YABS Introduction

Vithor Rosa Franco

2023-02-02

The YABS Package

This package is intended to provide a flexible tool for Bayesian Modeling. It is similar to LaplacesDemon and fmcmc, but the MCMC algorithms are written in C++, speeding up the calculations in comparison to these packages. In fact, most of our source code for the MCMC algorithms is inspired on LaplacesDemon and we use its structure to set the models to be estimated (as demonstrated below).

We also aim at implementing other MCMC algorithms and estimation methods (such as Variational Bayes and Laplaces Approximation) in the future, once again echoing the original intent of the LaplacesDemon package. Another useful tools (such as the calculation of Bayes Factors and other fit and diagnostic statistics) will also be implemented as we have the time to implement them.

Finally, you should be aware that, currently, YABS computational efficiency does not equal JAGS' or Stan's. However, the MCMC algorithms work as they should and, therefore, the estimates are reliable, specially when dealing with less complex models. Currently, YABS generates samples only for real-valued parameters (i.e., parameters that can have positive and negative real values). But, as shown below, one can easily use some shortcuts to deal with positive-only parameters.

Basic example: Multiple Linear Regression

To illustrate the basic usage of YABS, we will work with a simulation regarding the estimation of multiple linear regression parameters with the Random-walk Metropolis MCMC algorithm. Therefore, to begin with, we need to load the `YABS` package. We will also load the `compiler` package, which allows to compile R functions (which can make them run faster):

```
### Packages and functions====
require(YABS)
```

```
## Carregando pacotes exigidos: YABS
```

```
## Carregando pacotes exigidos: coda
```

```
require(compiler)
```

```
## Carregando pacotes exigidos: compiler
```

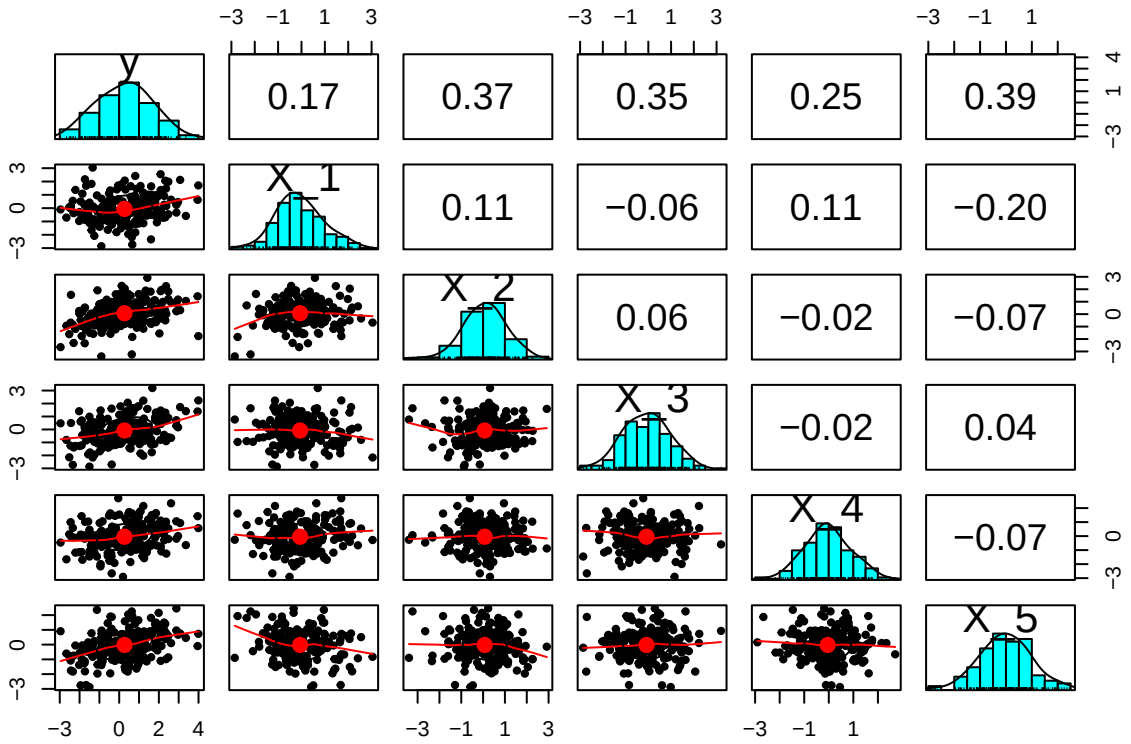
Data simulation

The next step is to simulate the regression data. For that, we are setting the seed to be equal to `1234`, the sample size is set to be equal to `200`, and the number of predictor variables is set to be equal to `5`. The predictor variables are independently sampled from normal distribution with mean equals to 0 and standard deviation equals to 1. The regression coefficients are sampled from a uniform distribution, with lowerbound equals to .30 and upperbound equals to .70. Then, the criterion variable is drawn from a normal distribution with the conditional mean equals to the linear combination of the weighted predictors and an intercept, and with the residual standard deviation equals to 1:

```r
### Random data====
seed <- 1234; N <- 200; V <- 5
set.seed(seed)
X <- sapply(1:V, function(g) rnorm(N, mean=0, sd=1))
colnames(X) <- paste0("X_",1:V)
betas <- runif(V+1, .3, .7)
y <- rnorm(N, cbind(1,X) %*% betas, 1)
```

We can graphically assess the relation between the predictor variables, and between the predictor variables and the criterion variable, using the `pairs.panels` function from the `psych` package:

```r
psych::pairs.panels(cbind(y,X))
```



Data List

In the `YABS` pipeline (which is mostly based on the pipeline of the LaplacesDemon package), the next step is preparing a data list to be sent to the `MCMC` function. The first thing you should do is to name the parameters and to identify their positions in the list of names. Of course, for doing that, you should already have a good understanding of what parameters will be included in your model. Because we are dealing with multiple linear regression, we know well what the parameters are: an intercept; a regression coefficient for each predictor; and the variance/standard deviation of the residuals. Traditionally, intercepts and regression coefficients are named as `beta`, and the standard deviation of the residuals is named as `sigma`. The `beta` parameters receive a number that indicates to which variable is that parameter related, with the intercept usually receiving the `beta0` parameter. For doing that, we first create a vector, where the `beta` parameters are defined first, and then we include the `sigma` parameter. To get the position of the parameters we use the `grep` function. We will use this function to allocate the `beta` parameters to an object, and then the `sigma` parameter to a second object.

```
### DATA LIST====
parm.names <- c(paste0("beta", 0:V), "sigma")
pos.beta   <- grep(pattern="beta", x=parm.names)
pos.sigma  <- grep(pattern="sigma", x=parm.names)
PGF <- function(Data) {
  beta  <- rnorm(n=ncol(Data$X)+1, mean=0, sd=1)
  sigma <- qnorm(pgamma(rgamma(n=1, shape=1e-2, rate=1e-2), shape=1e-2, rate=1e-2))
  return(c(beta, sigma))
}
Data <- list( X=X, y=y, n=N, parm.names=parm.names, pos.beta=pos.beta,
              pos.sigma=pos.sigma, PGF=PGF )
Initial.Values <- PGF(Data)
```

After specifying the names and positions of the parameters, we need to create a Parameter-Generating Function (PGF). The PGF can be used to generate initial values based on the prior distribution of the parameters. Because the regression coefficients `beta` can receive any value between -Inf and +Inf, their are set to be drawn from a normal distribution with mean equals to 0 and standard deviation equals to 1. The residual deviation `sigma`, however, can receive only positive values. Therefore, what we do is: first, sample from a gamma distribution, with shape and rate parameters equal to 1e-2. Then, we use the cumulative distribution function (CDF) of this gamma distribution to convert the sampled value from the 0 to +Inf scale to the 0 to 1 scale. Finally, this value is converted proportionally to the range -Inf to +Inf using the quantile function of the normal distribution. These steps are necessary as, currently, the MCMC algorithms implemented in `YABS` can only sample from unrestricted parameters. However, in the model specification, we will show you how to guarantee that the `sigma` parameter does not receives an inappropriate negative value.

We then create the data list that will be used by the model and by the chosen MCMC algorithm. In this list, we should include all variables, data frames, matrices, and any objects that will be used within the model. It is also mandatory that there is `parm.names` element included in the list, with the names of the parameters, as well as the `PGF` element. It is not necessary to generate/provide initial values by yourself, but we include this procedure in the code for the completeness of the example.

Model Specification

The second step in the `YABS` pipeline is to specify the model. Notice that this is done by defining a function, such is done when estimating parameters with the `optim` function (with some minor differences, specified below). There is also just some minor mandatory steps that you need to include in the model's output; the other steps in the specification of the model can be done as you will. But we do believe that the example below helps to make it clear what is being achieved at each step (an important feature when you need to debug your own models). Then, first, you need crate objects of the parameters of the model. This will help with making the code less cumbersome in the next steps. For that, you will use the positions created earlier: `pos.beta` for the `beta` parameters; and `pos.sigma` for the `sigma` parameters. In regards to the `sigma` parameter, because it necessarily has to receive a positive value, we use the `exp` function in the estimated value to guarantee that the operations will all be done in a positive value.

The second step in model specification is to calculate the priors of the parameters. As stated before when creating the `PGF`, the `beta` parameters are assumed to come from a normal distribution with mean equal to 0 and standard deviation equal to 1 (that is why we use the `dnorm` function). The `sigma` parameter is assumed to come from a gamma distribution with both shape and rate parameters equal to 1e-2 (that is why we use the `dgamma` function). It is necessary that you take the log of the probabilities, as this allows us to simply sum the priors and the likelihoods to calculate the (log-)posterior of the model. Finally, to simplify the code later, we create an object, `Lp`, which is the value of the log-prior of the model.

```
### MODEL====
Model <- function(parm, Data){
  ### Parameters
  beta  <- parm[Data$pos.beta]
```

```
  sigma <- exp(parm[Data$pos.sigma])

  ### Log-Prior
  beta.prior  <- sum(dnorm(x=beta, mean=0, sd=1, log=TRUE))
  sigma.prior <- sum(dgamma(x=sigma, shape=1e-2, rate=1e-2, log=TRUE))
  Lp <- beta.prior + sigma.prior

  ### Log-Likelihood
  mu <- cbind(1,Data$X) %*% beta
  LL <- sum(dnorm(x=Data$y, mean=mu, sd=sigma, log=TRUE))

  ### Log-Posterior
  LP <- LL + Lp
  Modelout <- list(LP=LP, Dev=-2*LL, Monitor=c(LL, Lp), yhat=rnorm(n=Data$n, mean=mu, sd=sigma), parm=pa
  return(Modelout)
}
Model <- compiler::cmpfun(Model)
```

After calculating the log-prior, we need to calculate the log-likelihood. For doing this, in the context of multiple linear regression, first, we need to calculate what are the values of `y` predicted by the combination of the current estimates of the `beta` parameters, and, second, to use this predictions and the `sigma` parameter to calculate the log-likelihood. The predictions made by a linear model is simple: we just need to multiple the matrix of the intercepts and of the predictors (i.e., `cbind(1,Data$X)`) by the vector of beta parameters. In the maximum-likelihood specification of a multiple linear regression, the likelihood is calculated by setting the mean to be equal to the predictions made by the model, and the `sigma` parameter is the standard deviation of the conditional normal distribution of the `y` variable. Again, we should take the log and sum all the values to get the log-likelihood of the model.

In the last step of the model specification, we need to calculate the log-posterior and tell the model what has to be returned as a result. First, calculating the log-posterior is trivial at this point: one simply has to add the log-prior to the log-likelihood. But one should be very careful when specifying what is returned by the model, as the outputs are all mandatory. `LP` is simply the log-posterior calculated by the model. `Dev` is the deviance, which is defined as minus two times the log-likelihood. `Monitor` is a numeric vector including intermediary values that are calculated within the model, including variables, parameters, and whatnot. In the example, we included the log-likelihood and the log-prior. `yhat` is a vector of predictions made by the model for the current values of the parameters. This is used later for specifying the posterior predictive distribution. `parm` is simply the current values of the estimated parameters. The final line is simply compiling the code of the model.

MCMC Parameters and Run

In the last step of YABS pipeline one will set the parameters of the MCMC algorithm. YABS was developed so the algorithms are as "plug-and-play" as possible. But some decisions are still up to the user. The first one is the number of samples that will be drawn with the MCMC algorithm. This is specified by the `iterations` argument. Next, the user has to specify the number of initial samples that will be discarded. This is traditionally known as the `burnin` stage. The user can choose at what frequency updates regarding sampling will be printed in the console. this is done using the `status` argument. `thinning` indicates that every nth iteration will be retained, while the other iterations are discarded. `adapt` is the number of initial adaptive samples and, if adaptation is required, it should be at least `500`. `nchains` is the number of sampling schemes (i.e., chains). `parallel` is a boolean argument indicating if the chains will be processed in parallel. `cores` is used only if `parallel = TRUE`; be careful to not set this number higher than the number of cores you really have available! `update.progress` indicates how many times the progress bar will be update in parallel processing. `opt.init` is a boolean indicating if initial values and initial step size should be optimized before the MCMC algorithm starts sampling. We strongly suggest that this is set to `TRUE`; it will make the

MCMC run a bit slower, but most likely more efficiently. If convergence is achieved at this point, the adaptive steps will not run. `par.cov` is an optional covariance matrix of the parameters (i.e., the diagonal represents the step sizes for the parameters to be estimated using the MCMC algorithms). The final argument, `algo`, indicates what MCMC algorithm will be used. We strongly suggest the readers to check the documentation to learn about additional details.

```
### FIT====
iterations <- 2500
burnin <- 500
status <- 100
thinning <- 1
adapt <- 500
nchains <- 2
parallel <- TRUE
cores <- nchains
update.progress <- 2
opt.init <- TRUE
par.cov <- NULL
algo <- "rwm"
fit <- MCMC(Model, Data, Initial.Values=Initial.Values,
            iterations=iterations, burnin=burnin, status=status,
            thinning=thinning, adapt=adapt, nchains=nchains,
            parallel=parallel, cores=cores, update.progress=update.progress,
            opt.init=opt.init, par.cov=par.cov, algo=algo)
```

```
## Improving initial values and finding initial step sizes with MAP estimation.
## Initial optimization was sufficient for estimating the step sizes.
## Algorithm: Random-walk Metropolis
##
## Running 2 chains in parallel
##
## It took 2.03 secs for the run to finish.
```

And that is it! You can check the results using the `print` or `summary` methods developed for `YABS` objects, like the one that is returned by the `MCMC` function. They are similar to the methods used for JAGS and Stan objects, so users can more easily understand (and sometimes compare) the outputs.

```
### SUMMARY====
fit        # Print method
```

```
## YABS output generated with RWM algorithm.
## MCMC ran for 0.000 minutes.
##
##                 EAP     sd     2.5%    97.5%  overlap0      f     ESS   PSRF
## beta0         0.318  0.076    0.165    0.479     FALSE  1.000  193.18  1.011
## beta1         0.293  0.067    0.164    0.431     FALSE  1.000  278.34  1.019
## beta2         0.502  0.069    0.358    0.649     FALSE  1.000  270.82  1.017
## beta3         0.445  0.063    0.318    0.559     FALSE  1.000  320.08  1.002
## beta4         0.385  0.075    0.234    0.539     FALSE  1.000  252.10  1.007
## beta5         0.649  0.067    0.518    0.780     FALSE  1.000  334.23  1.000
## sigma        -0.042  0.054   -0.141    0.071      TRUE  0.785  221.76  1.005
## deviance    550.454  4.257  544.772  560.797     FALSE  1.000  188.63  1.042
## aic         564.454  4.257  558.772  574.797     FALSE  1.000  188.63  1.042
##
## Successful convergence based on PSRF (or Rhat) values (all < 1.1).
```

```
print(fit) # Explicit print
```

```
## YABS output generated with RWM algorithm.
## MCMC ran for 0.000 minutes.
##
##                EAP     sd     2.5%    97.5%  overlap0      f     ESS   PSRF
## beta0        0.318  0.076    0.165    0.479     FALSE  1.000  193.18  1.011
## beta1        0.293  0.067    0.164    0.431     FALSE  1.000  278.34  1.019
## beta2        0.502  0.069    0.358    0.649     FALSE  1.000  270.82  1.017
## beta3        0.445  0.063    0.318    0.559     FALSE  1.000  320.08  1.002
## beta4        0.385  0.075    0.234    0.539     FALSE  1.000  252.10  1.007
## beta5        0.649  0.067    0.518    0.780     FALSE  1.000  334.23  1.000
## sigma       -0.042  0.054   -0.141    0.071      TRUE  0.785  221.76  1.005
## deviance   550.454  4.257  544.772  560.797     FALSE  1.000  188.63  1.042
## aic        564.454  4.257  558.772  574.797     FALSE  1.000  188.63  1.042
##
## Successful convergence based on PSRF (or Rhat) values (all < 1.1).
```

```
summary(fit) # Summary method
```

```
## YABS output generated with RWM algorithm.
## Estimates based on 2 chains of 3000 iterations,
## burn-in = 500 iterations, adaptation = 0 iterations, and thin rate = 1,
## yielding 6000 total samples from the joint posterior.
## MCMC ran for 0.000 minutes.
##
##                EAP     sd     2.5%    97.5%  overlap0      f     ESS   PSRF
## beta0        0.318  0.076    0.165    0.479     FALSE  1.000  193.18  1.011
## beta1        0.293  0.067    0.164    0.431     FALSE  1.000  278.34  1.019
## beta2        0.502  0.069    0.358    0.649     FALSE  1.000  270.82  1.017
## beta3        0.445  0.063    0.318    0.559     FALSE  1.000  320.08  1.002
## beta4        0.385  0.075    0.234    0.539     FALSE  1.000  252.10  1.007
## beta5        0.649  0.067    0.518    0.780     FALSE  1.000  334.23  1.000
## sigma       -0.042  0.054   -0.141    0.071      TRUE  0.785  221.76  1.005
## deviance   550.454  4.257  544.772  560.797     FALSE  1.000  188.63  1.042
## aic        564.454  4.257  558.772  574.797     FALSE  1.000  188.63  1.042
##
## Successful convergence based on PSRF (or Rhat) values (all < 1.1).
## PSRF is the potential scale reduction factor (at convergence, PSRF=1).
## ESS is the sample size of each posterior adjusted for autocorrelation.
##
## overlap0 checks if 0 falls in the parameter's 95% credible interval.
## f is the proportion of the posterior with the same sign as the mean;
## i.e., our confidence that the parameter is positive or negative.
##
## DIC info: (pD = var(deviance)/2).
## pD = 9.06 and DIC = 559.52
## DIC is an estimate of expected predictive error (lower is better).
## MPSRF = 1.042
## MPSRF is the multivariate potential scale reduction factor (at convergence, MPSRF=1).
```