Constructing Bayesian Models with Markov Chain Monte Carlo Techniques for Nuclear Astrophysics

Kevin Anderson
Supervised by Christian Iliadis

February 15, 2016

Todo list

Contents


```
## Last modified on
## Mon Feb 15 01:31:38 2016
## rjags info:

## Loading required package:  coda
## Linked to JAGS 4.0.0
## Loaded modules:  basemod,bugs
```

1 Bayesian modeling

explain Baye's theorem, priors, posteriors, MCMC, etc

1.1 MCMC

2 rjags

2.1 A basic rjags model

caching?

analyzing how well the MCMC mixed?

The overall idea with using rjags is to define a model structure, give this, our datasets, and prior distributions to the MCMC engine, and get back posterior distributions of specified parameters.

separate files for sections

First, the package needs to be loaded, and we may as well set a seed for the random number generator so that our results are reproducible:

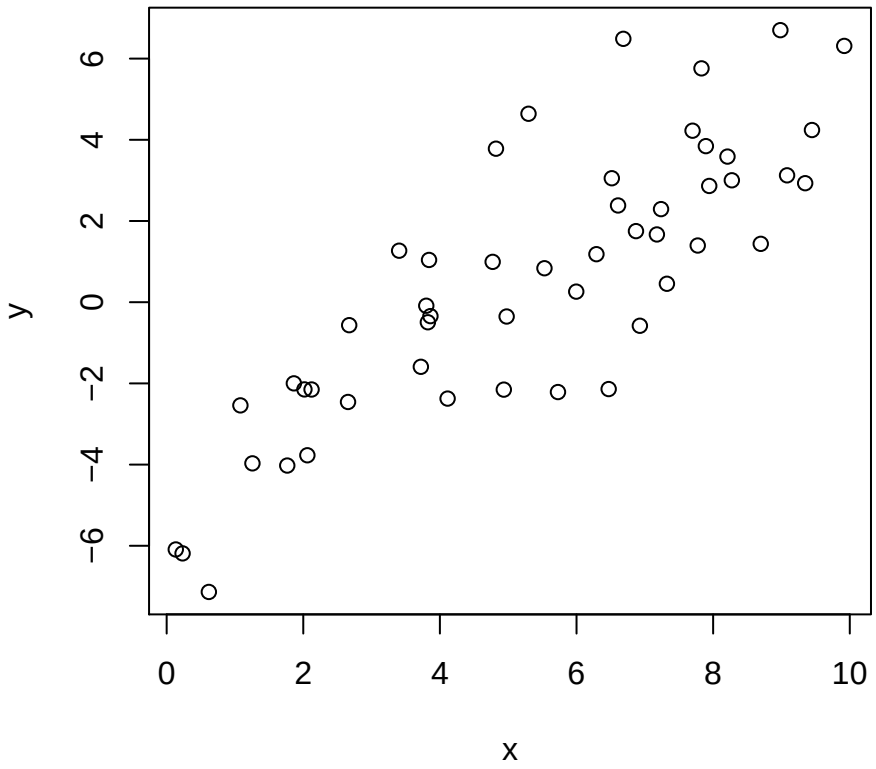
```
library(rjags)
set.seed(1)
```

Now let's make a basic model to test rjags. The simplest realistic example is a linear relationship with some y-scatter. Let's make an artificial dataset for testing:

```
# Create an artificial dataset:
true_slope = 1
true_intercept = -5
true_sigma = 2
N = 50

x = runif(N,0,10)
y = rnorm(N,true_slope * x + true_intercept, true_sigma)
```

At least for now, we should take a look at our data before analyzing it, just to make sure it is how we expect it to be:

```
plot(x,y)
```



Now let's talk about the model we will use. We are modeling the data as linear with gaussian scatter:

$$y_i \sim \mathcal{N}(ax_i + b, \sigma^2)$$

To actually implement this model, we will need to express it in the JAGS langauge. The JAGS language is a little different from normal programming languages like R. It is a declarative language, meaning that the code you write is not followed as a sequence of steps, but rather as a description of logic. For instance, the line `y.hat[i] = a * x[i] + b` does not evaluate the right side and set the left side equal to it, as R would. Instead, this code only specifies the model structure that JAGS should use.

For now, the model code will be separated into two sections. The first is for specifying priors on our parameters of interest. Here, we are interested in the slope `a`, intercept `b`, and standard deviation `sigma` of our dataset. The second section is for describing how our different variables are related to each other. Then, the whole chunk of JAGS code is wrapped in `model{...}` and passed into the JAGS engine.

For this dataset, the priors might be as follows. The slope `a` we expect to be non-negative and no greater than 5, so we could use a uniform density between 0 and 5. The intercept `b` could be anywhere between ± 10, and the scatter's standard deviation `sigma` is expected to be between 0 and 3. Note that the choice of priors is domain specific and not the emphasis here, but we have chosen priors that include the "true" values.

Let's write the JAGS code for this model and capture it into a string:

```
model_string = "
  model {
    # priors:
    a ~ dunif(0,5)
    b ~ dunif(-10,10)
    sigma ~ dunif(0,3)

    # structure:
    for (i in 1:N) {
        y[i] ~ dnorm(a * x[i] + b, pow(sigma, -2))
    }
  }
"
```

Density functions in JAGS always start with "d": `dunif` for uniform density, `dnorm` for gaussian density, `dlnorm` for lognormal density, etc. The tildes are read as "is distributed as". For instance, `a ~ dunif(0,5)` is read as "the variable `a` is distributed uniformly between 0 and 5".

In the structure block, we have a traditional for loop. This allows us to relate each `y` value to each `x` value such that each `y` value is normally distributed around the line $a*x+b$ with standard deviation `sigma`. For historical reasons, JAGS distribution functions use a "precision" that is defined as one over the variance. Usually it is more convenient to speak in terms of standard deviation and variance, so most models will have a small conversion between standard deviation and precision included.

Note that the order of lines is somewhat unimportant – the priors could have been included after the structure and it would make no difference.

Now that we have a model structure and some data, we can give this to JAGS and see what comes back. Normally the JAGS engine wants the model structure in a separate file, but to keep it easy we'll use a `textConnection` instead. This allows us to create a fake file whose contents are the model string we created above, and we'll give this to JAGS instead.

Now, we want to give our model structure and data to JAGS and get back the output of the MCMC process to analyze.

```
model = jags.model(file = textConnection(model_string),
                   data = list('x' = x,
                               'y' = y,
                               'N' = N),
                   n.chains = 1,
                   n.adapt = 1000,
                   inits = list('.RNG.name' = 'base::Mersenne-Twister',
                                '.RNG.seed' = 1))

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 3
##    Total graph size: 313
##
## Initializing model
```

This has created a model, and printed some information that is not particularly interesting. We can then actually generate a Markov chain with

```
output = coda.samples(model = model,
                      variable.names = c("a", "b", "sigma"),
                      n.iter=1000,
                      thin=1)
```

We get back an object `output` that contains the Markov chain. First, as always in R, have a look at the summary of it:

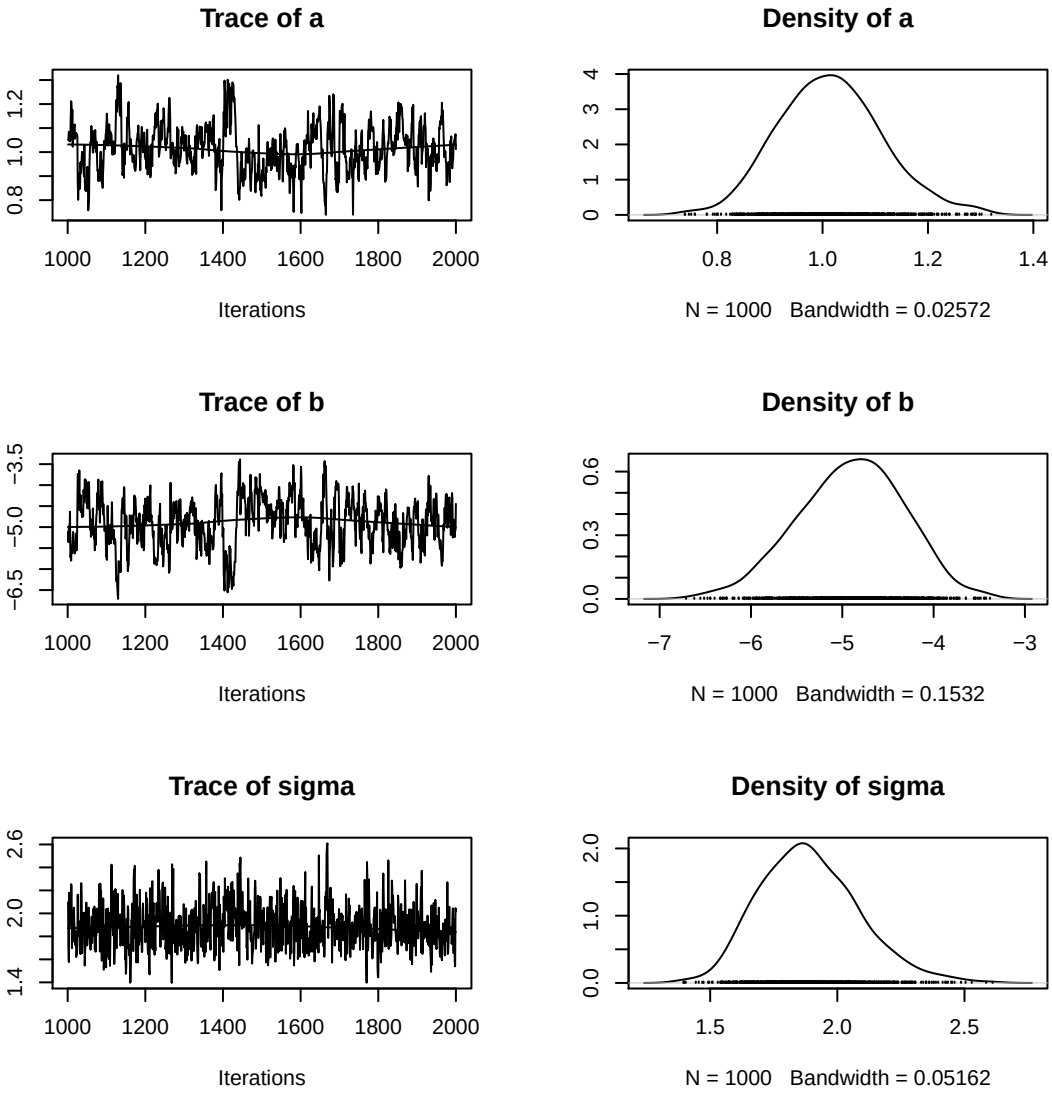
Talk about adapt, chains, thin, etc

```
print(summary(output))

##
## Iterations = 1001:2000
## Thinning interval = 1
## Number of chains = 1
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##          Mean      SD Naive SE Time-series SE
## a       1.015 0.09858 0.003117       0.010918
## b      -4.905 0.57907 0.018312       0.063502
## sigma   1.891 0.19389 0.006131       0.007875
##
## 2. Quantiles for each variable:
##
```

```
##           2.5%     25%    50%    75%  97.5%
## a       0.8372  0.9484  1.013  1.078  1.220
## b      -6.0694 -5.2835 -4.876 -4.513 -3.857
## sigma   1.5659  1.7509  1.876  2.016  2.306
```

This gives us information about the distributions of the parameters we asked JAGS to calculate (`a`, `b`, and `sigma`). Notice that the means for each parameter are quite close to the true values. We can also see this visually:

```
plot(output)
```



We see mixing diagrams on the left, and posterior distributions on the right. The mixing diagrams appear qualitatively acceptable, and the posterior distributions are centered on or quite near the true values of 1, -5, and 2, respectively. The model has successfully recovered the values we used to create the data. The total, self-contained code for this is as follows:

```
library(rjags)
set.seed(1)

true_slope = 1
true_intercept = -5
true_sigma = 2
N = 50
x = runif(N,0,10)
y = rnorm(N,true_slope * x + true_intercept, true_sigma)
plot(x,y)

model_string = "
model {
    # priors:
    a ~ dunif(0,5)
    b ~ dunif(-10,10)
    sigma ~ dunif(0,3)

    # structure:
    for (i in 1:N) {
    y[i] ~ dnorm(a * x[i] + b, pow(sigma, -2))
    }
}
"

model = jags.model(file = textConnection(model_string),
                   data = list('x' = x,
                   'y' = y,
                   'N' = N),
                   n.chains = 1,
                   n.adapt = 1000,
                   inits = list('.RNG.name' = 'base::Mersenne-Twister',
                                '.RNG.seed' = 1))

output = coda.samples(model = model,
                      variable.names = c("a", "b", "sigma"),
                      n.iter=1000,
                      thin=1)

print(summary(output))

plot(output)
```

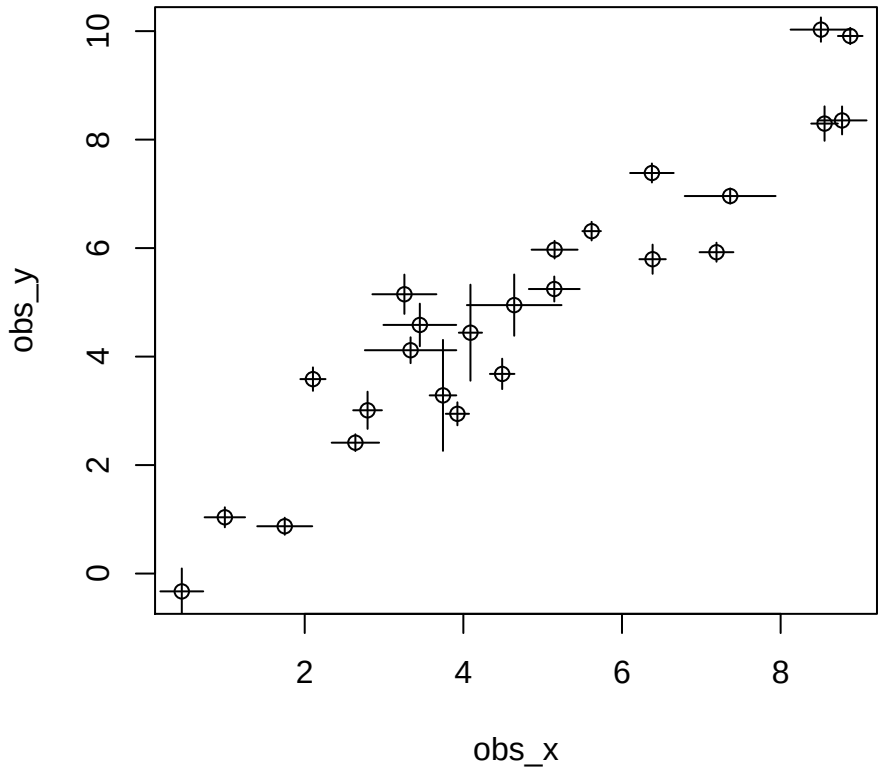
The rest of the examples in this paper will follow more or less the same format: create or load in some data, define the model, pass it to JAGS, and have a look at the output.

2.2 A more complex model

The basic model we assumed last time is rarely the most applicable to real-life data analysis. In general, there will be some measurement error (that is not necessarily constant across measurements) associated with the dataset in addition to scatter from other variables. Let's generate some data that reflects this:

```
N = 25
true_x = runif(N,0,10)
true_slope = 1
true_intercept = 0
scatter = 1
true_y = rnorm(N, true_slope * true_x + true_intercept, scatter)
# known measurement uncertainties
x_sigma = rlnorm(N, -2, 0.5)
y_sigma = rlnorm(N, -2, 0.5)
obs_x = rnorm(N, true_x, x_sigma)
obs_y = rnorm(N, true_y, y_sigma)
```

```
plot(obs_x, obs_y)
segments(obs_x, obs_y - 2*y_sigma, obs_x, obs_y + 2*y_sigma)
segments(obs_x - 2*x_sigma, obs_y, obs_x + 2*x_sigma, obs_y)
```



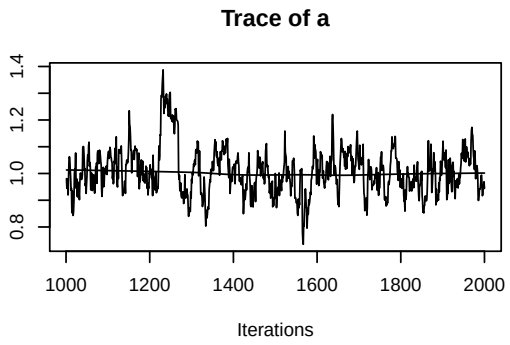
Clearly, the group scatter is larger than the individual measurement error bars allow, implying that one or more unmeasured variables are influencing the y values. The model we'll use is a linear relationship between y and x, with (uniform) measurement error on both y and x and additional scatter in y. The JAGS code is as follows:

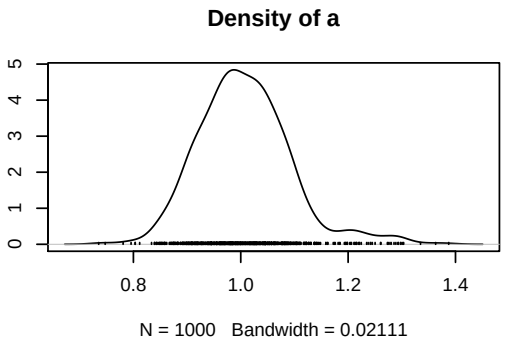
```
model_string = "
  model {
    # priors:
    a ~ dunif(0,5)
    b ~ dunif(-10,10)
    sigma ~ dunif(0,3)

    # structure:
    for (i in 1:N) {
        # the true x:
        x[i] ~ dunif(0,10)
        # the observed x:
        obs_x[i] ~ dnorm(x[i], pow(x_sigma[i],-2))

        # y, as it would be if it only depended on the true x:
        y[i] = a*x[i] + b
        # y, with the effect of the unmeasured confounding variable
        y_scatter[i] ~ dnorm(y[i], pow(sigma,-2))
        # y, with the confounding variable, with observational error:
        obs_y[i] ~ dnorm(y_scatter[i], pow(y_sigma[i],-2))
    }
  }
"
```

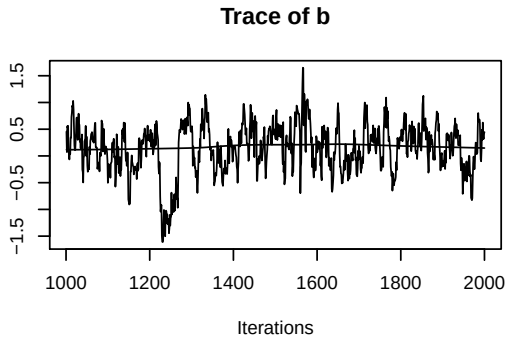
Now we proceed as before: feed the data and model structure into JAGS and look at the output:

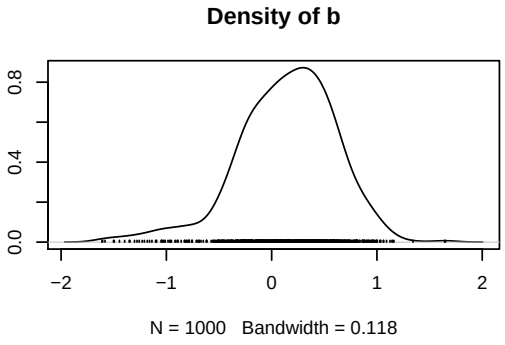
```
model = jags.model(file = textConnection(model_string),
                   data = list('obs_x' = obs_x,
                               'x_sigma' = x_sigma,
                               'obs_y' = obs_y,
                               'y_sigma' = y_sigma,
                               'N' = N),
                   n.chains = 1,
                   n.adapt = 1000,
                   inits = list('.RNG.name' = 'base::Mersenne-Twister',
                                '.RNG.seed' = 1))

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 53
##    Total graph size: 538
##
## Initializing model

output = coda.samples(model = model,
                      variable.names = c("a", "b", "sigma"),
                      n.iter=1000,
                      thin=1)
summary(output)
```

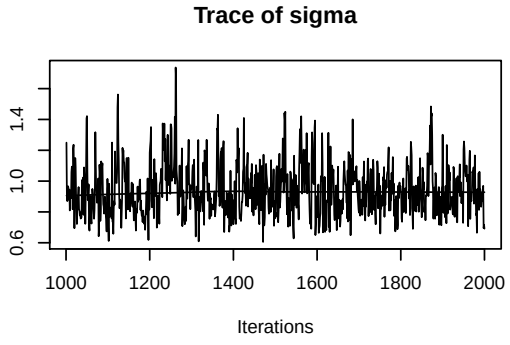
```
##
## Iterations = 1001:2000
## Thinning interval = 1
## Number of chains = 1
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##         Mean      SD Naive SE Time-series SE
## a     1.0070 0.08701 0.002751       0.011974
## b     0.1414 0.46736 0.014779       0.059455
## sigma 0.9430 0.16739 0.005293       0.009315
##
## 2. Quantiles for each variable:
##
##          2.5%     25%    50%    75% 97.5%
## a      0.8575  0.9493 1.0000 1.0556 1.221
## b     -1.0229 -0.1346 0.1749 0.4592 0.952
## sigma  0.6842  0.8164 0.9187 1.0424 1.343

plot(output)
```

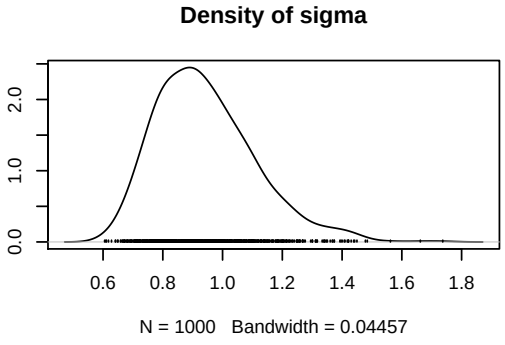

Trace of a
0.8
1.0
1.2
1.4
1000
1200
1400
1600
1800
2000
Iterations


Density of a
0
1
2
3
4
5
0.8
1.0
1.2
1.4
N = 1000 Bandwidth = 0.02111


Trace of b
−1.5
−0.5
0.5
1.5
1000
1200
1400
1600
1800
2000
Iterations


Density of b
0.0
0.4
0.8
−2
−1
0
1
2
N = 1000 Bandwidth = 0.118


Trace of sigma
0.6
1.0
1.4
1000
1200
1400
1600
1800
2000
Iterations


Density of sigma
0.0
1.0
2.0
0.6
0.8
1.0
1.2
1.4
1.6
1.8
N = 1000 Bandwidth = 0.04457