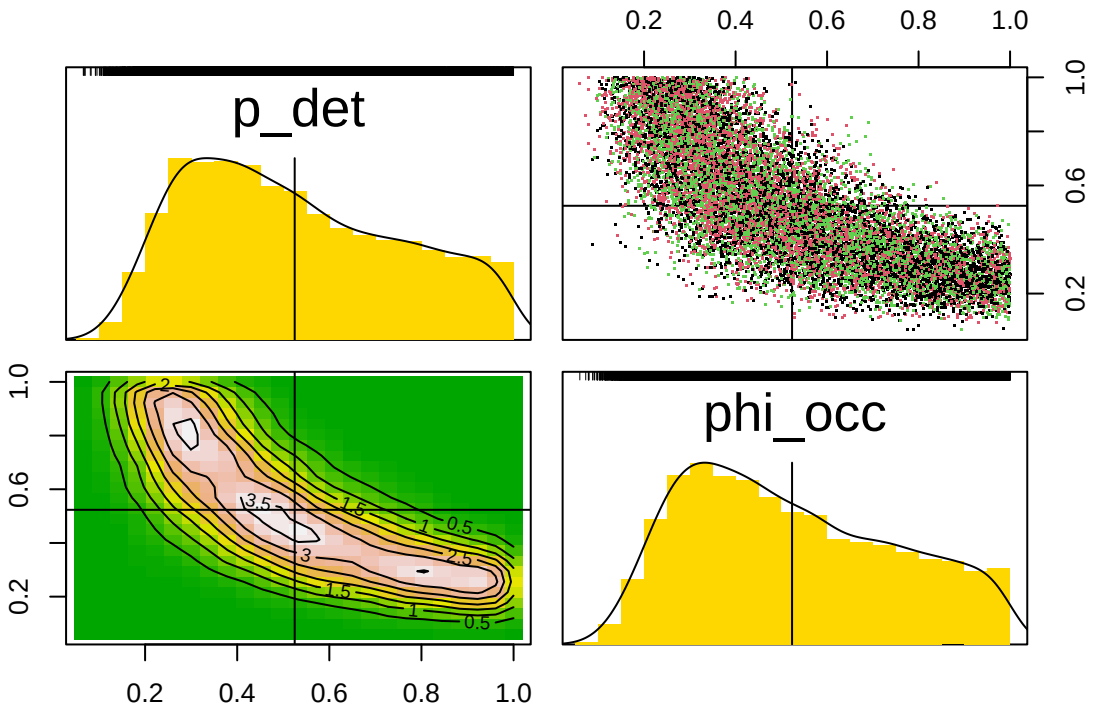
```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean     SD Naive SE Time-series SE
## p_det   0.5248 0.2275 0.001857        0.00956
## phi_occ 0.5235 0.2278 0.001860        0.01093
##
## 2. Quantiles for each variable:
##
##           2.5%    25%    50%    75%  97.5%
## p_det   0.1799 0.3363 0.4899 0.7023 0.9653
## phi_occ 0.1772 0.3341 0.4884 0.7019 0.9674
```

```
plot(Occ.Bayes)
```



This seems to work quite well! But our answers are quite weird (We know the truth!). Let us plot the two dimensional (joint) distribution of the parameters.

```
pairs(Occ.Bayes)
```



This suggests that the posterior distribution is banana shaped. But just looking at these plots, we cannot say for sure if our answers are correct or not.

Should we, then, accept the answers? Not so fast. Let us look at the model again. It is clear that we can estimate the product $p\phi$ given the data. But decomposing this product in p and ϕ is impossible. This is called 'non-estimability'.

In this case, this also is non-identifiability. There are several combinations of p and ϕ that lead to the same $p\phi$ and hence the same distribution of the observed data. Such situations are not uncommon when dealing with the hiearchical models in general, and measurement error models in particular.

We should not make any inferences about the probability of occupancy based on these data. You can change the priors and see what happens to the posteriors. You might find it interesting and educational.

Non-estimability: If there are two or more values in the parameter space that lead to identical likelihood value, such values are called 'non-estimable'.

Note: You may recall from the linear regression that if the covariates are perfectly correlated with each other, the regression coefficients are non-estimable. If covariate X_1 is perfectly correlated with X_2, these covariates separately give no additional information.

Bayesian result and its data cloning version If the posterior distribution converges to a non-degenerate distribution as the sample size increases, it implies that set of parameters is non-estimable.

If the posterior distribution converges to a non-degenerate distribution as the number of clones increases, it implies that set of parameters is non-estimable.

An immediate consequence of this result is that the variance of the posterior distribution does not converge to 0 (instead it converges to some positive number).

Let us modify our data cloning code to see what happens.

```
Occ.model.dc = function(){
  # Likelihood
  for(k in 1:ncl){
    for (i in 1:n){
        W[i,k] ~ dbin(phi_occ, 1)
        Y[i,k] ~ dbin(W[i,k] * p_det, 1)
    }
  }
  # Prior
  phi_occ ~ dbeta(1, 1)
  p_det ~ dbeta(1, 1)
}
```

We need to turn the original data into an array. And we need to add another index `ncl` for the cloned dimension. It gets multiplied by the number of clones.

```
Y = array(Y, dim=c(n, 1))
Y = dcdim(Y)
dat = list(Y=Y, n=n, ncl=1)
```

As previously, we need to initiate the `W`'s.

```
ini = list(W=array(rep(1, n), dim=c(n, 1)))
```

We need to clone these initial values as well. You should always check if this is doing the right job.

```
initfn = function(model, n.clones){
  W=array(rep(1, n), dim=c(n, 1))
  list(W=dclone(dcdim(W), n.clones))
}
initfn(n.clones=2)
```

```
## $W
##       clone.1 clone.2
##  [1,]       1       1
##  [2,]       1       1
##  [3,]       1       1
##  [4,]       1       1
##  [5,]       1       1
##  [6,]       1       1
##  [7,]       1       1
##  [8,]       1       1
##  [9,]       1       1
## [10,]       1       1
## [11,]       1       1
## [12,]       1       1
## [13,]       1       1
## [14,]       1       1
## [15,]       1       1
## [16,]       1       1
## [17,]       1       1
## [18,]       1       1
## [19,]       1       1
## [20,]       1       1
## [21,]       1       1
## [22,]       1       1
## [23,]       1       1
## [24,]       1       1
## [25,]       1       1
## [26,]       1       1
## [27,]       1       1
## [28,]       1       1
## [29,]       1       1
## [30,]       1       1
## attr(,"n.clones")
## [1] 2
## attr(,"n.clones")attr(,"method")
## [1] "dim"
## attr(,"n.clones")attr(,"method")attr(,"drop")
## [1] TRUE
```

Let's run data cloning.

```
Occ.DC = dc.fit(data=dat, params=c("phi_occ","p_det"), model=Occ.model.dc,
    n.clones=c(1, 2, 5), unchanged="n", multiply="ncl",
    inits=ini, initsfun=initfn)
```

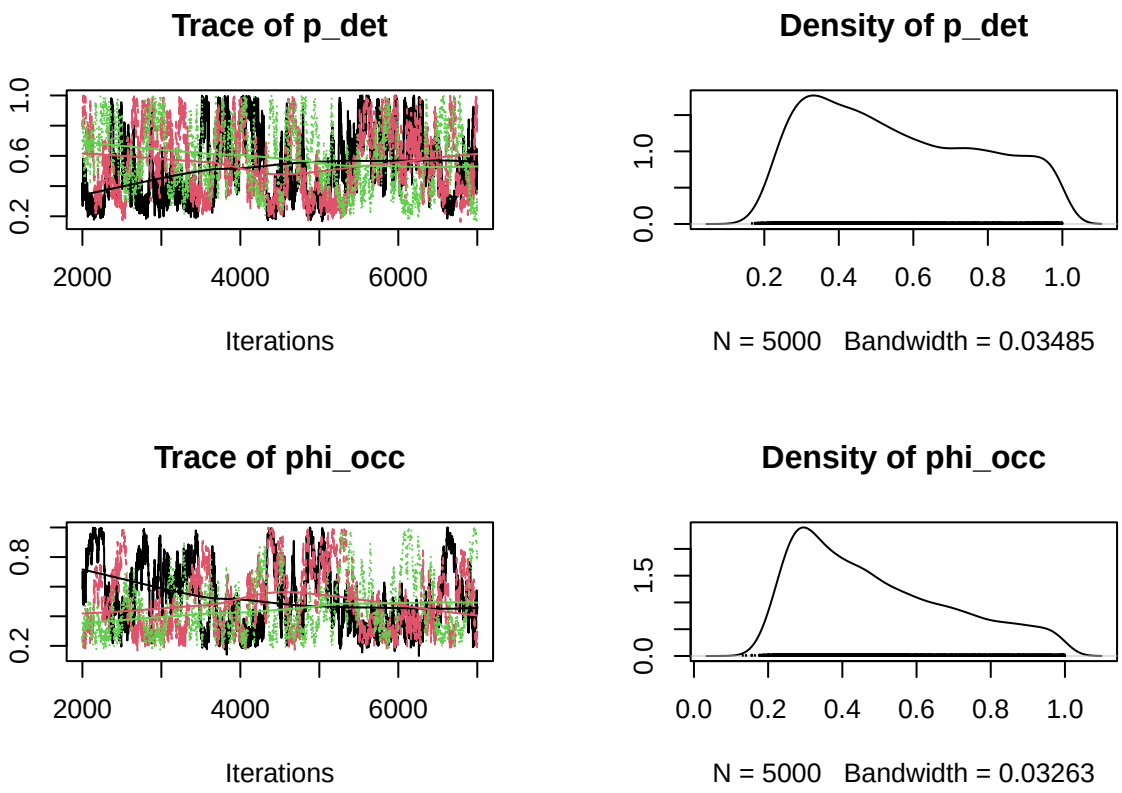
```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 32
##    Total graph size: 95
```

```
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 62
##    Total graph size: 185
##
## Initializing model
##
##
## Fitting model with 5 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 150
##    Unobserved stochastic nodes: 152
##    Total graph size: 455
##
## Initializing model
```

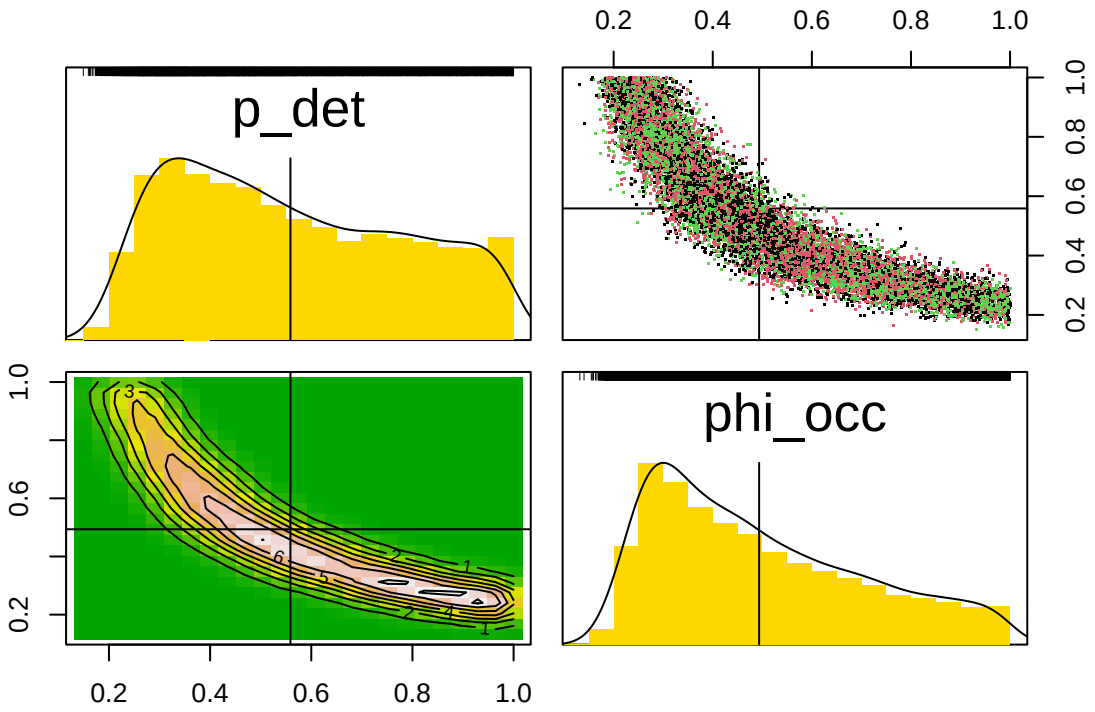
```
summary(Occ.DC)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean     SD DC SD Naive SE Time-series SE R hat
## p_det   0.5588 0.2250 0.503 0.001837        0.01786  1.01
## phi_occ 0.4936 0.2107 0.471 0.001720        0.02089  1.01
##
## 2. Quantiles for each variable:
##
##           2.5%    25%    50%    75%  97.5%
## p_det   0.2282 0.3644 0.5248 0.7460 0.9774
## phi_occ 0.2162 0.3164 0.4468 0.6402 0.9533
```

```
plot(Occ.DC)
```




```
pairs(Occ.DC)
```



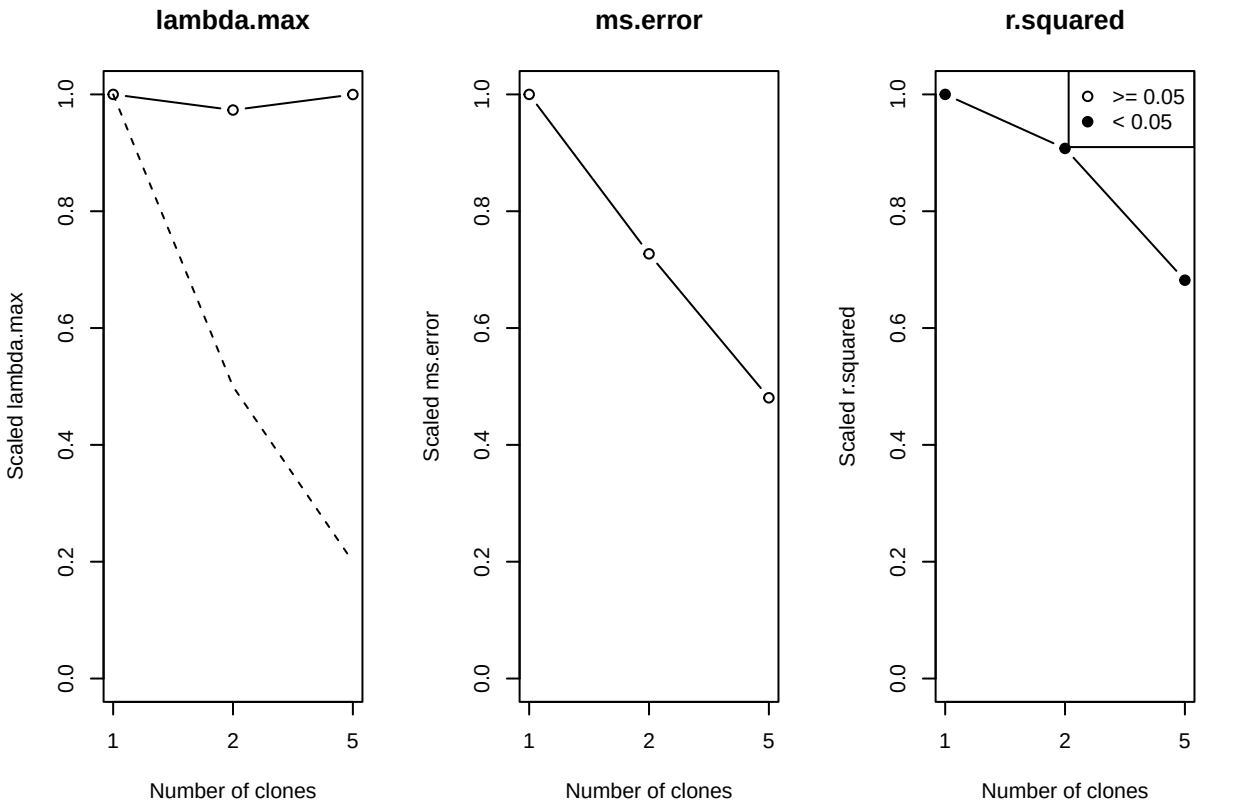
There are a couple of diagnostic tools available in 'dclone' for the non-estimability issue.

```
dcdiag(Occ.DC)
```

```
##   n.clones lambda.max  ms.error  r.squared    r.hat
## 1        1 0.08994617 0.2391589 0.02000742 1.011402
```

```
## 2         2 0.08753866 0.1738830 0.01815677 1.006861
## 3         5 0.08992766 0.1149417 0.01364303 1.011386
```

```
dcdiag(Occ.DC) |> plot()
```



1. Check if the `lambda.max` is converging to zero. The rate at which this converges to zero should be approximately $1/K$.
2. If the variance is converging to 0 but the rate is different than $1/K$, it implies the asymptotics is of a different order. If you find such an example in your work, please let me know.
3. Check the pairs plot to see if the likelihood function is converging to a manifold instead of a single point.

Availability of the estimability diagnostics is one of the most important features of data cloning. It will warn you if your scientific inferences could be misleading.

If the parameters are non-estimable, the only recourse one has is to change the model (add assumptions or collect different kind of data). There is always a possibility that, although the full parameter space may not be estimable, a function of the parameter might be estimable. If such a function is also of scientific interest, we can safely conduct scientific inferences based on estimates of such a function of the parameters.

Replicate surveys Suppose we visit the same cell several times. Assume that the visits are independent of each other and the true occupancy status remains the same throughout these surveys, then we can estimate the parameters. The model can be written as a hierarchical model:

- Hierarchy 1: $W_i \sim Bernoulli(\phi)$
- Hierarchy 2: $Y_{ij}|W_i = w_i \sim Bernoulli(pw_i)$

Notice that hierarchy 2 depends on hierarchy 1 result.

We can easily modify the earlier code to allow for multiple surveys. We will do such a modification with two surveys for each cell.

```
phi.true = 0.3
p.true = 0.7
n = 30
v = 2 # number of visits
W = rbinom(n, 1, phi.true)
Y = NULL
for (j in 1:v)
    Y <- cbind(Y, rbinom(n, 1, W * p.true))
```

Bayesian inference using JAGS and dclone Step 1: we need to define the model function.

```
Occ.model = function(){
  # Likelihood: Latent variables are random variables.
  for (i in 1:n){
    W[i] ~ dbin(phi_occ, 1)
    for (j in 1:v){
        Y[i,j] ~ dbin(W[i] * p_det, 1)}
  }
  # Priors
  phi_occ ~ dbeta(1, 1)
  p_det ~ dbeta(1, 1)
}
```

Now we need to provide the data to the model and generate random numbers from the posterior. We will discuss different options later.

```
dat = list(Y=Y, n=n, v=v)
ini = list(W=rep(1, n))

Occ.Bayes = jags.fit(data=dat, params=c("phi_occ","p_det"), model=Occ.model,
    inits=ini)
```

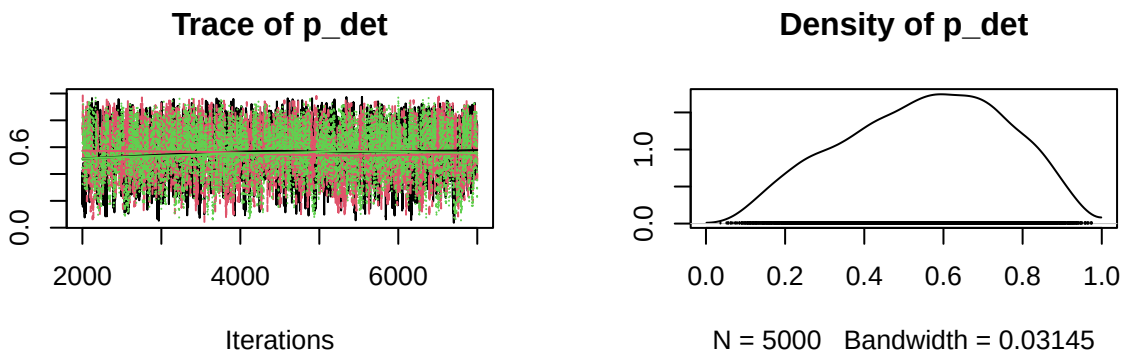
```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 32
##    Total graph size: 125
##
## Initializing model
```

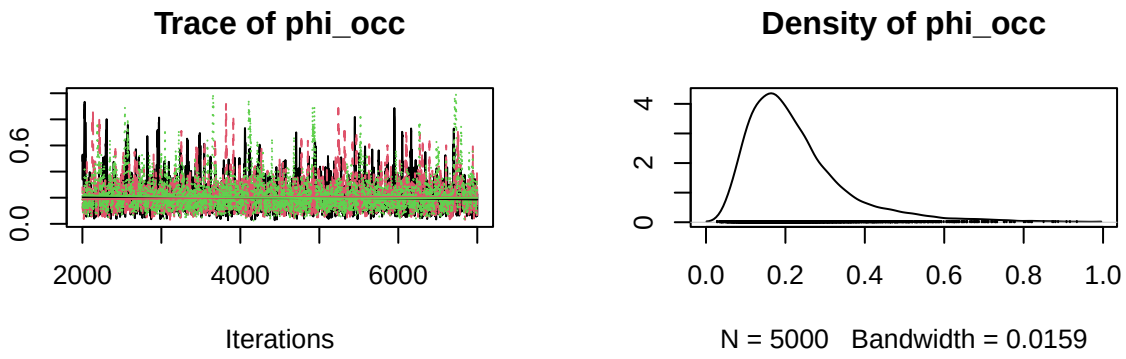
```
summary(Occ.Bayes)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## p_det   0.5507 0.2030 0.001657       0.005196
## phi_occ 0.2280 0.1328 0.001084       0.004389
```

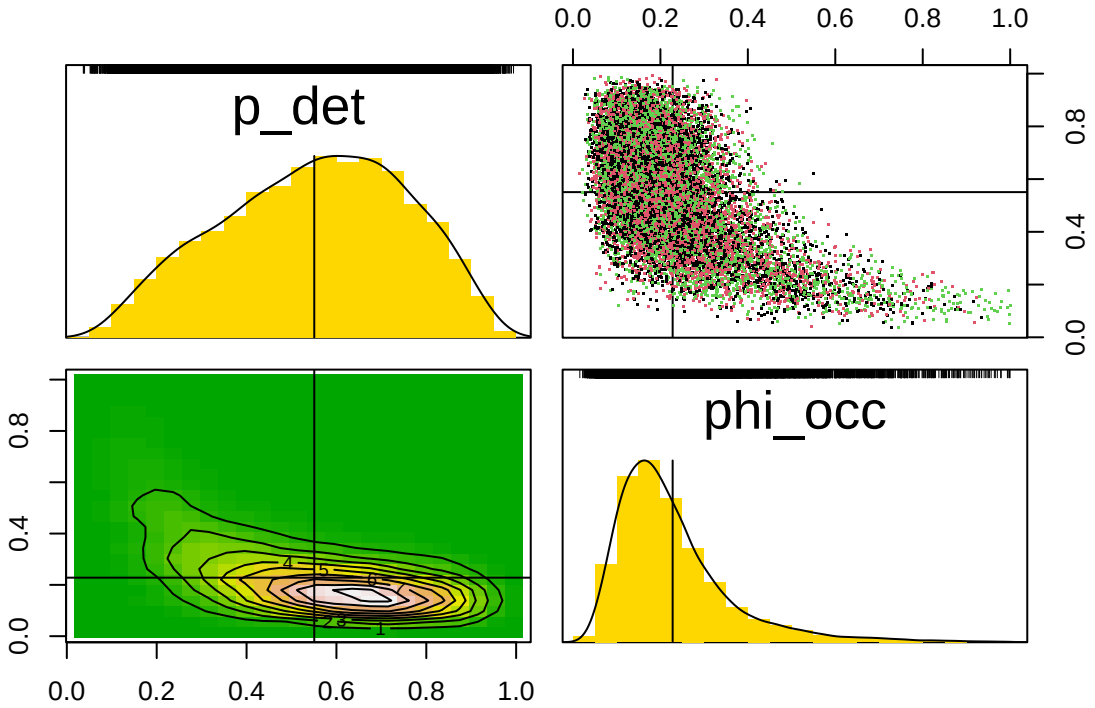
```
##
## 2. Quantiles for each variable:
##
##             2.5%    25%    50%    75%  97.5%
## p_det   0.15730 0.4019 0.5640 0.7086 0.8978
## phi_occ 0.06727 0.1397 0.1972 0.2772 0.5932
```

```
plot(Occ.Bayes)
```






```
pairs(Occ.Bayes)
```



These are nice unimodal posterior distributions.

We can modify the data cloning code to check if the parameters are, in fact, estimable.

```
Occ.model.dc = function(){
  # Likelihood
  for(k in 1:ncl){
    for (i in 1:n){
      W[i,k] ~ dbin(phi_occ, 1)
      for (j in 1:v){
        Y[i,j,k] ~ dbin(W[i,k] * p_det, 1)
      }
    }
  }
  # Prior
  phi_occ ~ dbeta(1, 1)
  p_det ~ dbeta(1, 1)
}
```

We need to turn the original data into an array. And we need to add another index `ncl` for the cloned dimension. It gets multiplied by the number of clones.

```
Y = array(Y, dim=c(n,v,1))
Y = dcdim(Y)
dat = list(Y=Y, n=n, v=v, ncl=1)

# As previously, we need to initiate the W's.
ini = list(W=array(rep(1, n), dim=c(n, 1)))
# We need to clone these initial values as well. You should always check if this is doing the right job
initfn =function(model, n.clones){
  W=array(rep(1,n), dim=c(n,1))
  list(W=dclone(dcdim(W), n.clones))
}
```

```r
Occ.DC = dc.fit(data=dat, params=c("phi_occ","p_det"), model=Occ.model.dc,
    n.clones=c(1, 2, 5),
    unchanged=c("n","v"), multiply="ncl",
    inits=ini, initsfun=initfn)
```

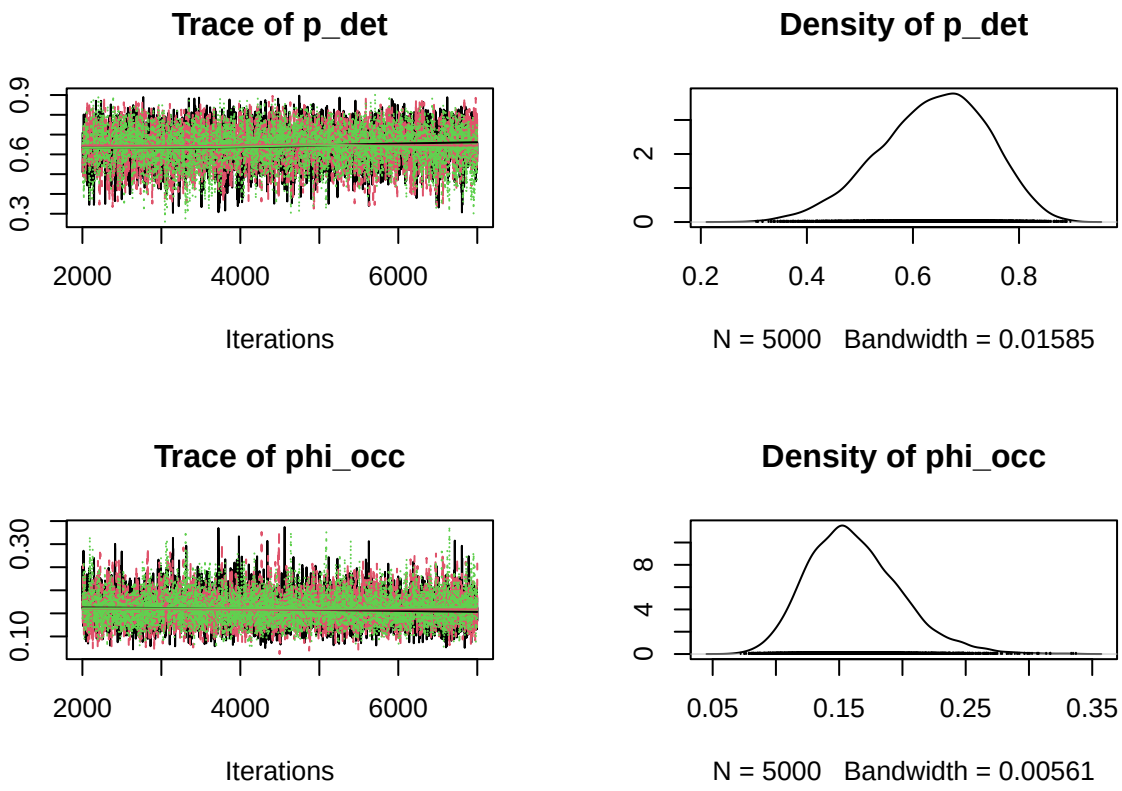
```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 32
##    Total graph size: 126
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 120
##    Unobserved stochastic nodes: 62
##    Total graph size: 246
##
## Initializing model
##
##
## Fitting model with 5 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 300
##    Unobserved stochastic nodes: 152
##    Total graph size: 606
##
## Initializing model
```

```r
summary(Occ.DC)
```

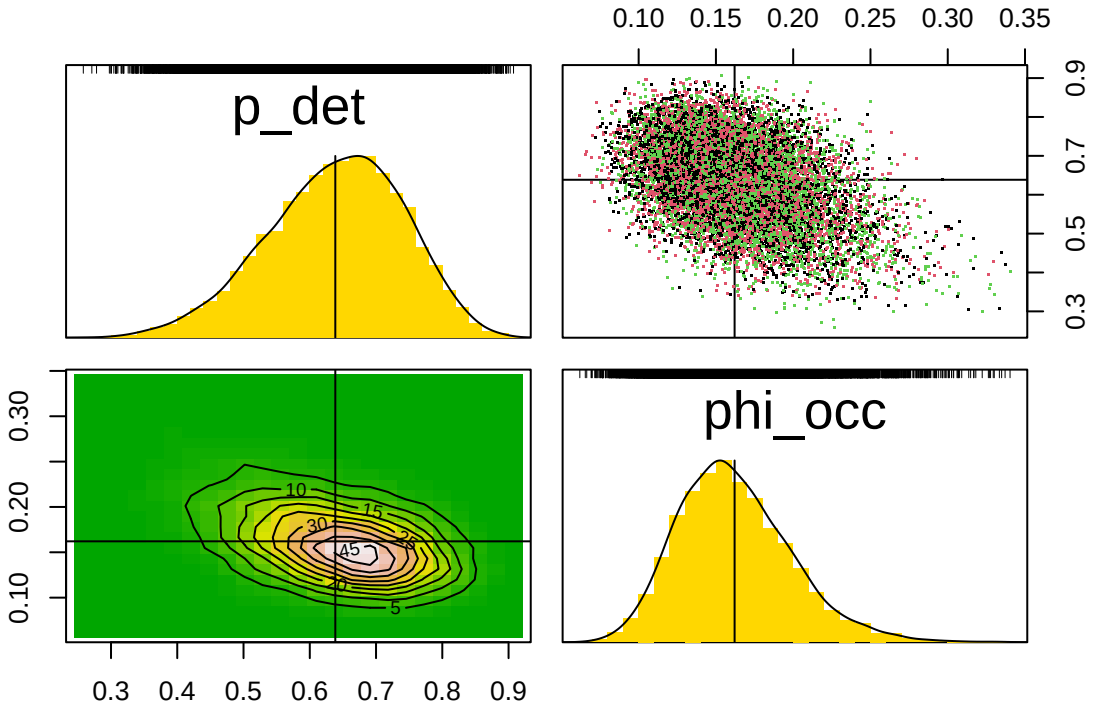
```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
```

```
##    plus standard error of the mean:
##
##            Mean      SD   DC SD  Naive SE Time-series SE R hat
## p_det   0.6385 0.10229 0.22873 0.0008352      0.0020676 1.001
## phi_occ 0.1621 0.03659 0.08182 0.0002988      0.0006217 1.002
##
## 2. Quantiles for each variable:
##
##            2.5%    25%    50%    75%  97.5%
## p_det   0.4231 0.5711 0.6460 0.7129 0.8180
## phi_occ 0.1005 0.1359 0.1584 0.1845 0.2446
```

```
plot(Occ.DC)
```



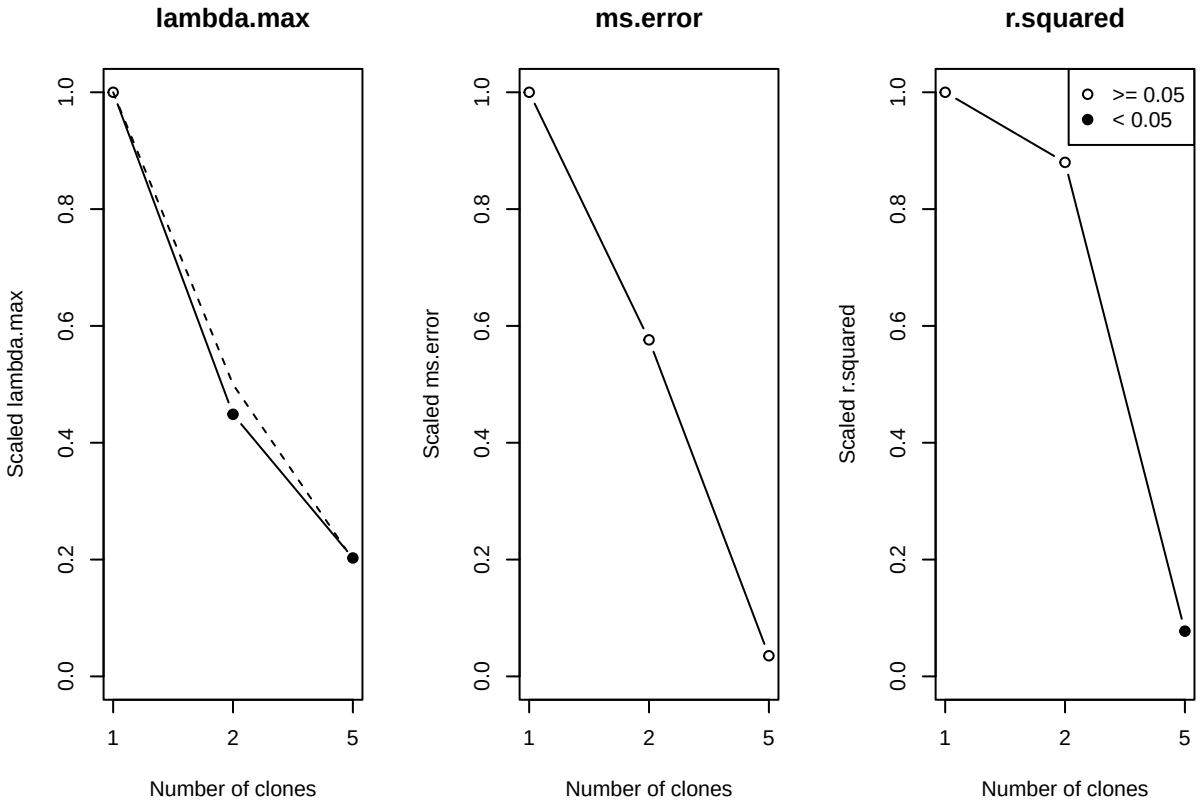

```
pairs(Occ.DC)
```




```
dcdiag(Occ.DC)
```

```
##   n.clones lambda.max   ms.error  r.squared    r.hat
## 1        1 0.05299637 1.49976776 0.13149513 1.002550
## 2        2 0.02377704 0.86408723 0.11571485 1.000889
## 3        5 0.01074515 0.05297445 0.01018441 1.001901
```

```
plot(dcdiag(Occ.DC))
```



Random effects in regression: Why and when?

We have shown how hierarchical models can be used to deal with measurement error. Now we will look at a few examples where we use them to combine data across several studies.

We will start with a simple (but extremely important) example that started the entire field of mixture as well as hierarchical models (Neyman and Scott, 1949).

Researchers in animal husbandary wanted to know how to improve the stock of animals such as milk cows and bulls. This played an important role in the 'white revolution' that lead to improving the nutrition in many countries. Following is a somewhat made up and highly simplified situation.

Suppose we have n cows. We want to know which cows have good genetic potential that can be passed on to the next generation. Each cow might have only a few calves. We measure the amount of milk by each calf.

Let Y_ij be the amount of milk produced by the j-th calf of the i-th cow. We can consider a linear model that uses the 'cow effect' (genetic) and 'environmental effect' to explain the amount of milk. This is same as one way ANOVA model.

$Y_{ij} = \mu + \alpha_i + \epsilon_{ij}$

Under the usual Gaussian error structure, we know that

$Y_{ij} \sim N(\mu_i, \sigma^2)$ where $i = 1, 2, ..., n$ and $j = 1, 2$ ($\mu_i = \mu + \alpha_i$).

There are $2n$ observations and $n + 1$ parameters. The number of parameters increases at the same rate as the number of observations. Note that the ratio of parameters to observations converges to 0.5. Roughly speaking, for the MLE to work, this ratio needs to go to 0. Generally the number of parameters is fixed and hence this condition is satisfied.

We simply do not have much information about each μ_i as there are only two observations corresponding to it. It also turns out that the ML estimator of σ^2 converges to $0.5 * \sigma^2$. Hence it is not consistent even though

the number of observations corresponding to it do converge to infinity. This was a major blow to the theory of maximum likelihood. Although it turns out Fisher had implicitly answered it a decade before this paper.

How can we reduce the number of parameters?

1. If there are covariates such as weight of the mother, mother's milk production are available, we can model $\mu_i = X_i\beta$.
2. Suppose covariates are not available or difficult to assess what to use as a covariate. If we assume that cows are kind of similar to each other, then we can assume that they come from a population of cows such that $\mu_i \sim N(\mu, \tau^2)$. It turns out, under such an assumption, we can estimate the parameters (μ, σ^2, τ^2) consistently.

This model is a hierarchical model.

- Hierarchy 1: $Y_{ij}|\mu_i \sim N(\mu_i, \sigma^2)$
- Hierarchy 2: $\mu_i \sim N(\mu, \tau^2)$

For a Bayesian approach, we put priors on the three parameters. This forms the third hierarchy.

This simple model can be used in many different situations.

1. Measurement error in covariates in regression
2. Random intercept regression model to account for missing covariates
3. Kalman filter models for time series with measurement error are of the same kind with a bit more complexity as we will see the third part.

Let us see what makes it a difficult model to analyze using the likelihood approach.

In order to write down the likelihood function, we need to compute the marginal distribution of the observations. Remember μ_i are not observed. Hence we have to integrate over them.

$$f(y_{ij}; \mu, \sigma^2, \tau^2) = \int f(y_{ij}|\mu_i) g(\mu_i) d\mu_i$$

Again, this is not a precise statement but it gives you the idea. This integral is one dimensional and hence can be computed analytically and also numerically. However, in many cases the dimension of the integral is quite large and hence neither of these solutions are available. In some cases, one can obtain Laplace approximation to this integral (INLA and related methods rely on this approximation). The most general approach is based on the MCMC algorithm.

```
n = 30
mu_true = 2
tau_true = 0.5
sigma_true = 1

mu = rnorm(n, mu_true, tau_true)
Y = cbind(
    rnorm(n, mu, sigma_true),
    rnorm(n, mu, sigma_true))
```

We will write the Bayesian model first. Remember the normal distribution is defined in terms of the precision (inverse of the variance).

```
LM_Bayes = function(){
  # Likelihood
  for (i in 1:n){
    mu[i] ~ dnorm(mu.t, prec_tau)
    for (j in 1:2){
      Y[i,j] ~ dnorm(mu[i], prec_sigma)
    }
```

```
  }
  # Priors
  mu.t ~ dnorm(0, 0.01)
  prec_tau ~ dgamma(0.1, 0.1)
  prec_sigma ~ dgamma(0.1, 0.1)
  # Parameters on the natural scale
  tau <- sqrt(1/prec_tau)
  sigma <- sqrt(1/prec_sigma)
}
```

Data in array form for data cloning purpose

```
Y = as.array(Y, dim=c(n,2,1))
Y = dcdim(Y)

dat = list(Y=Y, n=n)
LM_Bayes_fit = jags.fit(data=dat, params=c("mu.t","tau","sigma"), model=LM_Bayes)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 33
##    Total graph size: 102
##
## Initializing model
```

```
summary(LM_Bayes_fit)
```

```
##
## Iterations = 1001:6000
## Thinning interval = 1
## 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
## mu.t  2.0708 0.1662 0.0013567       0.003295
## sigma 1.0374 0.1186 0.0009686       0.001784
## tau   0.5112 0.1777 0.0014508       0.005292
##
## 2. Quantiles for each variable:
##
##         2.5%    25%    50%    75%  97.5%
## mu.t  1.7464 1.9586 2.0706 2.1789 2.4020
## sigma 0.8271 0.9548 1.0287 1.1135 1.2879
## tau   0.2232 0.3790 0.4953 0.6248 0.8959
```

We will modify it to do data cloning. This is useful to assure us that the parameters are estimable. It is also important for making it invariant to the choice of the priors and parameterization.

```
LM_DC = function(){
  # Likelihood
```

```
  for (k in 1:ncl){
    for (i in 1:n){
      mu[i,k] ~ dnorm(mu.t, prec_tau)
      for (j in 1:2){
        Y[i,j,k] ~ dnorm(mu[i,k], prec_sigma)
      }
    }
  }
  # Priors
  mu.t ~ dnorm(0, 0.01)
  prec_tau ~ dgamma(0.1, 0.1)
  prec_sigma ~ dgamma(0.1, 0.1)
  # Parameters on the natural scale
  tau <- sqrt(1/prec_tau)
  sigma <- sqrt(1/prec_sigma)
}
```

Data in array form for data cloning purpose.

```
Y = array(Y, dim=c(n,2,1))
Y = dcdim(Y)

dat = list(Y=Y, n=n, ncl=1)
LM_DC_fit = dc.fit(data=dat, params=c("mu.t","tau","sigma"), model=LM_DC,
    n.clones=c(1, 2, 5), unchanged="n", multiply="ncl")
```

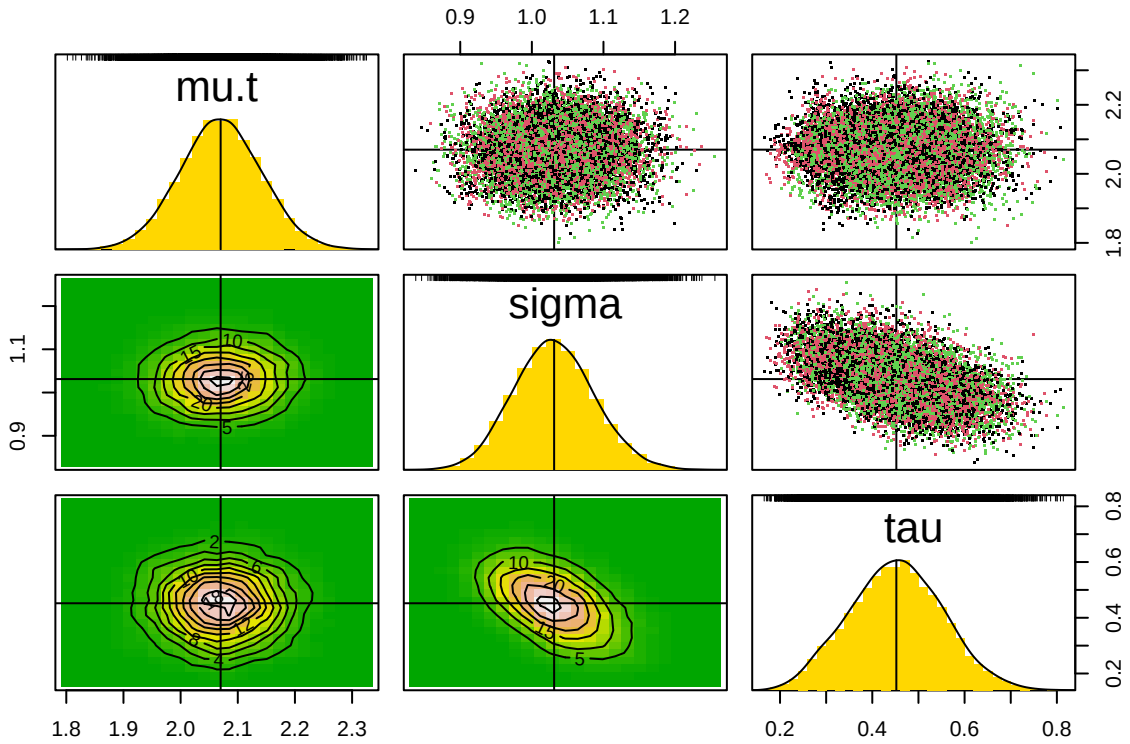
```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 33
##    Total graph size: 103
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 120
##    Unobserved stochastic nodes: 63
##    Total graph size: 193
##
## Initializing model
##
##
## Fitting model with 5 clones
```

```
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 300
##    Unobserved stochastic nodes: 153
##    Total graph size: 463
##
## Initializing model
```

```
summary(LM_DC_fit)
```

```
##
## Iterations = 1001:6000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##         Mean      SD  DC SD  Naive SE Time-series SE R hat
## mu.t  2.0700 0.07089 0.1585 0.0005788       0.001492 1.001
## sigma 1.0311 0.05563 0.1244 0.0004542       0.001333 1.000
## tau   0.4521 0.10039 0.2245 0.0008197       0.004533 1.004
##
## 2. Quantiles for each variable:
##
##         2.5%    25%    50%    75%  97.5%
## mu.t  1.9315 2.0225 2.0696 2.1176 2.2100
## sigma 0.9267 0.9929 1.0297 1.0679 1.1433
## tau   0.2608 0.3824 0.4523 0.5211 0.6522
```

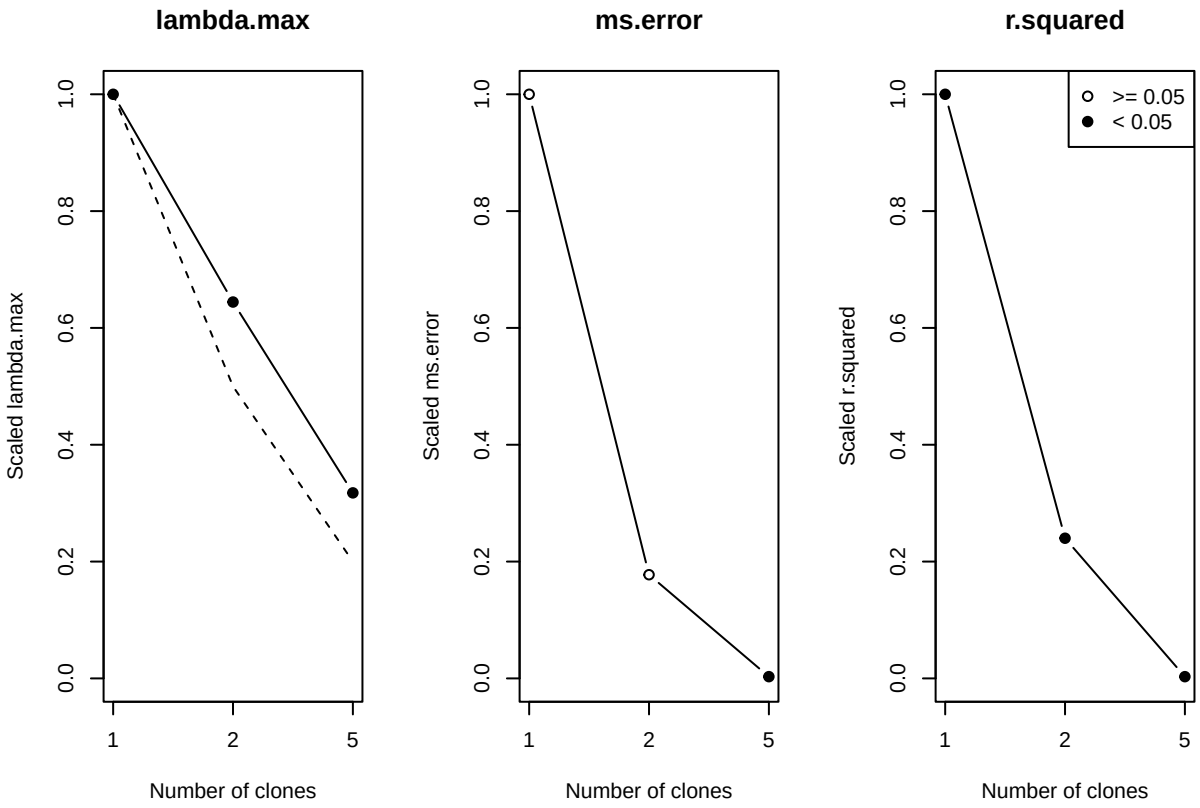
```
pairs(LM_DC_fit)
```




```
dcdiag(LM_DC_fit)
```

```
##   n.clones lambda.max    ms.error    r.squared    r.hat
## 1        1 0.03478998 0.345743278 0.0369969887 1.002886
## 2        2 0.02241605 0.061369558 0.0088769002 1.007382
## 3        5 0.01104988 0.001030745 0.0001092459 1.003334
```

```
plot(dcdiag(LM_DC_fit))
```



The best thing about the MCMC approach is that we can modify the prototype program to do Generalized linear mixed models.

Let us see how we can change the program to do Poisson regression with random intercepts. This is useful for accounting for missing covariates in the usual Poisson regression. This can also be used to account for site effect in abundance surveys.

Mathematically the model is:

- Hierarchy 1: $log(\lambda_i) \sim N(log(\lambda), \tau^2)$
- Hierarchy 2: $Y_i|\lambda_i \sim Poisson(\lambda_i)$

```
n = 30
mu_true = 2
tau_true = 0.5
sigma_true = 1

mu = rnorm(n, mu_true, tau_true)
Y = rpois(n, exp(mu))
```

We will write the Bayesian model first. Remember the normal distribution is defined in terms of the precision (inverse of the variance).

```
GLMM_Bayes = function(){
  # Likelihood
  for (i in 1:n){
    mu[i] ~ dnorm(mu.t, prec_tau)
    Y[i] ~ dpois(exp(mu[i]))
  }
  # Priors
  mu.t ~ dnorm(0, 0.01)
```

```
  prec_tau ~ dgamma(0.1, 0.1)
  # Parameters on the natural scale
  tau <- sqrt(1/prec_tau)
  lambda <- exp(mu.t)
}

dat = list(Y=Y, n=n)
GLMM_Bayes_fit = jags.fit(data=dat, params=c("lambda","tau"), model=GLMM_Bayes)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 32
##    Total graph size: 100
##
## Initializing model
```

```
summary(GLMM_Bayes_fit)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## lambda 7.8156 0.87360 0.0071329       0.011823
## tau    0.4712 0.09532 0.0007783       0.001669
##
## 2. Quantiles for each variable:
##
##          2.5%    25%    50%    75%  97.5%
## lambda 6.1877 7.2096 7.7945 8.3945 9.5838
## tau    0.3093 0.4038 0.4627 0.5289 0.6823
```

As usual, we can turn the crank for data cloning to check the estimability of the parameters.

```
GLMM_DC = function(){
  # Likelihood
  for (k in 1:ncl){
    for (i in 1:n){
      mu[i,k] ~ dnorm(mu.t,prec_tau)
      Y[i,k] ~ dpois(exp(mu[i,k]))
    }
  }
  # Priors
  mu.t ~ dnorm(0, 0.01)
  prec_tau ~ dgamma(0.1, 0.1)
  # Parameters on the natural scale
  tau <- sqrt(1/prec_tau)
```

```
  lambda <- exp(mu.t)
}

# Data in array form for data cloning purpose

Y = array(Y, dim=c(n,1))
Y = dcdim(Y)

dat = list(Y=Y, n=n, ncl=1)
GLMM_DC_fit = dc.fit(data=dat, params=c("lambda","tau"), model=GLMM_DC,
    n.clones=c(1, 2, 5), unchanged="n", multiply="ncl")
```

```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 32
##    Total graph size: 101
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 60
##    Unobserved stochastic nodes: 62
##    Total graph size: 191
##
## Initializing model
##
##
## Fitting model with 5 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 150
##    Unobserved stochastic nodes: 152
##    Total graph size: 461
##
## Initializing model
```

```
summary(GLMM_DC_fit)
```

```
##
```

```
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##          Mean      SD   DC SD  Naive SE Time-series SE R hat
## lambda 7.8123 0.36975 0.82678 0.0030190      0.0052485 1.001
## tau    0.4427 0.04073 0.09108 0.0003326      0.0008063 1.000
##
## 2. Quantiles for each variable:
##
##          2.5%    25%    50%    75% 97.5%
## lambda 7.0854 7.5619 7.8071 8.0586 8.551
## tau    0.3664 0.4146 0.4416 0.4699 0.526
```

```
exp(mu_true)
```

```
## [1] 7.389056
```

```
tau_true
```

```
## [1] 0.5
```

Estimating the effect of a treatment from multi-center clinical trials In clinical trials, we are interested in estimating the effect of the treatment. One of the simplest forms of clinical trial is where we split a group of patients in two groups randomly. One of the group gets the treatment and the other gets a placebo.

We can then estimate the difference in the outcomes. This may be done using a simple t-test if the outcome is a continuous measurement. If the patients are quite different from each other in terms of say age, blood pressure or some such physical characteristics that may affect the outcome, we adjust them by using a regression approach. We include these other covariates and the treatment/control indicator variable in the model. The effect of the treatment after adjusting for other covariates can be studied using such a regression model.

Often in practice, we may not have access to such covariates. Using random intercept model in regression is one way out of such a situation. In this case, we do consider the differences between patients but without ascribing them to any specific, known values of the covariates. The model one may consider is:

$$Y_i = \beta_{0i} + \beta I_{(Treatment)} + \epsilon_i$$

As we have seen in the previous example (Neyman-Scott problem), this model is non-estimable. One way to make it estimable is by using a hierarchical structure:

Hierarchy 2: $\beta_{0i} \sim N(\beta_0, \tau^2)$

Homework: Check the validity of the following statement without doing any mathematics. You can use data cloning to do that.

This leads to estimability for β, the parameter of interest. Although it does not lead to estimation of the variances (τ, σ).

> An approach not described in this course: We can use profile likelihood for the parameter β. This eliminates the 'nuisance parameters' β_0, τ, σ. Computing the profile likelihood and quantification

of uncertainty for inferences based on it for hierarchical models can be tackled using data cloning. This could be another course!

Suppose the outcome is binary, survival for 5 years vs. failure before 5 years. In this case, a convenient model is a binary regression model such as a Logistic regression model.

Hierarchy 1:

$$P(Y_i = 1) = \frac{exp(\beta_{0i} + \beta I_{(Treatment)})}{1 + exp(\beta_{0i} + \beta I_{(Treatment)})}$$

Hierarchy 2: $\beta_{0i} \sim N(\beta_0, \tau^2)$

This is an example of a Generalized Linear Mixed Model. Fortunately, for this model all parameters are estimable as we will see using data cloning. The random intercept here could be accounting for differences in the clinical centers (hospitals), assuming we have only two patients, one in control and one in the treatment group. If we have multiple patients in each group, we may include another random effect to account for differences in the patients within each group. For example, we may consider a model:

Hierarchy 1:

$$P(Y_{ij} = 1) = \frac{exp(\beta_{0i} + \beta I_{(Treatment)} + \beta_{1j})}{1 + exp(\beta_{0i} + \beta I_{(Treatment)} + \beta_{1j})}$$

Hierarchy 2: $\beta_{0i} \sim N(\beta_0, \tau^2)$, $\beta_{1j} \sim N(\beta_1, \tau^2)$

We can include interactions between random effects and so on. We will not go into these complex models in this course.

Let us see how one can modify the prototype program to analyze the random intercept Logistic regression model.

Each center has one control and one treatment patient.

```
n = 300      # Number of clinical centers
mu_true = 0
tau_true = 0.5
delta = 1   # Treatment effect
mu = rnorm(n, mu_true, tau_true)
Y = cbind(
    rbinom(n, 1, plogis(mu)),
    rbinom(n, 1, plogis(mu+delta)))
```

We will write the Bayesian model first. Remember the normal distribution is defined in terms of the precision (inverse of the variance).

```
GLMM_Bayes = function(){
  # Likelihood
  for (i in 1:n){
    mu[i] ~ dnorm(mu.t,prec_tau)
    Y[i,1] ~ dbin(ilogit(mu[i]),1)
    Y[i,2] ~ dbin(ilogit(mu[i]+delta),1)
  }
  # Priors
  mu.t ~ dnorm(0, 0.01)
  prec_tau ~ dgamma(0.1, 0.1)
  delta ~ dnorm(0, 0.1)
  # Parameters on the natural scale
```

```
  tau <- sqrt(1/prec_tau)
}

dat = list(Y=Y, n=n)
GLMM_Bayes_fit = jags.fit(data=dat, params=c("delta","mu.t","tau"), model=GLMM_Bayes)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 600
##    Unobserved stochastic nodes: 303
##    Total graph size: 1810
##
## Initializing model
```

```
summary(GLMM_Bayes_fit)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## delta 0.93188 0.1783 0.0014559       0.007235
## mu.t  0.03716 0.1205 0.0009842       0.007174
## tau   0.49883 0.1876 0.0015317       0.021648
##
## 2. Quantiles for each variable:
##
##          2.5%      25%     50%    75%  97.5%
## delta  0.5863  0.80974 0.93100 1.0510 1.2875
## mu.t  -0.2007 -0.04259 0.03627 0.1166 0.2768
## tau    0.2117  0.35318 0.47522 0.6181 0.9210
```

We will modify this to get the MLE using data cloning.

```
GLMM_DC = function(){
  # Likelihood
  for (k in 1:ncl){
    for (i in 1:n){
      mu[i,k] ~ dnorm(mu.t,prec_tau)
      Y[i,1,k] ~ dbin(ilogit(mu[i,k]),1)
      Y[i,2,k] ~ dbin(ilogit(mu[i,k]+delta),1)
    }
  }
  # Priors
  mu.t ~ dnorm(0, 0.01)
  prec_tau ~ dgamma(0.1, 0.1)
  delta ~ dnorm(0, 0.1)
  # Parameters on the natural scale
```

```
  tau <- sqrt(1/prec_tau)
}

Y = array(Y, dim=c(dim(Y),1))
Y = dcdim(Y)
dat = list(Y=Y, n=n, ncl=1)

GLMM_DC_fit = dc.fit(data=dat, params=c("delta","mu.t","tau"), model=GLMM_DC,
    n.clones=c(1, 2, 5), unchanged="n", multiply="ncl")
```

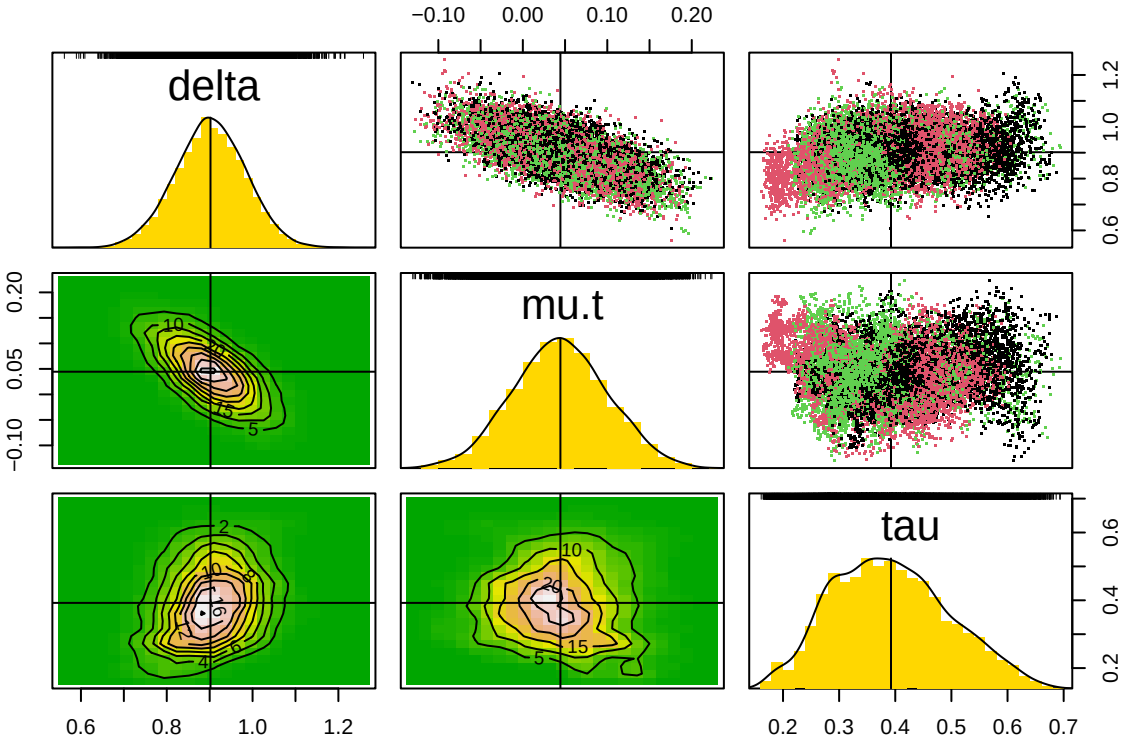
```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 600
##    Unobserved stochastic nodes: 303
##    Total graph size: 1811
##
## Initializing model
##
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 1200
##    Unobserved stochastic nodes: 603
##    Total graph size: 3611
##
## Initializing model
##
##
## Fitting model with 5 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 3000
##    Unobserved stochastic nodes: 1503
##    Total graph size: 9011
##
## Initializing model
```

```
summary(GLMM_DC_fit)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
```

```
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 5
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##          Mean      SD  DC SD  Naive SE Time-series SE R hat
## delta 0.90192 0.08162 0.1825 0.0006664       0.004446 1.005
## mu.t  0.04449 0.05555 0.1242 0.0004536       0.004207 1.005
## tau   0.39240 0.10233 0.2288 0.0008355       0.019597 1.082
##
## 2. Quantiles for each variable:
##
##           2.5%      25%     50%     75%  97.5%
## delta  0.73971 0.847555 0.90183 0.95712 1.0599
## mu.t  -0.06254 0.006781 0.04417 0.08174 0.1537
## tau    0.21052 0.313770 0.38634 0.46188 0.6025
```

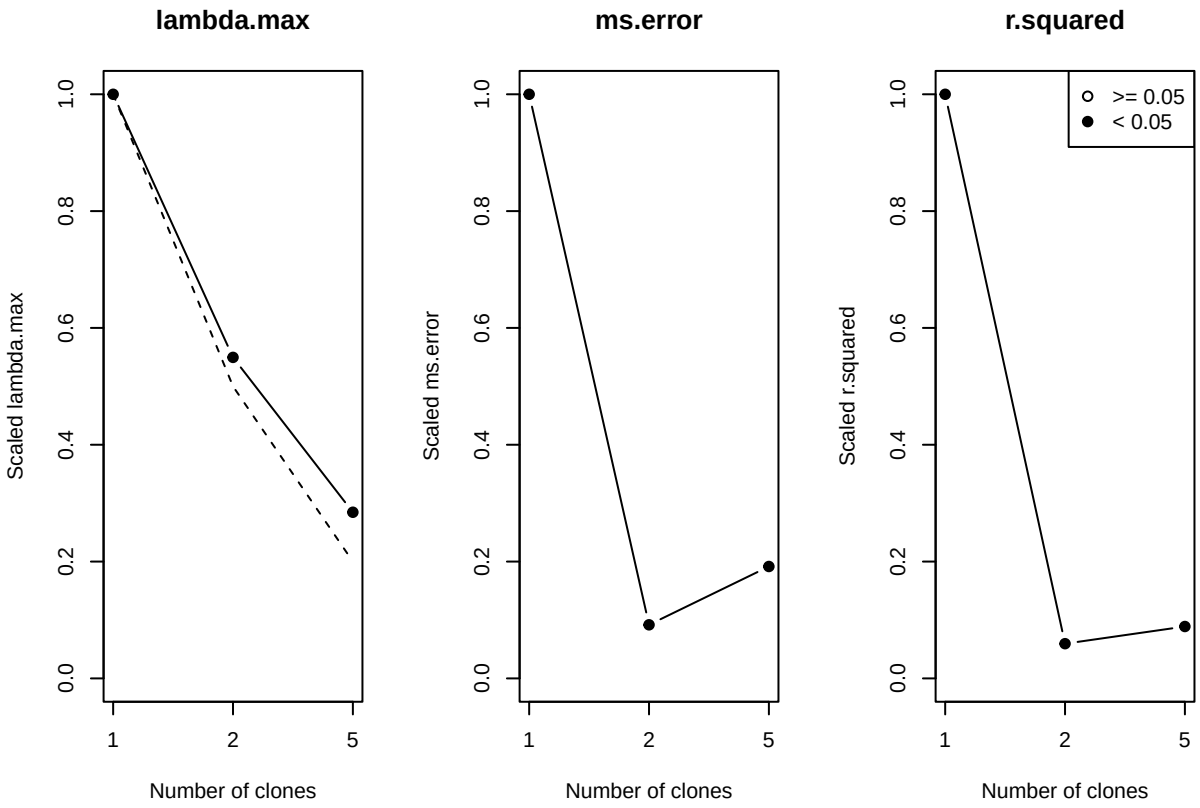
```
pairs(GLMM_DC_fit)
```




```
dcdiag(GLMM_DC_fit)
```

```
##   n.clones lambda.max    ms.error    r.squared    r.hat
## 1        1 0.04109938 0.021983801 0.0034295968 1.078344
## 2        2 0.02258659 0.002016758 0.0002038430 1.110367
## 3        5 0.01168980 0.004211409 0.0003046215 1.066967
```

```
plot(dcdiag(GLMM_DC_fit))
```



Following is a real data set on multi-center clinical trials. Number of patients in the treatment are `"nt"` and those who survived are `"rt"`. Similarly `"nc"` are the number of patients in the control group and `"rc"` are the ones that survived.

```
OR.data = list(
    "rt" =
        c(3, 7, 5, 102, 28, 4, 98, 60, 25, 138, 64, 45, 9, 57, 25, 33,
        28, 8, 6, 32, 27, 22),
    "nt" =
        c(38, 114, 69, 1533, 355, 59, 945, 632, 278, 1916, 873, 263,
        291, 858, 154, 207, 251, 151, 174, 209, 391, 680),
    "rc" =
        c(3, 14, 11, 127, 27, 6, 152, 48, 37, 188, 52, 47, 16, 45, 31,
        38, 12, 6, 3, 40, 43, 39),
    "nc" =
        c(39, 116, 93, 1520, 365, 52, 939, 471, 282, 1921, 583, 266,
        293, 883, 147, 213, 122, 154, 134, 218, 364, 674),
    "Num" =
        22)
```

The model functions for DD are as follows.

```
DC.MLE.fn = function() {
  for (k in 1:ncl){
    for (i in 1:Num) {
      rt[i,k] ~ dbin(pt[i,k], nt[i,k])
      rc[i,k] ~ dbin(pc[i,k], nc[i,k])
      logit(pc[i,k]) <- mu[i,k]
      logit(pt[i,k]) <- mu[i,k] + delta
      mu[i,k] ~ dnorm(alpha,tau1)
```

```
    }
  }
  # Priors
  parms ~ dmnorm(MuP, PrecP)
  delta <- parms[1]
  tau1 <- exp(parms[3])
  sigma1 <- 1/sqrt(tau1)
  alpha <- parms[2]
}

# Analysis
rt = dcdim(array(OR.data$rt, dim=c(22,1)))
nt = dcdim(array(OR.data$nt, dim=c(22,1)))
rc = dcdim(array(OR.data$rc, dim=c(22,1)))
nc = dcdim(array(OR.data$nt, dim=c(22,1)))
Num = OR.data$Num

dat = list(rt=rt, rc=rc, nt=nt, nc=nc, Num=Num, ncl=1,
    MuP=rep(0, 3), PrecP=diag(0.01, 3, 3))

DC.MLE = dc.fit(data=dat, params="parms", model=DC.MLE.fn,
    n.clones=c(1, 4, 16),
    multiply="ncl", unchanged=c("Num","MuP","PrecP"),
    n.chains=5, n.update=1000, n.iter=5000, n.adapt=2000)
```

```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 44
##    Unobserved stochastic nodes: 23
##    Total graph size: 200
##
## Initializing model
##
##
## Fitting model with 4 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 176
##    Unobserved stochastic nodes: 89
##    Total graph size: 728
##
## Initializing model
##
##
## Fitting model with 16 clones
##
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 704
##    Unobserved stochastic nodes: 353
##    Total graph size: 2840
##
## Initializing model
```

```
## Warning in rjags::jags.model(model, data, n.chains = n.chains, n.adapt =
## n.adapt, : Adaptation incomplete
```

```
## NOTE: Stopping adaptation
```

```
# Check the convergence and MLE estimates etc.
summary(DC.MLE)
```

```
##
## Iterations = 3001:8000
## Thinning interval = 1
## Number of chains = 5
## Sample size per chain = 5000
## Number of clones = 16
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##             Mean      SD   DC SD  Naive SE Time-series SE R hat
## parms[1] -0.1955 0.01231 0.04924 7.786e-05      0.0002802 1.002
## parms[2] -2.2617 0.02923 0.11693 1.849e-04      0.0008561 1.001
## parms[3]  1.3682 0.08470 0.33881 5.357e-04      0.0057503 1.016
##
## 2. Quantiles for each variable:
##
##             2.5%     25%     50%     75%   97.5%
## parms[1] -0.2197 -0.2039 -0.1955 -0.1871 -0.1709
## parms[2] -2.3210 -2.2806 -2.2615 -2.2419 -2.2050
## parms[3]  1.1986  1.3089  1.3711  1.4289  1.5240
```

```
dctable(DC.MLE)
```

```
## $`parms[1]`
##   n.clones       mean         sd       2.5%        25%        50%        75%
## 1        1 -0.1973707 0.04882413 -0.2922414 -0.2303891 -0.1968447 -0.1656099
## 2        4 -0.1953488 0.02456418 -0.2437239 -0.2114546 -0.1957253 -0.1786423
## 3       16 -0.1955118 0.01231011 -0.2196742 -0.2039059 -0.1954575 -0.1870530
##         97.5%    r.hat
## 1 -0.09902557 1.001068
## 2 -0.14685750 1.004636
## 3 -0.17093008 1.001770
##
## $`parms[2]`
##   n.clones      mean         sd      2.5%       25%       50%       75%
## 1        1 -2.262106 0.11970014 -2.500912 -2.339115 -2.265262 -2.183176
## 2        4 -2.263658 0.05930786 -2.382808 -2.302366 -2.263866 -2.222933
## 3       16 -2.261720 0.02923260 -2.321001 -2.280571 -2.261458 -2.241946
```

```
##       97.5%    r.hat
## 1 -2.027982 1.004418
## 2 -2.147162 1.002644
## 3 -2.204990 1.000899
##
## $`parms[3]`
##   n.clones     mean         sd      2.5%      25%      50%      75%    97.5%
## 1        1 1.279883 0.34168799 0.5891004 1.047885 1.297497 1.522527 1.934923
## 2        4 1.345197 0.17618020 0.9991377 1.226808 1.346657 1.466221 1.677294
## 3       16 1.368193 0.08470238 1.1986150 1.308896 1.371087 1.428885 1.523956
##      r.hat
## 1 1.014400
## 2 1.035741
## 3 1.015575
##
## attr(,"class")
## [1] "dctable"
```

```
dcdiag(DC.MLE)
```

```
##   n.clones  lambda.max    ms.error    r.squared    r.hat
## 1        1 0.116837770 0.086742336 0.0102639556 1.015434
## 2        4 0.031053136 0.002988072 0.0002607811 1.033455
## 3       16 0.007174575 0.015756814 0.0021480634 1.015211
```

It should be clear by now that we can modify any Bayesian analysis to get Maximum likelihood estimate quite easily by adding one dimension to the data and a do loop over the clones.

In the next part, we will discuss how to analyzed time series data.

Time series

In this part, we will demonstrate how one can use MCMC and data cloning to analyze time series data sets. These examples can be extended to longitudinal data sets, spatial data sets quite easily.

These models also tend to have missing observations. The code can be modified easily to account for the missing observations for estimating the parameters. We may also want to *predict* the values of the missing observations. We will demonstrate how to predict missing data or forecast future observations.

Auto-regression of order 1 (AR(1) process)

This is one of the most basic time series models. The AR(1) model can be written as:

$$Y_i = \rho Y_{i-1} + \epsilon_i$$

where $i = 1, 2, ..., n$ and $\epsilon_i \sim N(0, \sigma^2)$ are indepedent random variables (N is shorthand for the Normal distribution).

This model says that the next year's value is related to the past year's value. That is, the value in the past year is a good predictor for the next year's value. Hence the term 'auto-regression'. This model can be used to model many different phenomena that proceed in time. For example, tomorrow's temperature can be predicted using today's temperature (except in Alberta!). Next day's stock price is likely to be related to today's price and so on.

This model can be modified to include covariates. Hence, we can write:

$$Y_i = X_i\beta + \rho(Y_{i-1} - X_{i-1}\beta) + \epsilon_i$$

This allows for correlated environmental noise in regression.

This model has been used to model changes in wildlife populations. It is useful in epidemiology and so on. Many econometric models are derived from this basic model.

Of course, reality is most of the times more complicated than this model. For example, one may not observe the response without error. This is called an observation error. We may not (most of the times, we will not) observe the true population size but only an estimate (or an index) of the true population size. Thus, the observed value is not the true response. Such cases are modelled using a hierarchical structure:

- Hierarchy 1 (True state model): $X_i = \rho X_{i-1} + \epsilon_i$
- Hierarchy 2 (Observation model): $Y_i = X_i + \eta_i$ where η_i is observation error and $\eta_i \sim N(0, \tau^2)$ are independent random variables

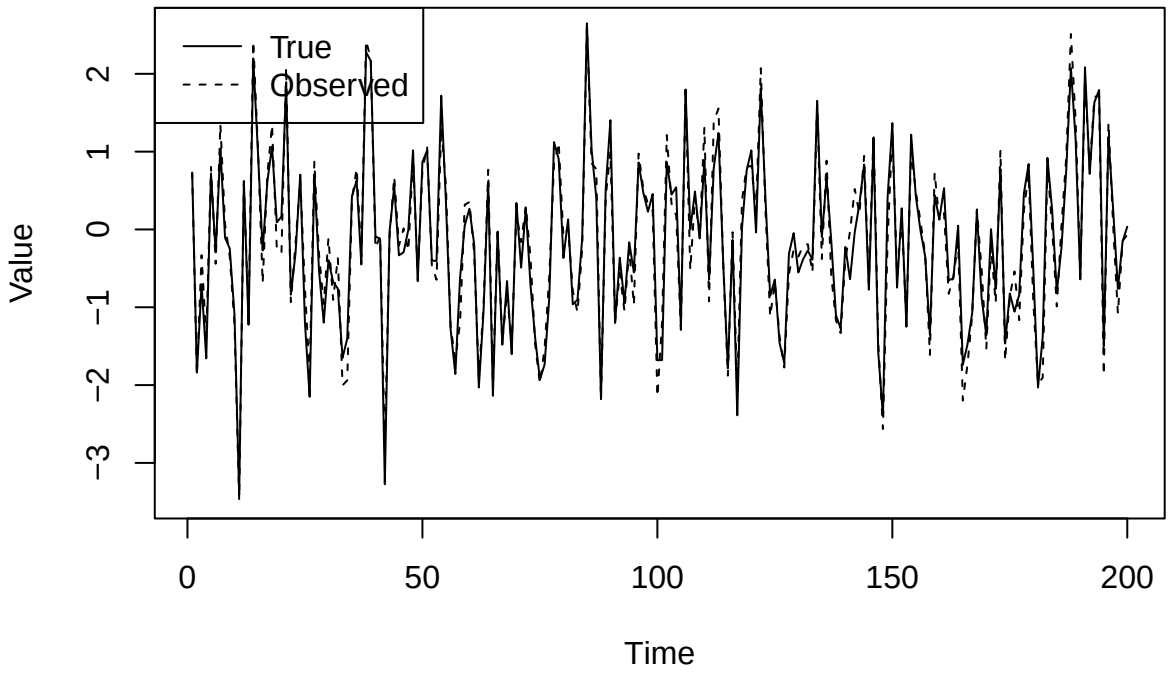
This is what is called a 'Kalman filter' after a famous Hungarian electrical engineer, Professor Rudolf Kalman. This is a particular case of the model class 'State space models'. They consist of at least two hierarchies: one models the true underlying phenomenon and the other the observation process that models the error due to observation process.

Under the Normal distribution assumption, the mathematics can be worked out for the simple linear model to conduct the likelihood inference. But once we enter the non-linear time series modelling or non-Gaussian observation processes, mathematics become nearly impossible. We will see an example of this a bit later. It will also illustrate why the MCMC algorithm is considered one of the greatest inventions in modern science.

For the time being, let us avoid all the mathematics and see if we can use JAGS and dclone to conduct the statistical analysis.

No math please!!! – this is our motto.

```
library(dclone)
T_max = 200  # Number of time steps
rho = 0.3
sigma.e = 1
tau.e = 0.25
X = rep(0, T_max)

# This is the stationary distribution for the AR(1) process
X[1] = rnorm(1, 0, sigma.e/sqrt(1-rho^2))

for (t in 2:(T_max)){
  X[t] = rho*X[t-1] + rnorm(1, 0, sigma.e)
}

# Add observation error
Y = rnorm(length(X), X, tau.e)

plot(Y, type="l", lty=2, xlab="Time", ylab="Value")
lines(X)
legend("topleft", lty=c(1, 2), legend=c("True", "Observed"))
```



We will start with the Bayesian approach.

```
AR1_Bayes_model = function(){
  # Likelihood
  prec.1 <- (1-rho*rho) * prec.e
  X[1] ~ dnorm(0, prec.1)
  Y[1] ~ dnorm(X[1], prec.t)
  for (t in 2:T_max){
    mu[t] <- rho * X[(t-1)]
    X[t] ~ dnorm(mu[t], prec.e)
    Y[t] ~ dnorm(X[t], prec.t)
  }
  # Priors
  rho ~ dunif(-1, 1)
  prec.e ~ dgamma(0.1, 0.1)
  prec.t ~ dgamma(0.1, 0.1)
}
```

Get the data and run the analysis.

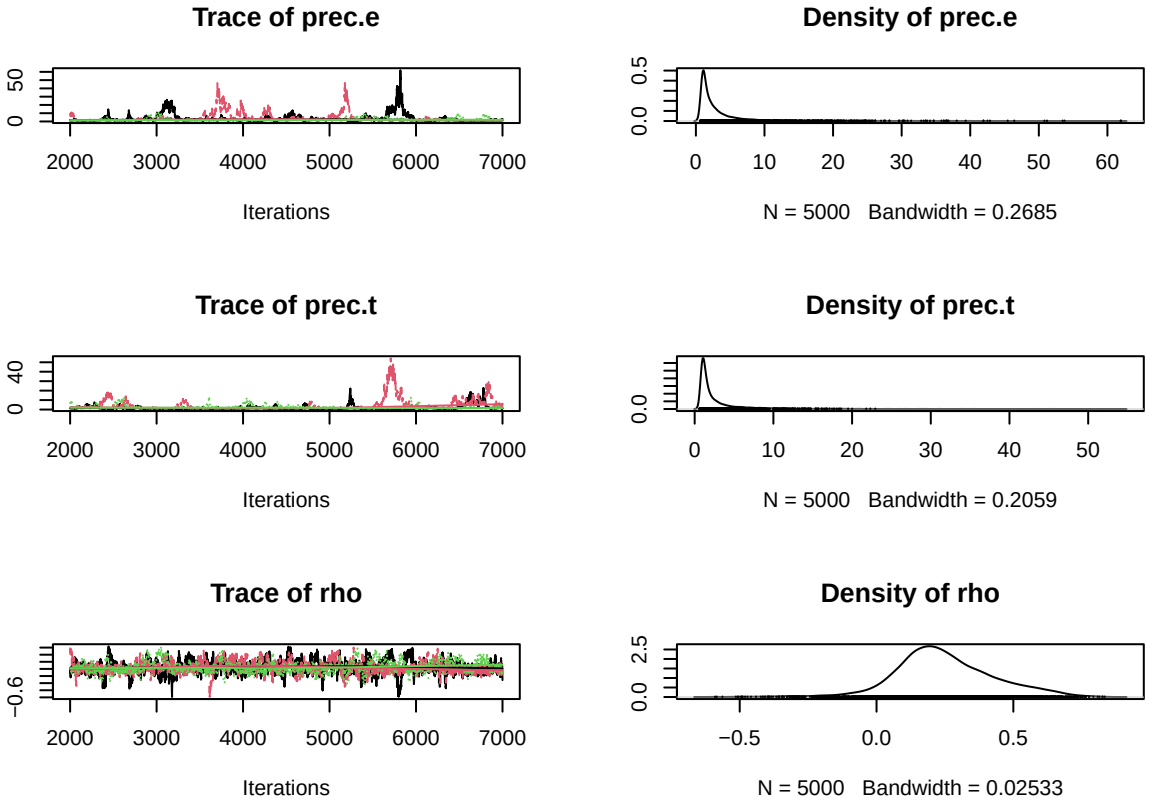
```
Y = as.vector(Y)
dat = list(Y=Y, T_max=T_max)
ini = list(X=Y)
AR1_Bayes_fit = jags.fit(data=dat, params=c("rho","prec.e","prec.t"), model=AR1_Bayes_model)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 200
##    Unobserved stochastic nodes: 203
##    Total graph size: 610
##
## Initializing model
```

```
summary(AR1_Bayes_fit)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## prec.e 3.2672 4.5219  0.03692       0.549416
## prec.t 2.8826 4.2343  0.03457       0.514753
## rho    0.2495 0.1788  0.00146       0.008952
##
## 2. Quantiles for each variable:
##
##            2.5%   25%    50%    75%   97.5%
## prec.e  0.77934 1.102 1.7671 3.4243 16.4790
## prec.t  0.76952 1.053 1.4765 2.8346 13.7220
## rho    -0.08395 0.137 0.2316 0.3561  0.6336
```

```
plot(AR1_Bayes_fit)
```



We will modify this to get the MLE using data cloning.

```
AR1_DC_model = function(){
  # Likelihood
```

```
  for (k in 1:ncl){
    X[1,k] ~ dnorm(0, prec.1)
    Y[1,k] ~ dnorm(X[1,k], prec.t)
    for (t in 2:T_max){
        mu[t,k] <- rho*X[(t-1),k]
        X[t,k] ~ dnorm(mu[t,k], prec.e)
        Y[t,k] ~ dnorm(X[t,k], prec.t)
    }
  }
  # Priors
  rho ~ dunif(-1, 1)
  prec.e ~ dgamma(0.1, 0.1)
  prec.t ~ dgamma(0.1, 0.1)
  prec.1 <- (1-rho*rho) * prec.e
}
```

Get the data and run the analysis with data cloning.

```
Y = array(Y, dim=c(length(Y), 1))
Y = dcdim(Y)
dat = list(Y=Y, T_max=T_max, ncl=1)
ini = list(X=Y)
initfn = function(model, n.clones){
  return(list(X=dclone(Y, n.clones)))
}
AR1_DC_fit = dc.fit(data=dat,params=c("rho","prec.t","prec.e"),model=AR1_DC_model,
    unchanged="T_max",multiply="ncl",
    n.clones=c(1, 10, 20),
    inits=ini, initsfun=initfn)
```

```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 200
##    Unobserved stochastic nodes: 203
##    Total graph size: 611
##
## Initializing model
##
##
## Fitting model with 10 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 2000
##    Unobserved stochastic nodes: 2003
##    Total graph size: 6002
##
## Initializing model
```

```
##
##
## Fitting model with 20 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 4000
##    Unobserved stochastic nodes: 4003
##    Total graph size: 11992
##
## Initializing model
```

```
## Warning in dclone::.dcFit(data, params, model, inits, n.clones, multiply =
## multiply, : chains convergence problem, see R.hat values
```

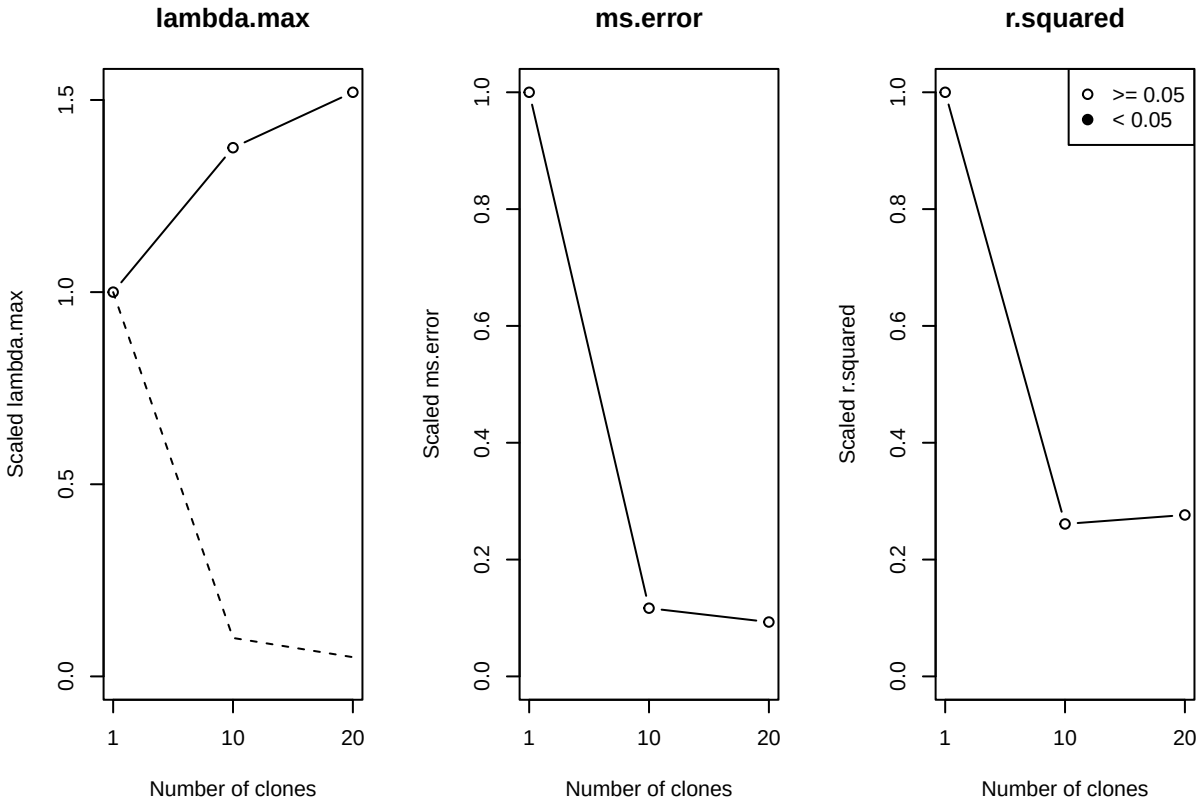
```
summary(AR1_DC_fit)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 20
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##          Mean     SD   DC SD Naive SE Time-series SE R hat
## prec.e 1.4855 0.5100  2.2806 0.004164        0.11983 1.225
## prec.t 3.8956 5.1014 22.8140 0.041653        1.81300 1.846
## rho    0.2366 0.0692  0.3095 0.000565        0.01449 1.268
##
## 2. Quantiles for each variable:
##
##          2.5%    25%    50%    75%   97.5%
## prec.e 0.8143 1.0746 1.4047 1.8396  2.5926
## prec.t 1.1835 1.4449 1.8639 3.0632 21.5614
## rho    0.1289 0.1794 0.2306 0.2879  0.3765
```

```
dcdiag(AR1_DC_fit)
```

```
##   n.clones lambda.max  ms.error r.squared    r.hat
## 1        1   17.17816 31.630965 0.3936079 1.055714
## 2       10   23.63365  3.693459 0.1027215 1.162062
## 3       20   26.11184  2.943588 0.1087841 1.539028
```

```
plot(dcdiag(AR1_DC_fit))
```



Try running the above code when true `rho=0`.

- Are the parameters estimable?
- Does the Bayesian approach tell you that?

Without the estimability diagnostics, you could be easily mislead by the Bayesian approach. You will merrily go around with the scientific inference when the parameters are not estimable.

Different types of measurement errors

1: Clipped or Censored time series

Suppose the underlying process is AR(1) but the observed process is a clipped process such that it is 1 if X is positive and 0 if X is negative. This is called a clipped time series. Similarly you may observe Y to belong to an interval. This is an interval censored data. The above model can be modified to accommodate such a process. An easier way to model binary, count or proportion time series is as follows.

Modelling binary and count data time series For modelling binary time series, we can consider the observation process as:

$Y_t \sim Bernoulli(p_t)$ where $log(\frac{p_t}{1-p_t}) = \gamma * X_t$

For modelling count data time series, we can consider

$Y_t \sim Poisson(\lambda_t)$ where $log\lambda_t = \gamma * X_t$.

This is a time series generalization of the GLMM that we considered before.

Caveat: It is extremely important that you check for the estimability of the parameters for these models. Because of clipping and other observation processes, you are more likely to run into estimability issues. As far as we know, data cloning is the only method that allows estimability diagnostics as part of the estimation process. See Lele (2010).

Non-linear time series analysis Now we will consider a non-linear time series model, Beverton-Holt growth model, that is commonly used in ecology.

In ecology and population biology, one wants to understand how abundance changes over time. Following Malthus' thinking, it is also evident that, in a finite environment, abundance cannot increase without limit.

Thus, the growth is usually exponential at the beginning (low population, ignore the Allee effect for now) and then it slows down as we approach the carrying capacity. One commonly used model is the Beverton-Holt model (discrete analog to the continuous Logistic model):

Let $log(N_t) = X_t$ where N_t is the abundance at time t. A general form for the population growth models is:

$X_t = m_t + \sigma Z_t$

where Z_t is $Normal(0, 1)$ random variable. This is similar to the AR(1) process but with a non-linear mean structure.

If $m_t = log(\lambda) + x_t - log(1 + \beta N_t)$, the population growth model is called the Beverton-Holt model. It has an upper limit $K = \frac{\lambda - 1}{\beta}$ called the Carrying capacity, the maximum population size that can be attained (with some perturbation).

In practice, we usually have to conduct some sampling to *estimate* the abundance. Hence there is measurement error. We can represent this process by using hierarchical model:

- Hierarchy 1: Process model, $X_t | X_{t-1} = x_{t-1} \sim Normal(m(x_{t-1}), \sigma^2)$
- Hierarchy 2: Observation model, $Y_t | N_t \sim Poisson(N_t)$

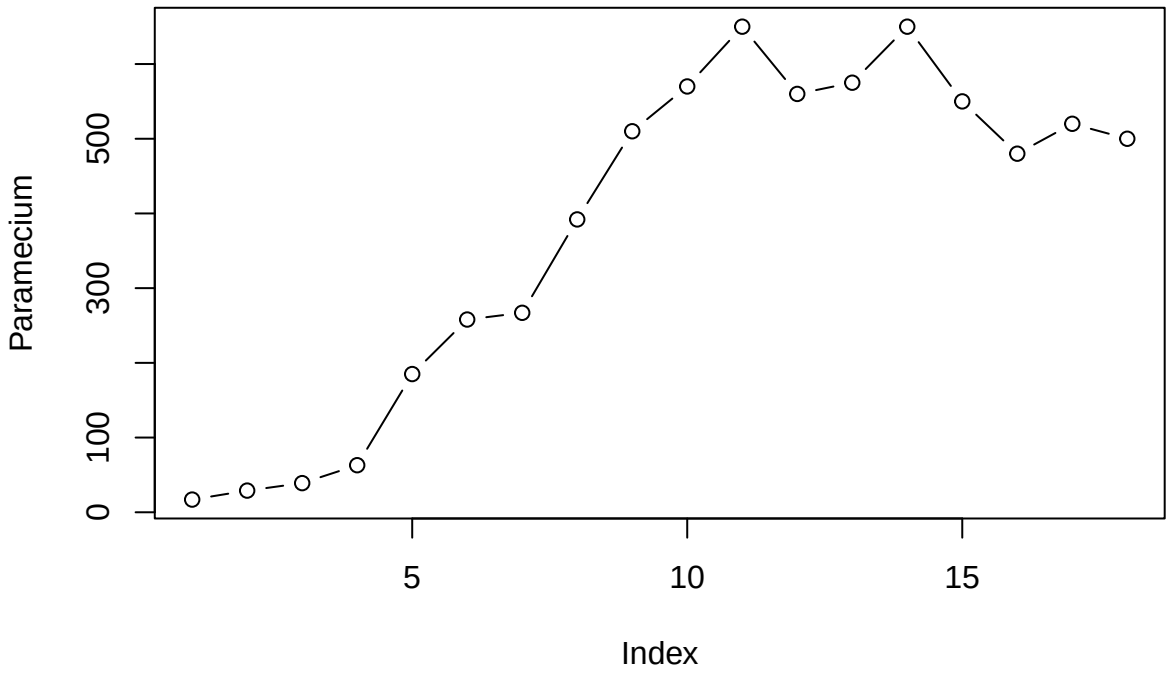
One can use other observation models as well. We can write the likelihood function for this using a T_{max} (length of the time series) dimensional integral. If the time series is of length 30, this will be a 30 dimensional integral. In order to compute the MLE, we will need to evaluate this integral repeatedly until the numerical optimization routine converges. This is a nearly impossible task.

The code to analyze this non-linear time series with non-Gaussian observation error can be written as follows.

```
BH_Bayes_fn= function() {
  # Likelihood
  X[1] ~ dnorm(mu0, 1 / sigma^2) # Initial condition
  for(i in 2:(n+1)){
    Y[(i-1)] ~ dpois(exp(X[i]))
    X[i] ~ dnorm(mu[i], 1 / sigma^2)
    mu[i] <- X[(i-1)] + log(lambda) - log(1 + beta * exp(X[(i-1)]))
  }

  # Priors on model parameters: They are on the real line.
  ln.beta ~ dnorm(0, 0.1)
  ln.sigma ~ dnorm(0, 0.1)
  ln.tmp ~ dnorm(0, 0.1)

  # Parameters on the natural scale
  beta <- exp(ln.beta)
  sigma <- exp(ln.sigma)
  tmp <- exp(ln.tmp)
  lambda <- tmp + 1
  mu0 <- log(2)  + log(lambda) - log(1 + beta * 2)
}
```

Gause's *Paramecium* data.

```
Paramecium = c(17,29,39,63,185,258,267,392,510,570,650,560,575,650,550,480,520,500)
plot(Paramecium, type="b")
```



Bayesian analysis.

```
Y = Paramecium
dat = list(n=length(Y), Y=Y)
BH_Bayes_fit = jags.fit(data=dat, params=c("ln.tmp","ln.beta","ln.sigma"), model=BH_Bayes_fn)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 18
##    Unobserved stochastic nodes: 22
##    Total graph size: 167
##
## Initializing model
```

```
summary(BH_Bayes_fit)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## ln.beta   -6.159 0.1682 0.001373       0.004658
## ln.sigma -1.963 0.2794 0.002282       0.006859
## ln.tmp     0.152 0.1288 0.001051       0.003569
##
## 2. Quantiles for each variable:
##
```

```
##             2.5%      25%     50%     75%   97.5%
## ln.beta  -6.5144 -6.25754 -6.1535 -6.0553 -5.8375
## ln.sigma -2.4898 -2.15673 -1.9697 -1.7766 -1.3889
## ln.tmp   -0.1236  0.07691  0.1545  0.2324  0.4014
```

We can easily modify this program to obtain the MLE and its asymptotic variance.

```
BH_DC_fn= function() {
  # Likelihood
  for (k in 1:ncl) {
    for(i in 2:(n+1)){
      Y[(i-1), k] ~ dpois(exp(X[i, k]))
      X[i, k] ~ dnorm(mu[i, k], 1 / sigma^2)
      mu[i, k] <- X[(i-1), k] + log(lambda) - log(1 + beta * exp(X[(i-1), k]))
    }
    X[1, k] ~ dnorm(mu0, 1 / sigma^2)
  }

  # Priors on model parameters: They are on the real line.
  ln.beta ~ dnorm(0, 0.1)
  ln.sigma ~ dnorm(0, 0.1)
  ln.tmp ~ dnorm(0, 0.1)

  # Parameters on the natural scale
  beta <- exp(ln.beta)
  sigma <- exp(ln.sigma)
  tmp <- exp(ln.tmp)
  lambda <- tmp + 1
  mu0 <- log(2)  + log(lambda) - log(1 + beta * 2)
}
```

Assemble the data and fit with data cloning.

```
Y = array(Y, dim=c(length(Y), 1))
Y = dcdim(Y)
dat = list(ncl=1, n=18, Y=Y)

n.clones = c(1, 5, 10)
params = c("ln.tmp", "ln.beta", "ln.sigma")
BH_MLE = dc.fit(data=dat, params=params, model=BH_DC_fn,
    n.clones=n.clones,
    multiply="ncl", unchanged="n",
    n.chains=5, n.update=1000, n.iter=5000, n.adapt=2000)
```

```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 18
##    Unobserved stochastic nodes: 22
##    Total graph size: 168
##
## Initializing model
```

```
##
##
## Fitting model with 5 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 90
##    Unobserved stochastic nodes: 98
##    Total graph size: 752
##
## Initializing model
##
##
## Fitting model with 10 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 180
##    Unobserved stochastic nodes: 193
##    Total graph size: 1482
##
## Initializing model
```

```
dcdiag(BH_MLE)
```

```
##   n.clones  lambda.max    ms.error   r.squared    r.hat
## 1        1 0.079318280 1.340935091 0.078280947 1.002020
## 2        5 0.014781776 0.036431139 0.003579339 1.003387
## 3       10 0.007119748 0.009173014 0.001175542 1.004161
```

Parameters for this model are estimable. It will be interesting to see if one can use Negative Binomial distribution (one additional parameter) instead of the Poisson distribution. Are the parameters still estimable? (TBD!!!)

Prediction

We are also interested in predicting the true population abundances as well as forecasting the future trajectory of the abundances. This is quite easy under the Bayesian paradigm. The frequentist paradigm involves an additional step.

Let us see how to use the Bayesian paradigm and MCMC to do this. In the Bayesian paradigm, there is no difference between parameters and the unobserved states. They both are considered random variables.

On the other hand, in the frequentist paradigm parameters are fixed but unknown (not random) whereas the unobserved states are true random variables.

We (the instructors) consider these to be different.

1. Information about the parameters converges to infinity as the sample size increases. Thus, we can *estimate* them with high degree of confidence. The *confidence* intervals shrink as we increase the sample size.
2. Information about the states (random variables) does not converge to infinity as the sample size increases. The *prediction* intervals do not shrink.

This should be familiar to most of you from your regression class.

```
BH_Bayes_fn= function() {
  # Likelihood
  X[1] ~ dnorm(mu0, 1 / sigma^2) # Initial condition
  N[1] <- exp(X[1])
  for(i in 2:(n+1)){
    Y[(i-1)] ~ dpois(exp(X[i]))
    X[i] ~ dnorm(mu[i], 1 / sigma^2)
    mu[i] <- X[(i-1)] + log(lambda) - log(1 + beta * exp(X[(i-1)]))
    N[i] <- exp(X[i])
  }

  # Priors on model parameters: They are on the real line.
  ln.beta ~ dnorm(0, 0.1)
  ln.sigma ~ dnorm(0, 0.1)
  ln.tmp ~ dnorm(0, 0.1)

  # Parameters on the natural scale
  beta <- exp(ln.beta)
  sigma <- exp(ln.sigma)
  tmp <- exp(ln.tmp)
  lambda <- tmp + 1
  mu0 <- log(2)  + log(lambda) - log(1 + beta * 2)
}

# Gause's data
Y = Paramecium
dat = list(n=length(Y), Y=Y)
BH_Bayes_fit = jags.fit(data=dat, params=c("N"), model=BH_Bayes_fn)
```

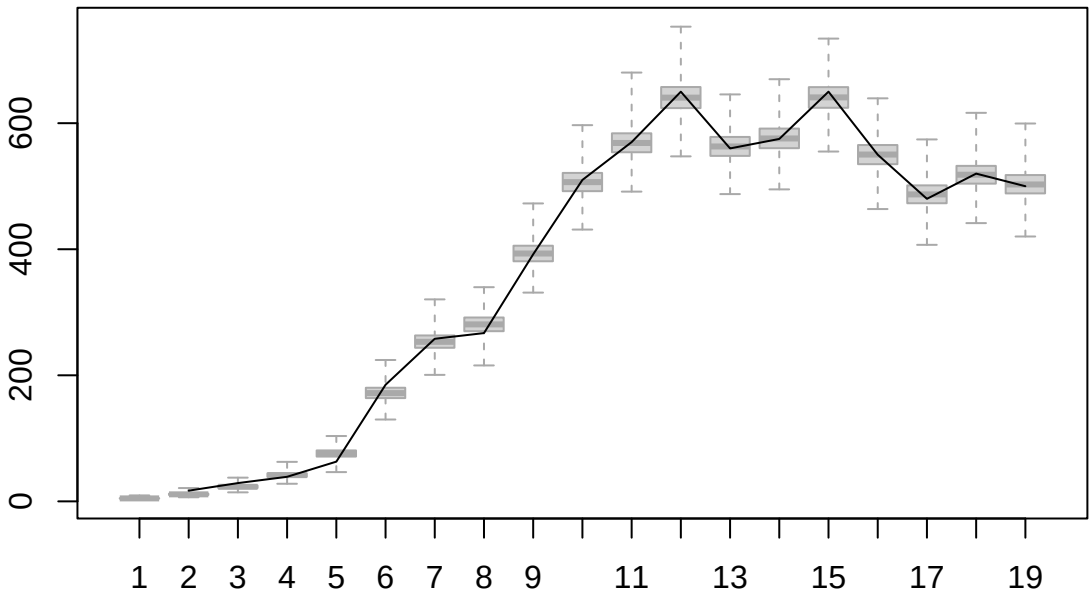
```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 18
##    Unobserved stochastic nodes: 22
##    Total graph size: 167
##
## Initializing model
```

```
summary(BH_Bayes_fit)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## N[1]    4.788  0.6471 0.005284        0.01109
## N[2]   11.337  1.7078 0.013944        0.03597
```

```
## N[3]   23.408  3.0427 0.024844        0.06047
## N[4]   41.788  4.2906 0.035033        0.06095
## N[5]   75.921  7.1091 0.058045        0.11332
## N[6]  172.312 12.1629 0.099309        0.16468
## N[7]  253.483 14.5640 0.118915        0.16141
## N[8]  280.953 16.2334 0.132545        0.21153
## N[9]  393.483 18.3419 0.149761        0.19432
## N[10] 506.843 21.3347 0.174197        0.22663
## N[11] 568.842 22.3126 0.182182        0.23379
## N[12] 641.004 24.8118 0.202588        0.27861
## N[13] 563.353 22.6504 0.184940        0.22956
## N[14] 576.108 22.9901 0.187713        0.24476
## N[15] 641.006 24.3622 0.198916        0.26669
## N[16] 550.315 22.3359 0.182372        0.23074
## N[17] 487.096 20.8929 0.170590        0.23019
## N[18] 518.383 21.1955 0.173060        0.21651
## N[19] 503.344 21.6138 0.176476        0.22738
##
## 2. Quantiles for each variable:
##
##          2.5%     25%     50%     75%   97.5%
## N[1]    3.738   4.346   4.713   5.138   6.286
## N[2]    8.598  10.145  11.122  12.285  15.320
## N[3]   18.320  21.250  23.126  25.225  30.354
## N[4]   33.793  38.871  41.641  44.531  50.767
## N[5]   62.081  71.106  75.885  80.810  89.695
## N[6]  150.252 163.843 171.747 180.238 197.466
## N[7]  225.541 243.659 253.152 263.210 282.845
## N[8]  249.424 269.992 280.783 291.714 312.883
## N[9]  358.530 380.926 393.235 405.600 430.507
## N[10] 466.128 492.154 506.524 521.096 549.515
## N[11] 525.198 553.893 568.640 583.957 612.522
## N[12] 594.136 623.969 640.536 657.382 690.400
## N[13] 518.619 548.142 563.084 578.202 608.372
## N[14] 532.357 560.375 575.763 591.459 622.484
## N[15] 594.161 624.437 640.922 657.228 689.650
## N[16] 507.098 534.972 550.239 565.370 594.468
## N[17] 446.663 473.019 487.009 501.341 528.436
## N[18] 477.772 504.098 518.243 532.297 561.471
## N[19] 461.773 488.626 502.807 517.715 546.875
```

```
boxplot(unname(as.matrix(BH_Bayes_fit)), range=0, border="darkgrey")
lines(c(NA, Y))
```



If we want to obtain predictions that are invariant to parameterization and correct in the frequentist sense, we need to modify this approach slightly. We need to change the prior distribution to the asymptotic distribution of the MLE. This can be done quite easily as follows.

```
BH_DC_pred_fn= function() {
  # Likelihood
  X[1] ~ dnorm(mu0, 1 / sigma^2) # Initial condition
  N[1] <- exp(X[1])
  for(i in 2:(n+1)){
    Y[(i-1)] ~ dpois(exp(X[i]))
    X[i] ~ dnorm(mu[i], 1 / sigma^2)
    mu[i] <- X[(i-1)] + log(lambda) - log(1 + beta * exp(X[(i-1)]))
    N[i] <- exp(X[i])
  }

  # Priors on model parameters: they are on the real line.
  parms ~ dmnorm(MuPost,PrecPost)
  ln.beta <- parms[1]
  ln.sigma <- parms[2]
  ln.tmp <- parms[3]

  # Parameters on the natural scale
  beta <- exp(ln.beta)
  sigma <- exp(ln.sigma)
  tmp <- exp(ln.tmp)
  lambda <- tmp + 1
  mu0 <- log(2)  + log(lambda) - log(1 + beta * 2)
}

# Gause's data
Y = Paramecium
dat = list(n=length(Y), Y=Y, MuPost=coef(BH_MLE), PrecPost=solve(vcov(BH_MLE)))
BH_DC_Pred = jags.fit(data=dat, params=c("N"), model=BH_DC_pred_fn)
```

```
## Compiling model graph
##    Resolving undeclared variables
```

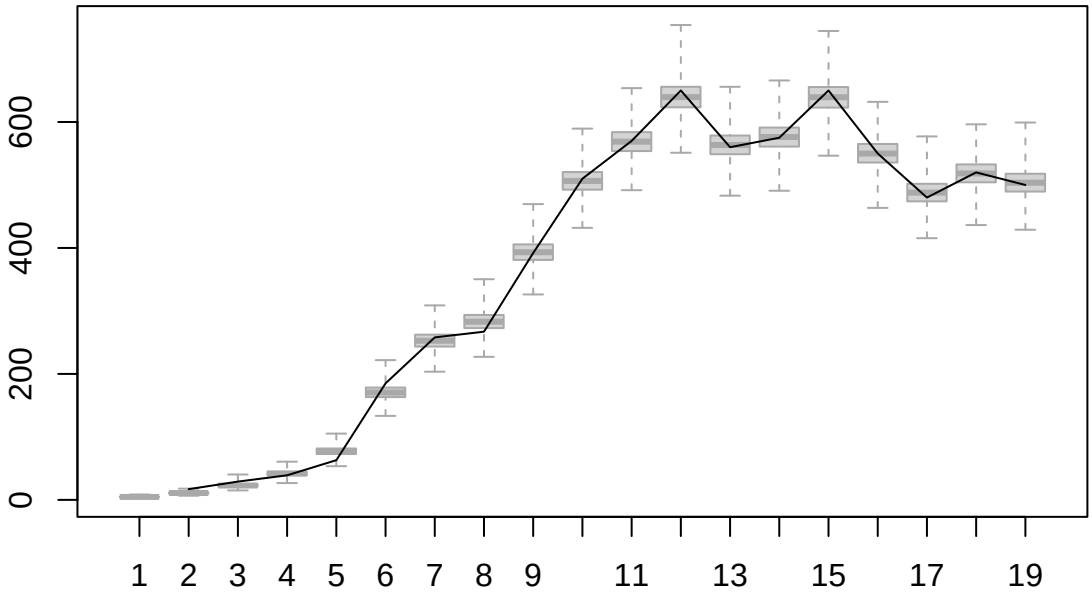
```
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 18
##    Unobserved stochastic nodes: 20
##    Total graph size: 180
##
## Initializing model
```

```
summary(BH_DC_Pred)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## N[1]    4.722  0.5448 0.004449       0.009378
## N[2]   11.039  1.4451 0.011799       0.037977
## N[3]   23.004  2.7313 0.022301       0.063120
## N[4]   42.019  4.1807 0.034135       0.066155
## N[5]   77.229  6.5877 0.053788       0.102595
## N[6]  170.861 11.2605 0.091942       0.175478
## N[7]  253.011 14.1126 0.115229       0.160916
## N[8]  283.298 15.4386 0.126056       0.232494
## N[9]  393.533 18.1970 0.148578       0.198947
## N[10] 506.786 20.7812 0.169678       0.221530
## N[11] 569.154 22.2417 0.181603       0.233948
## N[12] 639.742 24.1506 0.197189       0.260184
## N[13] 563.663 22.1869 0.181155       0.233501
## N[14] 576.371 22.6210 0.184700       0.237258
## N[15] 639.674 24.1764 0.197400       0.270523
## N[16] 550.480 21.9210 0.178984       0.229432
## N[17] 487.986 20.6620 0.168705       0.218373
## N[18] 518.655 21.3568 0.174378       0.221537
## N[19] 503.869 21.0004 0.171467       0.220335
##
## 2. Quantiles for each variable:
##
##          2.5%    25%     50%     75%  97.5%
## N[1]    3.788   4.35   4.678   5.039   5.92
## N[2]    8.643  10.04  10.892  11.862  14.39
## N[3]   18.320  21.10  22.777  24.679  28.96
## N[4]   34.348  39.15  41.850  44.724  50.64
## N[5]   64.593  72.78  77.142  81.605  90.29
## N[6]  149.909 163.13 170.470 178.343 193.82
## N[7]  226.550 243.38 252.679 262.332 281.55
## N[8]  253.510 272.79 282.843 293.615 313.94
## N[9]  358.823 381.00 393.208 405.657 429.79
## N[10] 466.840 492.58 506.370 520.569 548.51
## N[11] 526.913 553.84 568.811 583.960 613.59
## N[12] 592.928 623.35 639.479 655.902 687.98
```

```
## N[13] 520.962 548.83 563.455 578.486 607.37
## N[14] 532.837 560.91 576.233 591.166 621.46
## N[15] 593.710 622.86 639.304 655.553 688.32
## N[16] 508.355 535.63 549.918 565.058 595.00
## N[17] 448.810 473.99 487.854 501.864 528.93
## N[18] 477.719 504.25 518.303 532.673 561.76
## N[19] 463.767 489.54 503.573 517.831 545.10
```

```
boxplot(unname(as.matrix(BH_DC_Pred)), range=0, border="darkgrey")
lines(c(NA, Y))
```



If we want to forecast future observations, we modify the program slightly. To do this, we pretend as if the process had run longer than T_{max} but we could not 'observe' those future states. Thus, it becomes a missing data problem. Let us see how we can do this.

```
BH_Bayes_fn= function() {
  X[1] ~ dnorm(mu0, 1 / sigma^2) # Initial condition
  N[1] <- exp(X[1])
  for(i in 2:(n+1)){
    Y[(i-1)] ~ dpois(exp(X[i]))
  }
  for (i in 2:N_future){
    X[i] ~ dnorm(mu[i], 1 / sigma^2)
    mu[i] <- X[(i-1)] + log(lambda) - log(1 + beta * exp(X[(i-1)]))
    N[i] <- exp(X[i])
  }

  # Priors on model parameters: they are on the real line.
  ln.beta ~ dnorm(0, 0.1)
  ln.sigma ~ dnorm(0, 0.1)
  ln.tmp ~ dnorm(0, 0.1)

  # Parameters on the natural scale
  beta <- exp(ln.beta)
  sigma <- exp(ln.sigma)
  tmp <- exp(ln.tmp)
```

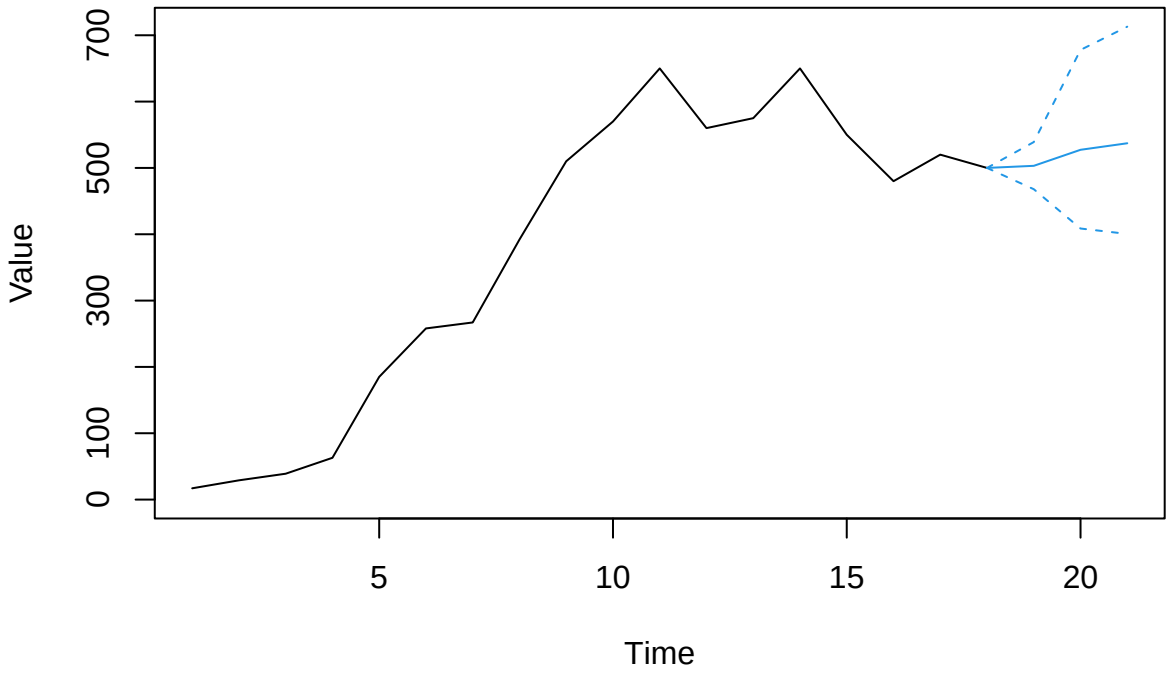
```
  lambda <- tmp + 1
  mu0 <- log(2)  + log(lambda) - log(1 + beta * 2)
}

# Gause's data
Y = Paramecium

# We want to predict 3 years in future.
dat = list(n=length(Y), Y=Y, N_future=length(Y)+3)

BH_Bayes_fit = jags.fit(data=dat, params=c("N[19:21]"), model=BH_Bayes_fn)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 18
##    Unobserved stochastic nodes: 24
##    Total graph size: 182
##
## Initializing model
```

```
summary(BH_Bayes_fit)
```

```
##
## Iterations = 2001:7000
## Thinning interval = 1
## 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
## N[19] 503.4 21.51   0.1756         0.2368
## N[20] 533.0 84.47   0.6897         0.6952
## N[21] 544.7 98.09   0.8009         0.8509
##
## 2. Quantiles for each variable:
##
##        2.5%   25%   50%   75% 97.5%
## N[19] 461.7 489.1 503.2 517.8 546.2
## N[20] 382.9 478.6 527.3 580.2 723.6
## N[21] 372.8 481.1 537.2 599.4 756.6
```

```
pred = mcmcapply(BH_Bayes_fit, quantile, c(0.05, 0.5, 0.95))
plot(c(Y, NA, NA, NA), type="l", ylim=c(0, max(Y, pred)), xlab="Time", ylab="Value")
matlines(c(18:21),
    t(cbind(rep(Y[length(Y)], 3), pred)),
    col=4, lty=c(2,1,2))
```



To get the frequentist version, we modify this prediction function similarly. The prior is equal to the asymptotic distribution of the MLE. We will leave it to you to try that out.

Other considerations

What is going on inside jags.fit

```
library(dclone)

n = 30
X1 = rnorm(n)
X = model.matrix(~X1)
beta.true = c(0.5, 1)
link_mu = X %*% beta.true

# Linear regression model
mu = link_mu
sigma.e = 1
Y = rnorm(n,mean=mu,sd=sigma.e)

Normal.model = function(){
  # Likelihood
  for (i in 1:n){
    mu[i] <- X[i,] %*% beta
    Y[i] ~ dnorm(mu[i],prec.e)
  }
  # Prior
  beta[1] ~ dnorm(0, 1)
  beta[2] ~ dnorm(0, 1)
  prec.e ~ dlnorm(0, 1)
}

dat = list(Y=Y, X=X, n=n)
```

```
Normal.Bayes = jags.fit(data=dat, params=c("beta","prec.e"), model=Normal.model)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 3
##    Total graph size: 157
##
## Initializing model
```

What just happened? `jags.fit` is a wrapper around some rjags functions.

```
m <- jagsModel(file = Normal.model, data=dat, n.chains=3)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 3
##    Total graph size: 157
##
## Initializing model
```

```
m
```

```
## JAGS model:
##
## model
## {
##     for (i in 1:n) {
##         mu[i] <- X[i, ] %*% beta
##         Y[i] ~ dnorm(mu[i], prec.e)
##     }
##     beta[1] ~ dnorm(0.00000E+00, 1)
##     beta[2] ~ dnorm(0.00000E+00, 1)
##     prec.e ~ dlnorm(0.00000E+00, 1)
## }
## Fully observed variables:
##  X Y n
```

```
str(m)
```

```
## List of 8
##  $ ptr      :function ()
##  $ data     :function ()
##  $ model    :function ()
##  $ state    :function (internal = FALSE)
##  $ nchain   :function ()
##  $ iter     :function ()
##  $ sync     :function ()
##  $ recompile:function ()
##  - attr(*, "class")= chr "jags"
```

```
str(m$state())
```

```
## List of 3
##  $ :List of 2
##   ..$ beta  : num [1:2] 0.465 0.51
##   ..$ prec.e: num 0.729
##  $ :List of 2
##   ..$ beta  : num [1:2] 0.382 1.078
##   ..$ prec.e: num 1.1
##  $ :List of 2
##   ..$ beta  : num [1:2] 0.255 0.601
##   ..$ prec.e: num 1.16
```

```
m$iter()
```

```
## [1] 1000
```

```
update(m, n.iter=1000)
str(m$state())
```

```
## List of 3
##  $ :List of 2
##   ..$ beta  : num [1:2] 0.354 0.954
##   ..$ prec.e: num 0.868
##  $ :List of 2
##   ..$ beta  : num [1:2] -0.12 1.14
##   ..$ prec.e: num 0.785
##  $ :List of 2
##   ..$ beta  : num [1:2] 0.614 0.64
##   ..$ prec.e: num 0.947
```

```
m$iter()
```

```
## [1] 2000
```

```
s = codaSamples(m, variable.names=c("beta","prec.e"), n.iter=5000)
str(s)
```

```
## List of 3
##  $ : 'mcmc' num [1:5000, 1:3] 0.297 0.733 0.222 0.347 0.625 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:3] "beta[1]" "beta[2]" "prec.e"
##   ..- attr(*, "mcpar")= num [1:3] 2001 7000 1
##  $ : 'mcmc' num [1:5000, 1:3] 0.561 0.236 0.383 0.304 0.203 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:3] "beta[1]" "beta[2]" "prec.e"
##   ..- attr(*, "mcpar")= num [1:3] 2001 7000 1
##  $ : 'mcmc' num [1:5000, 1:3] 0.217 0.515 0.28 0.375 0.303 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:3] "beta[1]" "beta[2]" "prec.e"
##   ..- attr(*, "mcpar")= num [1:3] 2001 7000 1
##  - attr(*, "class")= chr "mcmc.list"
```

```
head(s)
```

```
## [[1]]
## Markov Chain Monte Carlo (MCMC) output:
## Start = 2001
## End = 2007
## Thinning interval = 1
##        beta[1]   beta[2]    prec.e
## [1,] 0.2965745 1.1463867 0.7142611
## [2,] 0.7334239 0.4813399 0.7484122
## [3,] 0.2222536 0.7594546 0.9784450
## [4,] 0.3471564 0.6223774 0.8319005
## [5,] 0.6252544 0.5673678 0.9899675
## [6,] 0.2430912 0.8724851 0.9888543
## [7,] 0.3876264 0.8300351 0.8535297
##
## [[2]]
## Markov Chain Monte Carlo (MCMC) output:
## Start = 2001
## End = 2007
## Thinning interval = 1
##        beta[1]   beta[2]    prec.e
## [1,] 0.5614720 0.9339085 0.7482101
## [2,] 0.2360447 0.4347648 0.7318908
## [3,] 0.3825761 1.1090671 1.0531644
## [4,] 0.3042906 1.2539124 0.9147583
## [5,] 0.2026730 0.9954057 1.3673165
## [6,] 0.1432080 0.6362736 1.6050587
## [7,] 0.3893933 0.8184627 1.0526788
##
## [[3]]
## Markov Chain Monte Carlo (MCMC) output:
## Start = 2001
## End = 2007
## Thinning interval = 1
##        beta[1]   beta[2]    prec.e
## [1,] 0.2168627 0.6591851 0.6948995
## [2,] 0.5146242 0.9977200 0.9796337
## [3,] 0.2795655 0.6898132 0.9516174
## [4,] 0.3746862 1.0390761 1.4479757
## [5,] 0.3025834 0.6041060 0.8610630
## [6,] 0.3567208 0.7711426 1.0789292
## [7,] 0.4334158 1.0928831 0.9447659
```

```
m$iter()
```

```
## [1] 7000
```

```
update(m, n.iter=1000)
m$iter()
```

```
## [1] 8000
```

```
s = codaSamples(m, variable.names=c("beta","prec.e"), n.iter=5000)
head(s)
```

```
## [[1]]
## Markov Chain Monte Carlo (MCMC) output:
```

```
## Start = 8001
## End = 8007
## Thinning interval = 1
##        beta[1]   beta[2]    prec.e
## [1,] 0.6900750 0.6587970 0.9836451
## [2,] 0.6324532 0.2726929 0.6240867
## [3,] 0.7339562 1.0351869 0.6253472
## [4,] 0.3359121 0.4203707 0.9864727
## [5,] 0.3696975 0.7394955 0.8585435
## [6,] 0.2847715 0.7354384 0.7110213
## [7,] 0.7340895 0.3767809 0.7807552
##
## [[2]]
## Markov Chain Monte Carlo (MCMC) output:
## Start = 8001
## End = 8007
## Thinning interval = 1
##        beta[1]   beta[2]    prec.e
## [1,] 0.5736244 0.7197135 0.6447050
## [2,] 0.2019368 1.1435595 0.6597455
## [3,] 0.6706809 0.6420006 0.6400720
## [4,] 0.2708777 0.5132735 0.6563516
## [5,] 0.5861063 0.3516867 0.8114697
## [6,] 0.5901346 0.7024149 0.8424021
## [7,] 0.2514644 0.9958727 0.7100265
##
## [[3]]
## Markov Chain Monte Carlo (MCMC) output:
## Start = 8001
## End = 8007
## Thinning interval = 1
##        beta[1]   beta[2]    prec.e
## [1,] 0.1955071 0.9587224 0.6198626
## [2,] 0.2666655 0.8827584 1.2043835
## [3,] 0.5401302 1.0946010 1.1667307
## [4,] 0.2690755 0.6034173 1.1830975
## [5,] 0.1064966 0.7630939 1.0074623
## [6,] 0.2572718 0.5016849 0.7997400
## [7,] 0.5293561 0.8768627 1.2621465
```

```
m$iter()
```

```
## [1] 13000
```

Can we further update `Normal.Bayes`?

```
m2 <- updated.model(Normal.Bayes)
m2$iter()
```

```
## [1] 7000
```

```
update(m2, n.iter=1000)
m2$iter()
```

```
## [1] 8000
```

Couple of things to cover here:

```r
str(formals(jags.fit))
```

```
## Dotted pair list of 11
##  $ data         : symbol
##  $ params       : symbol
##  $ model        : symbol
##  $ inits        : NULL
##  $ n.chains     : num 3
##  $ n.adapt      : num 1000
##  $ n.update     : num 1000
##  $ thin         : num 1
##  $ n.iter       : num 5000
##  $ updated.model: logi TRUE
##  $ ...          : symbol
```

Implicitly, we do:

```r
Normal.Bayes = jags.fit(
    data=dat,
    params=c("beta","prec.e"),
    model=Normal.model,
    inits=NULL,
    n.chains=3,
    n.adapt=1000,
    n.update=1000,
    thin=1,
    n.iter=5000,
    updated.model=TRUE)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 3
##    Total graph size: 157
##
## Initializing model
```

Working with MCMC lists

```r
mcmcapply(Normal.Bayes, sd)
```

```
##   beta[1]   beta[2]    prec.e
## 0.1965299 0.2170194 0.2323096
```

```r
mcmcapply(Normal.Bayes, quantile, c(0.05, 0.5, 0.95))
```

```
##        beta[1]   beta[2]    prec.e
## 5%  0.06935189 0.4255625 0.5602015
## 50% 0.39474914 0.7786494 0.8813594
## 95% 0.71091977 1.1308439 1.3214703
```

```r
quantile(Normal.Bayes, c(0.05, 0.5, 0.95))
```

```
##        beta[1]   beta[2]    prec.e
```

```
## 5%  0.06935189 0.4255625 0.5602015
## 50% 0.39474914 0.7786494 0.8813594
## 95% 0.71091977 1.1308439 1.3214703
```

```
str(as.matrix(Normal.Bayes))
```

```
##  num [1:15000, 1:3] 0.6698 0.4373 0.0782 0.4037 -0.1856 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:3] "beta[1]" "beta[2]" "prec.e"
```

We need to talk about RNGs

```
library(rjags)
```

```
## Warning: package 'rjags' was built under R version 4.4.1
```

```
## Linked to JAGS 4.3.1
```

```
## Loaded modules: basemod,bugs
```

```
str(parallel.seeds("base::BaseRNG", 5))
```

```
## List of 5
##  $ :List of 2
##   ..$ .RNG.name : chr "base::Super-Duper"
##   ..$ .RNG.state: int [1:2] -1993729999 -831051
##  $ :List of 2
##   ..$ .RNG.name : chr "base::Mersenne-Twister"
##   ..$ .RNG.state: int [1:625] 1 -1752577910 2094455660 -308494097 2105183013 -1210042916 -1354124366
##  $ :List of 2
##   ..$ .RNG.name : chr "base::Wichmann-Hill"
##   ..$ .RNG.state: int [1:3] 26706 9323 30096
##  $ :List of 2
##   ..$ .RNG.name : chr "base::Marsaglia-Multicarry"
##   ..$ .RNG.state: int [1:2] 654142287 964138968
##  $ :List of 2
##   ..$ .RNG.name : chr "base::Super-Duper"
##   ..$ .RNG.state: int [1:2] -994728961 -475764793
```

```
## The lecuyer module provides the RngStream factory, which allows large
## numbers of independent parallel RNGs to be generated.
load.module("lecuyer")
```

```
## module lecuyer loaded
```

```
list.factories(type="rng")
```

```
##              factory status
## 1 lecuyer::RngStream   TRUE
## 2      base::BaseRNG   TRUE
```

```
str(parallel.seeds("lecuyer::RngStream", 5))
```

```
## List of 5
##  $ :List of 2
##   ..$ .RNG.name : chr "lecuyer::RngStream"
##   ..$ .RNG.state: int [1:6] -352249749 2051297473 283856647 -181914435 -1111916701 -62035847
##  $ :List of 2
```

```
##   ..$ .RNG.name : chr "lecuyer::RngStream"
##   ..$ .RNG.state: int [1:6] 1744773278 1859880457 1114554539 -2268655 1899602234 -392228607
##  $ :List of 2
##   ..$ .RNG.name : chr "lecuyer::RngStream"
##   ..$ .RNG.state: int [1:6] -1216114281 -1896121500 1053032022 1211338842 2023107399 1028140010
##  $ :List of 2
##   ..$ .RNG.name : chr "lecuyer::RngStream"
##   ..$ .RNG.state: int [1:6] -958688469 1591196597 -484472749 1415237359 -1337881580 -995428655
##  $ :List of 2
##   ..$ .RNG.name : chr "lecuyer::RngStream"
##   ..$ .RNG.state: int [1:6] 1946038493 1769869172 1927613076 1868950640 -2120604533 347747616
```

Can we run chains in parallel?

```
system.time({
    Normal.Bayes = jags.fit(
        data=dat, params=c("beta","prec.e"), model=Normal.model,
        n.chains=4,
        n.update=10^6)
})
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 30
##    Unobserved stochastic nodes: 3
##    Total graph size: 157
##
## Initializing model
```

```
##    user  system elapsed
##  18.827   0.014  18.949
```

```
if (.Platform$OS.type != "windows") {
system.time({
    Normal.Bayes = jags.parfit(
        cl=10,
        data=dat, params=c("beta","prec.e"), model=Normal.model,
        n.chains=4,
        n.update=10^6)
})
}
```

```
##
## Parallel computation in progress
```

```
##    user  system elapsed
##  19.574   0.106   4.999
```

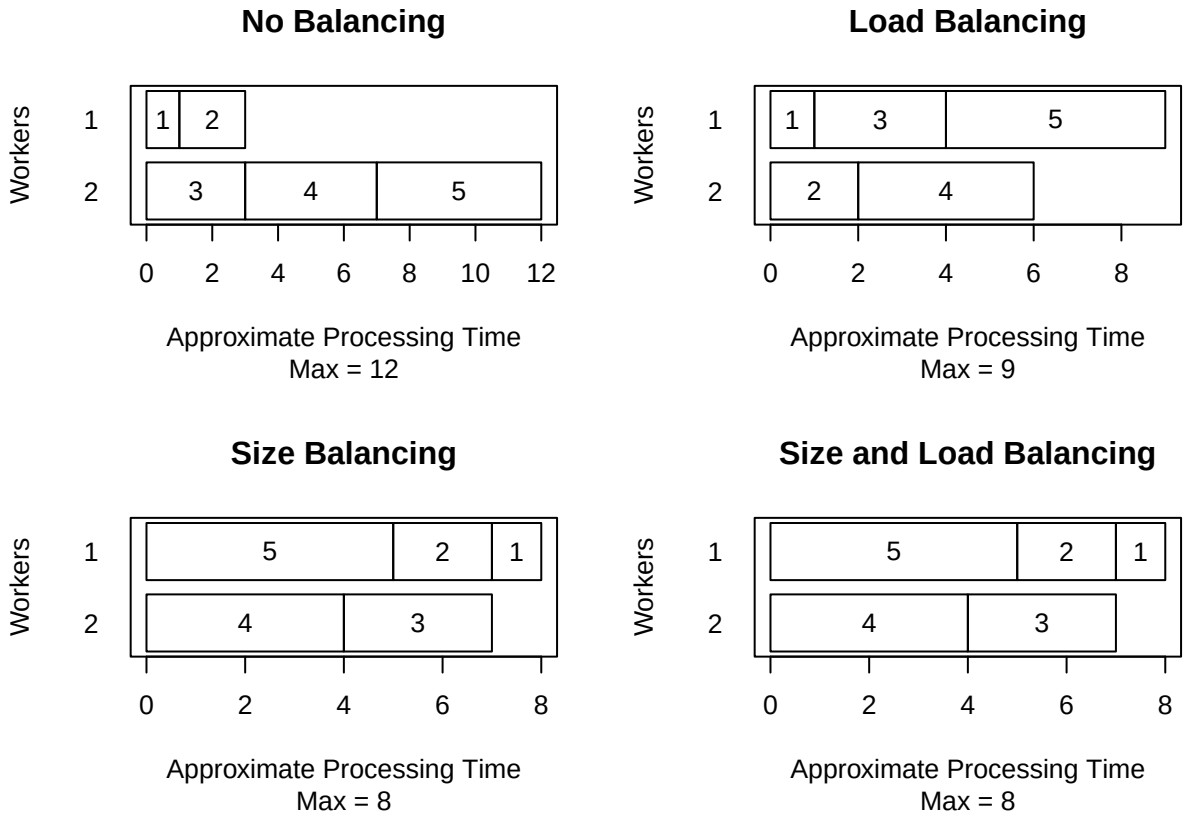
I like DC but I don't have time . . .

Well, check `dc.parfit`

```
## determine the number of workers needed
clusterSize(1:5)
```

```
##   workers none load size both
## 1       1   15   15   15   15
## 2       2   12    9    8    8
## 3       3    9    7    5    5
## 4       4    7    6    5    5
## 5       5    5    5    5    5
```

```
## visually compare balancing options
opar <- par(mfrow=c(2, 2))
plotClusterSize(2,1:5, "none")
plotClusterSize(2,1:5, "load")
plotClusterSize(2,1:5, "size")
plotClusterSize(2,1:5, "both")
```




```
par(opar)
```

Parallel chains, size balancing, both.

References

Lele, S.R., B. Dennis and F. Lutscher, 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods. Ecology Letters, 10:551-563.

Lele, S. R., and Dennis, B., 2009. Bayesian methods for hierarchical models: are ecologists making a Faustian bargain? Ecological Applications, 19:581-584.

Ponciano, J. M., Taper, M. L., Dennis, B., and Lele, S. R., 2009. Hierarchical models in ecology: confidence intervals, hypothesis testing, and model selection using data cloning. Ecology, 90:356-362.

Lele, S. R., Nadeem, K., and Schmuland, B., 2010. Estimability and likelihood inference for generalized linear mixed models using data cloning. Journal of the American Statistical Association, 105:1617-1625.

Lele, S. R., 2010. Model complexity and information in the data: Could it be a house built on sand? Ecology, 91(12):3493-3496.

Sólymos, P., 2010. dclone: Data Cloning in R. The R Journal, 2(2):29-37.

Lele, S. R., 2020. Consequences of lack of parameterization invariance of non-informative bayesian analysis for wildlife management: survival of San Joaquin Kit Fox and declines in amphibian populations. Front. Ecol. Evol., 7:501.

Lele, S. R., 2020. How should we quantify uncertainty in statistical inference? Front. Ecol. Evol., 8:35.