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

Kevin Anderson
Supervised by Christian Iliadis

March 21, 2016

## Todo list

use the glm package . . . 1
caching? . . . 2
analyzing how well the MCMC mixed? . . . 2
example of Bayes' theorem . . . 2
why is the denominator problematic? . . . 2
Talk about adapt, chains, thin, etc . . . 5
how to fix additive systematic errors? . . . 18

## Contents

**1 Preface** **2**

**2 Background** **2**
2.1 Bayes' Theorem . . . 2
2.2 MCMC . . . 2

**3 Bayesian Regression** **2**
3.1 Basic Linear Model . . . 2
3.2 Linear Model with Error Bars . . . 7
3.3 Linear Model with Multiple Datasets and Systematic Error . . . 12

**4 Choosing a Prior** **18**
4.1 Uninformative Priors . . . 19
4.2 Informative Priors . . . 21

```
## Last modified on
## Mon Mar 21 21:23:24 2016
## rjags info:

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

use the glm package

# 1 Preface

This paper is an introduction to constructing Bayesian regression models using the R programming language and the R package rjags. Familiarity with the R language and basic statistical background (means, standard deviations, etc) are necessary to understand the material.

The intent of this paper is to show how easy it is to fit complex regression models to experimental data with rjags. The techniques applied in this paper are surprisingly powerful, given their simplicity.

# 2 Background

caching?

analyzing how well the MCMC mixed?

## 2.1 Bayes' Theorem

First, some notation: for an event $A$, the probability of the event occurring is given by $P(A)$. For instance, the probabilities of an unbiased coin flip can be written as $P(\text{heads}) = 0.5$ and $P(\text{tails}) = 0.5$.

In many cases, we wish to know the conditional probability of an event. $P(A|B)$ represents the probability of event $A$ occurring, given that event $B$ occurs. To give an example of the difference between this and standard probability, if today is Wednesday (event $A$), the probability of tomorrow being Thursday (event $B$) is $P(B|A) = 1$, while without the condition, the probability is random: $P(B) = 1/7$.

The whole subject of Bayesian Inference rests on a mathematical rule called Bayes' theorem:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

Here, $H$ stands for a hypothesis to be tested and $E$ stands for the evidence (data). The right side contains three terms. $P(E|H)$ is the likelihood of measuring the evidence, assuming that the hypothesis is true. The two probabilities $P(H)$ and $P(E)$ are called priors or prior distributions and represent relevant knowledge beyond the dataset itself. They are called "priors" because we know these probabilities before doing the analysis.

The left side of Bayes' theorem reads "The probability of $H$ given the evidence", which represents exactly what we are interested in – how likely is this hypothesis, according to the data that we collected? This probability is known as the posterior or the posterior distribution. In this paper, we'll focus on fitting model parameters to datasets, and so what we are interested in are real numbers: slopes, intercepts, standard deviations, etc. These of course lie on the real axis and are continuous, and so the posterior for some model will be probability distributions for each parameter.

example of Bayes' theorem

why is the denominator problematic?

## 2.2 MCMC

Monte Carlo methods are algorithms that use randomly generated numbers to estimate numbers that are too hard to calculate analytically. A common use case for Monte Carlo techniques is evaluating complex multidimensional integrals.

# 3 Bayesian Regression

## 3.1 Basic Linear Model

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.

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:

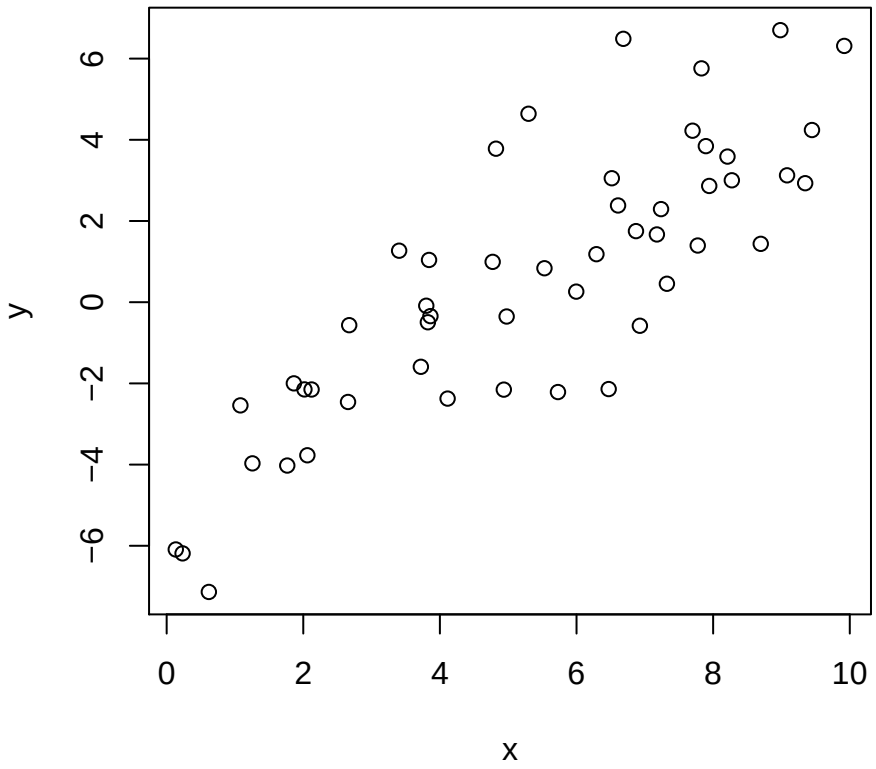
```
library(rjags)
set.seed(1)
```

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 $\pm 10$, 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:

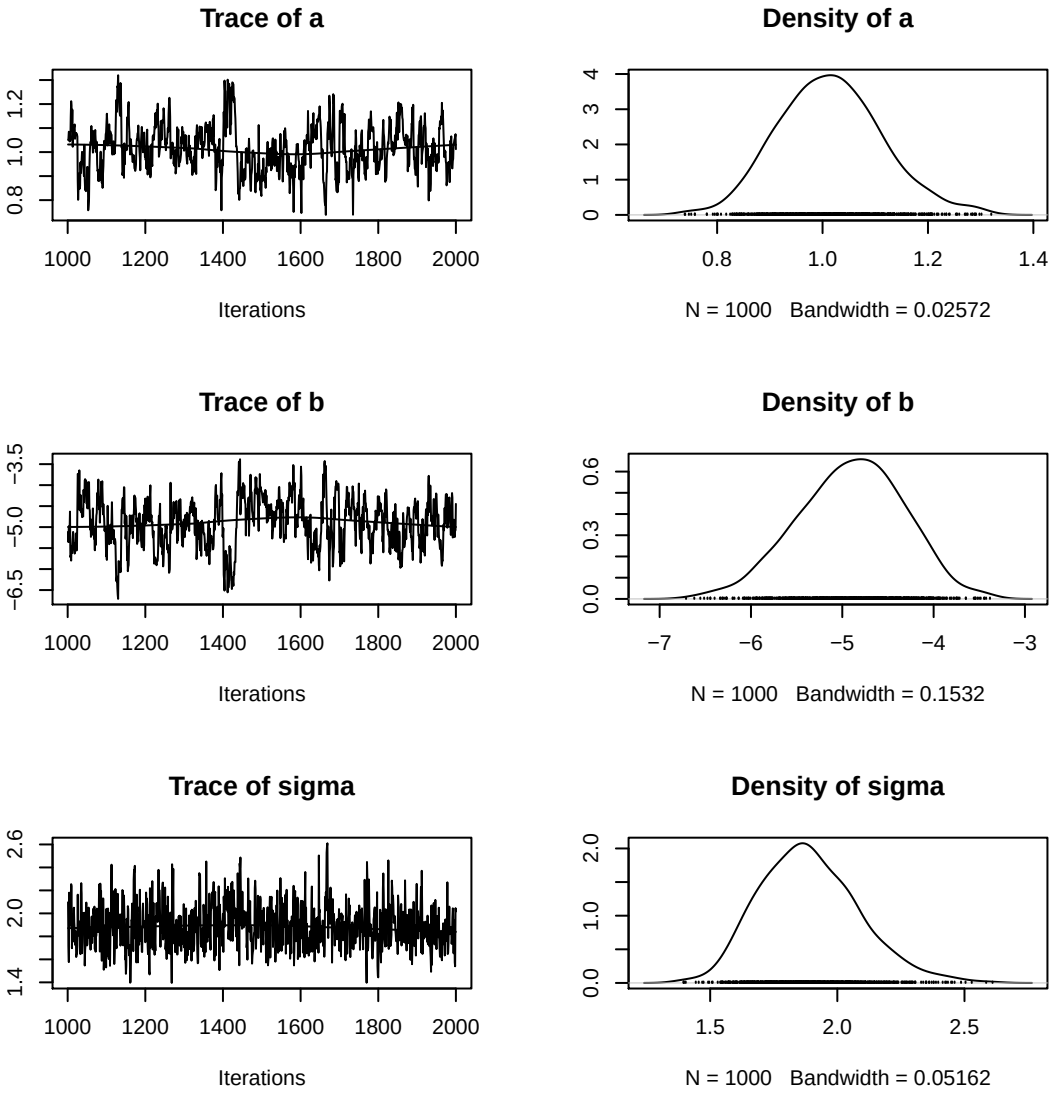
Talk about adapt, chains, thin, etc

```
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)
```

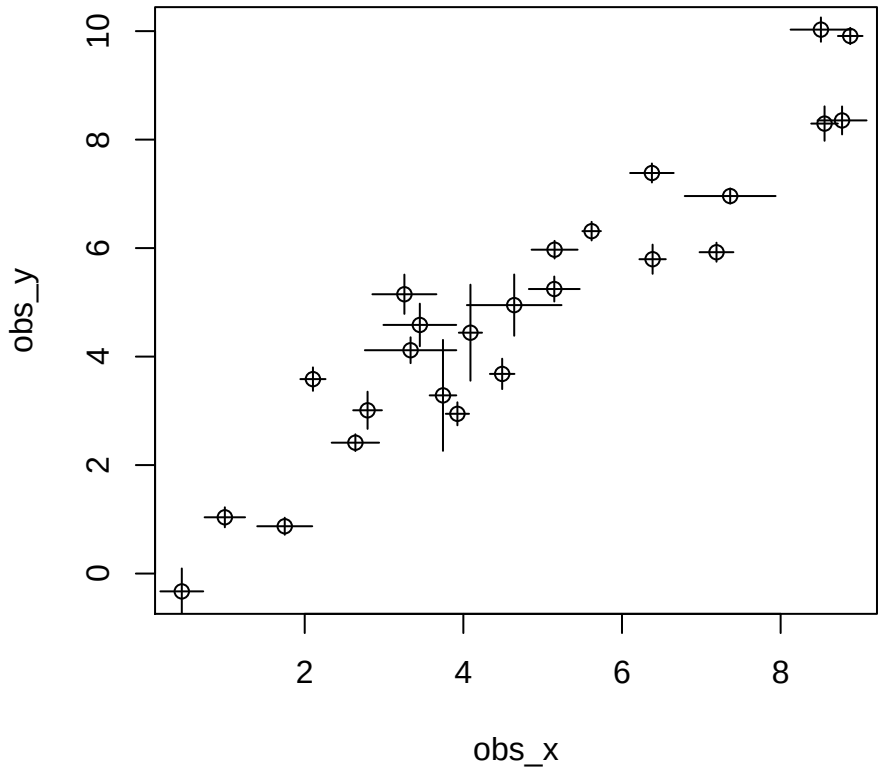
The rest of the examples in this paper will follow more or less the same format: create or load in some data, define the model, pass it to JAGS, and have a look at the output.

## 3.2 Linear Model with Error Bars

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 reflect 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)
```



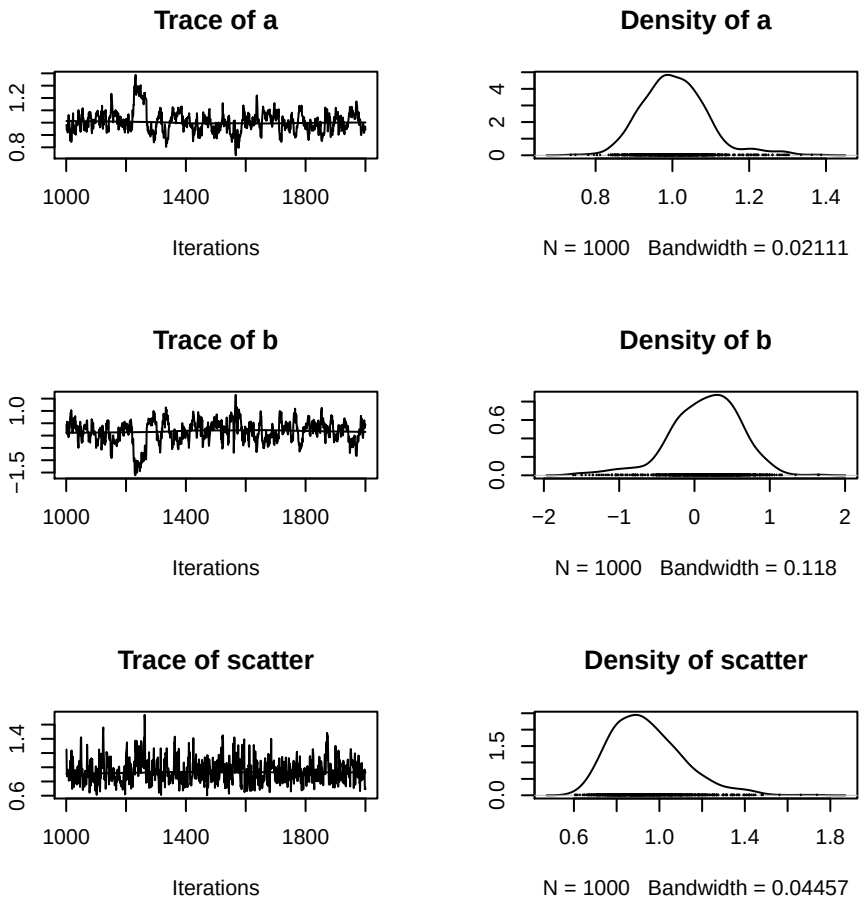

Clearly, the group scatter is larger than the individual measurement error bars allow, implying that one or more unmeasured variables are influencing the y values. The model we'll use is a linear relationship between y and x, with (uniform) measurement error on both y and x and additional scatter in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

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

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

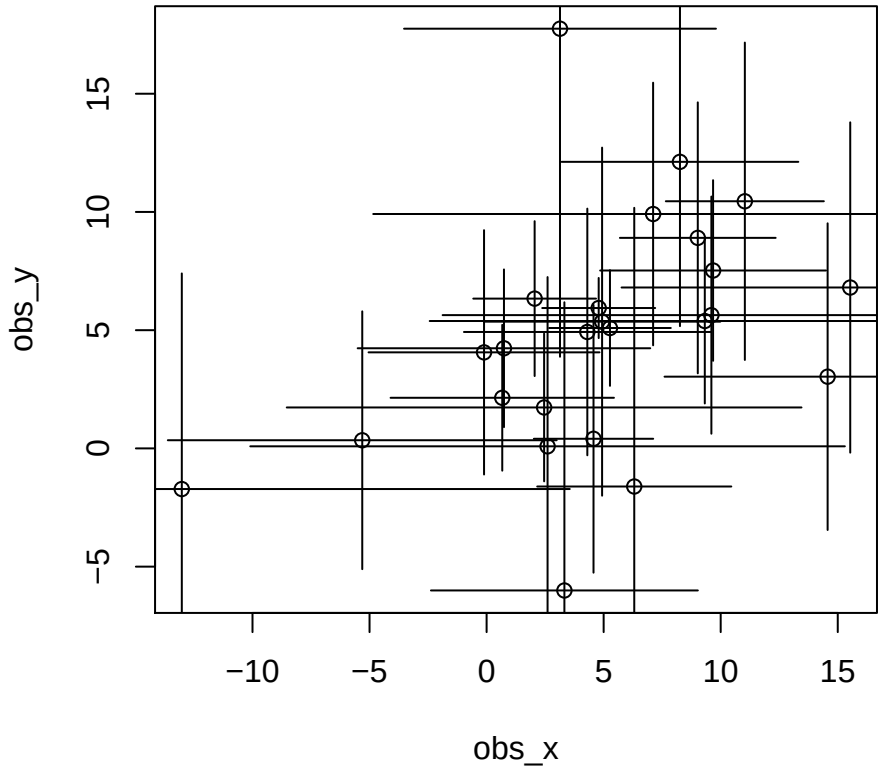
```
plot(output)
```



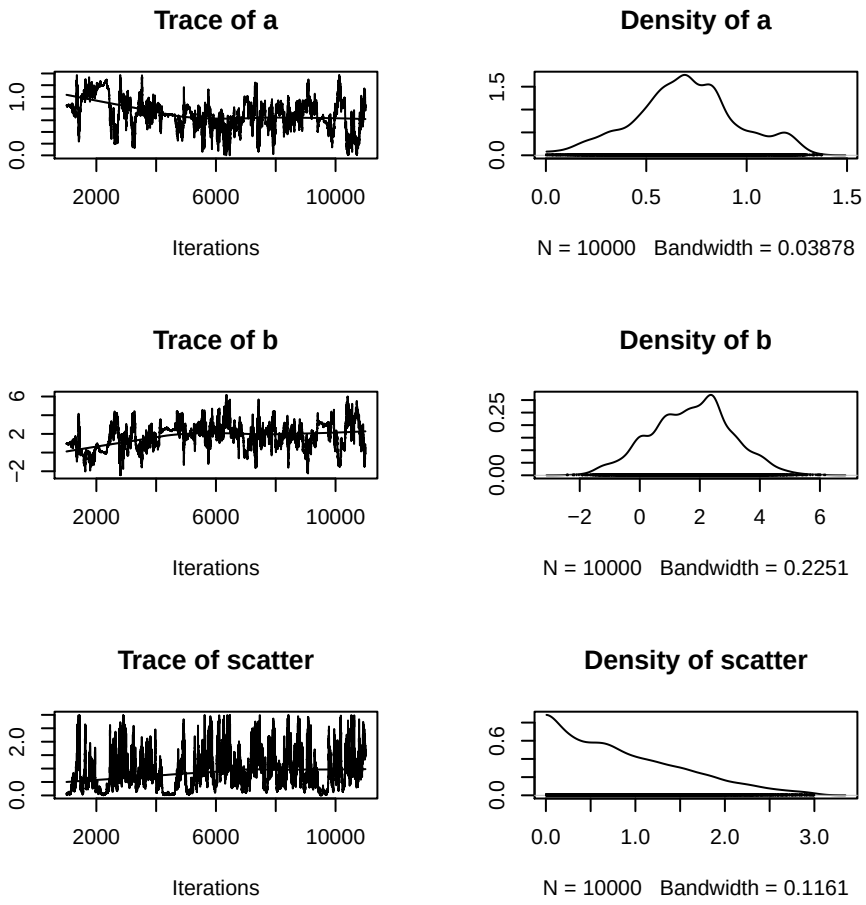

Again, the model has reproduced the parameters used to generate the data. What would happen if we ran the code again, but increased the measurement uncertainty?

```
## these values are the same as in the previous model:
#N = 25
#true_x = runif(N,0,10)
#true_slope = 1
#true_intercept = 0
#scatter = 0.25
#true_y = rnorm(N, true_slope * true_x + true_intercept, scatter)
## known measurement uncertainties (much larger than before)
x_sigma = rlnorm(N, 1, 0.5)
y_sigma = rlnorm(N, 1, 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)
```




This time, the uncertainty bars are much larger, and you would think that very little information could be extracted from these data. Using the same model as before, let's see what the output looks like:



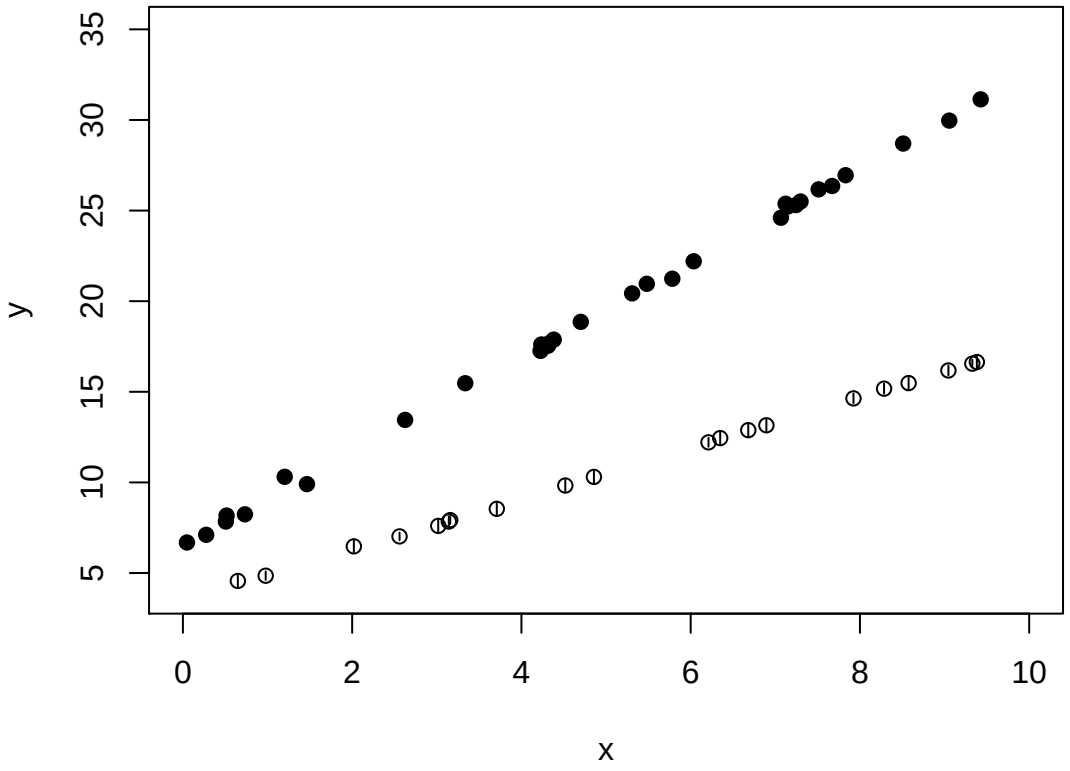

Here, the number of MCMC iterations has been increased to 10,000. With the 1,000 iterations used before, the mixing diagrams did not appear robust. The slope and intercept posterior distributions are not as smooth and are wider than before, and the scatter posterior is maximized far from the true value. As expected, using uninformative data results in uninformative posteriors.

## 3.3 Linear Model with Multiple Datasets and Systematic Error

Systematic error is something that is typically difficult to deal with in statistical analysis, but is easy here. First, let's examine the case when the systematic error is proportional to `x`. Imagine we have two data sets with qualitatively different slopes:

```
## generate some data
slope = 2
intercept = 5

N1 = 20
shift1 = 0.7;
obs_x1 = runif(N1,0,10)
err_y1 = runif(N1, 0.1, 0.2)
obs_y1 = shift1 * rnorm(N1, slope * obs_x1 + intercept, err_y1)

N2 = 30
```

```
shift2 = 1.3;
obs_x2 = runif(N2,0,10)
err_y2 = runif(N2, 0.1, 0.2)
obs_y2 = shift2 * rnorm(N2, slope * obs_x2 + intercept, err_y2)

plot(obs_x1, obs_y1, xlim = c(0,10),ylim = c(4,35), xlab = "x", ylab = "y")
points(obs_x2, obs_y2, pch=19)
segments(obs_x1, obs_y1 - 2*err_y1, obs_x1, obs_y1 + 2*err_y1)
segments(obs_x2, obs_y2 - 2*err_y2, obs_x2, obs_y2 + 2*err_y2)
```



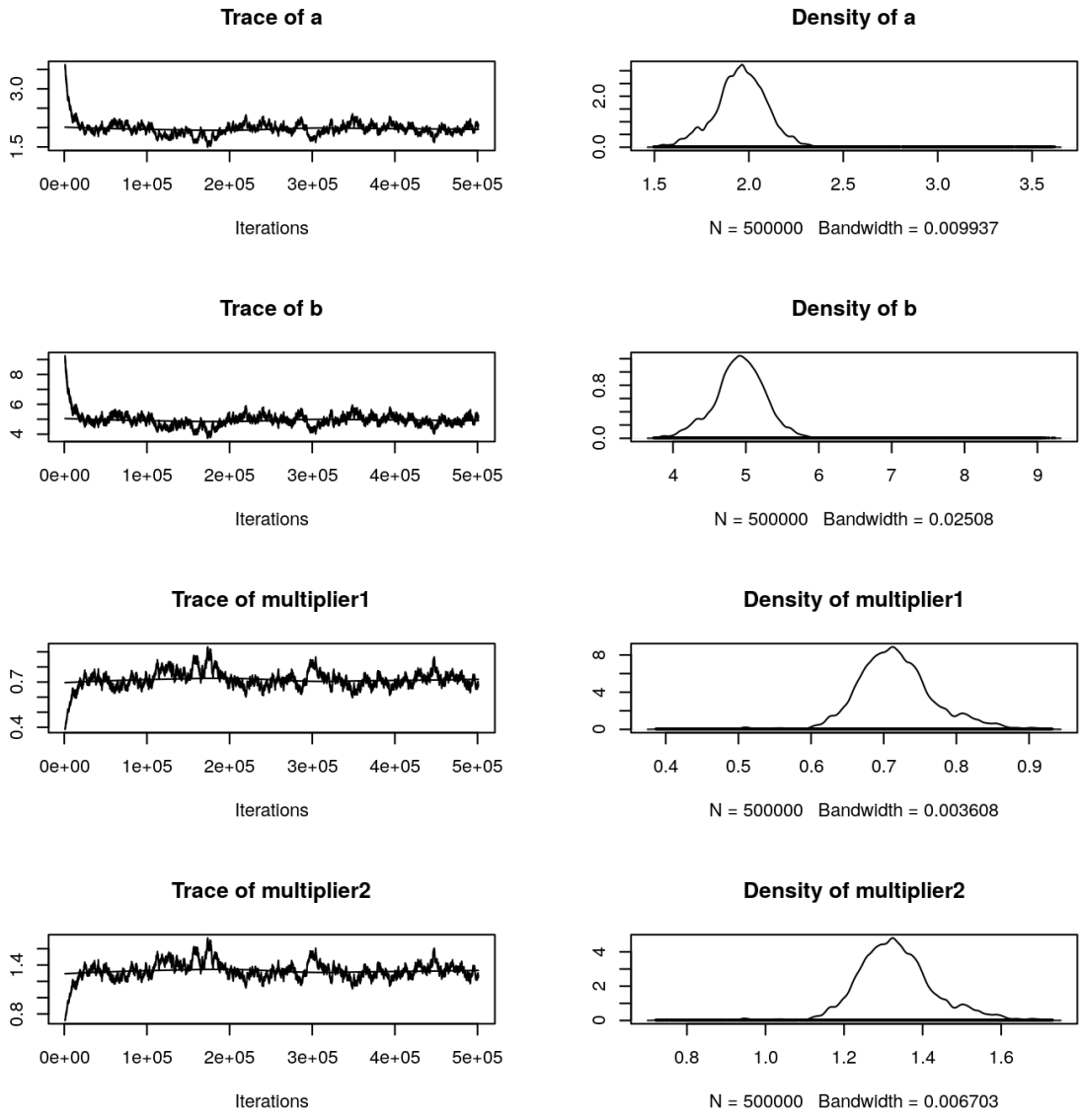

This could be caused by, say, the result of a lab instrument that became miscalibrated between the data collections. Some researchers would take a weighted average of some kind to combine the two datasets, but that method is statistically dubious at best. Let's construct and run a model that includes a multiplicative factor for the two datasets:

```
model_string = "
  model {
    ## priors:
    a ~ dunif(0,5)
    b ~ dunif(-10,10)
    ## shifts must be > 0. Choose centered near 1 (0% shift), with ~0.1 = 10% spread
```

```
    multiplier1 ~ dlnorm(log(1.0), pow(log(1.1),-2))
    multiplier2 ~ dlnorm(log(1.0), pow(log(1.1),-2))

    ## structure:
    for (i in 1:length(obs_x1)) {
        y1[i] = multiplier1 * (a*obs_x1[i] + b)
        obs_y1[i] ~ dnorm(y1[i], pow(err_y1[i],-2))
    }
    for (i in 1:length(obs_x2)) {
        y2[i] = multiplier2 * (a*obs_x2[i] + b)
        obs_y2[i] ~ dnorm(y2[i], pow(err_y2[i],-2))
    }
  }
"

model = jags.model(file = textConnection(model_string),
                   data = list('obs_x1' = obs_x1,
                               'obs_x2' = obs_x2,
                               'obs_y1' = obs_y1,
                               'obs_y2' = obs_y2,
                               'err_y1' = err_y1,
                               'err_y2' = err_y2),
                   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", "multiplier1", "multiplier2"),
                      n.iter=500000,
                      #n.iter =10000,
                      thin=1)
```

Note that the number of iterations has been increased to 500,000 from the standard 10,000. This model's added complexity required more iterations for the Markov chains to converge. The posterior distributions are amazingly accurate:

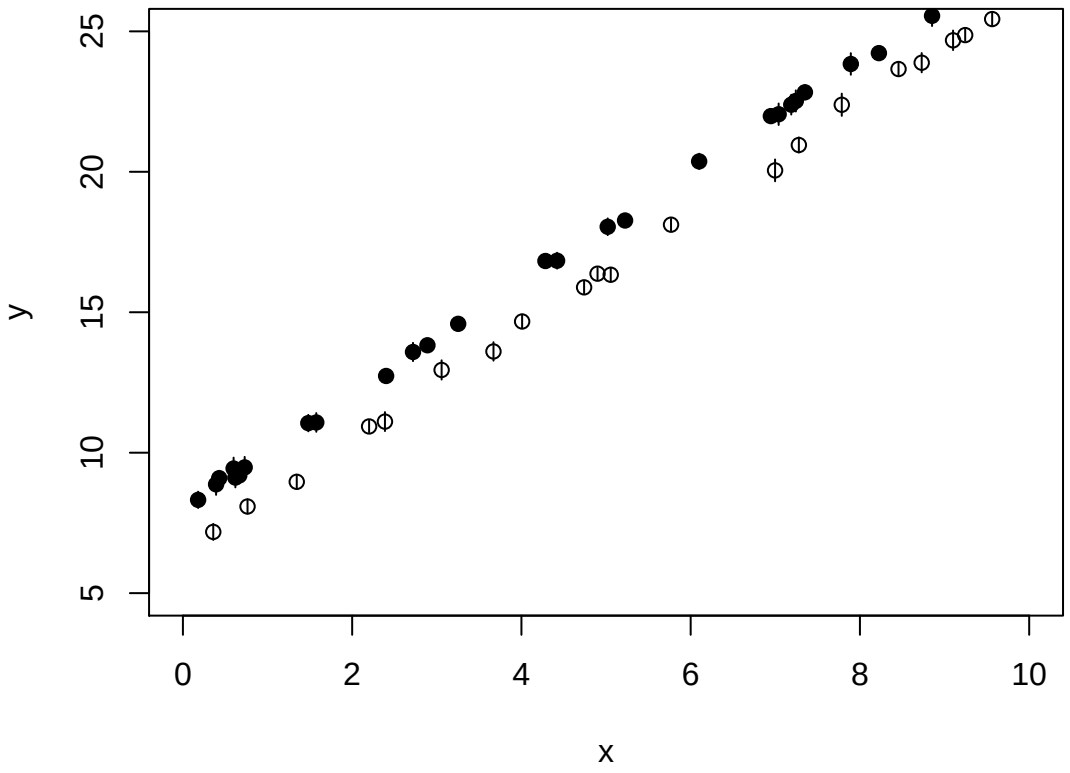
```
plot(output)
```




The slope, intercept, and both systematic error multipliers are all centered exactly on their true values. Now, let's look at an additive systematic error. Imagine we have two datasets, each with a positive systematic error (implying that taking a weighted average would not help very much).

```
## generate some data
slope = 2
intercept = 5

N1 = 20
shift1 = 1.5;
obs_x1 = runif(N1,0,10)
err_y1 = runif(N1, 0.1, 0.2)
obs_y1 = shift1 + rnorm(N1, slope * obs_x1 + intercept, err_y1)

N2 = 30
```

```
shift2 = 3;
obs_x2 = runif(N2,0,10)
err_y2 = runif(N2, 0.1, 0.2)
obs_y2 = shift2 + rnorm(N2, slope * obs_x2 + intercept, err_y2)

plot(obs_x1, obs_y1, xlim = c(0,10),ylim = c(5,25), xlab = "x", ylab = "y")
points(obs_x2, obs_y2, pch=19)
segments(obs_x1, obs_y1 - 2*err_y1, obs_x1, obs_y1 + 2*err_y1)
segments(obs_x2, obs_y2 - 2*err_y2, obs_x2, obs_y2 + 2*err_y2)
```



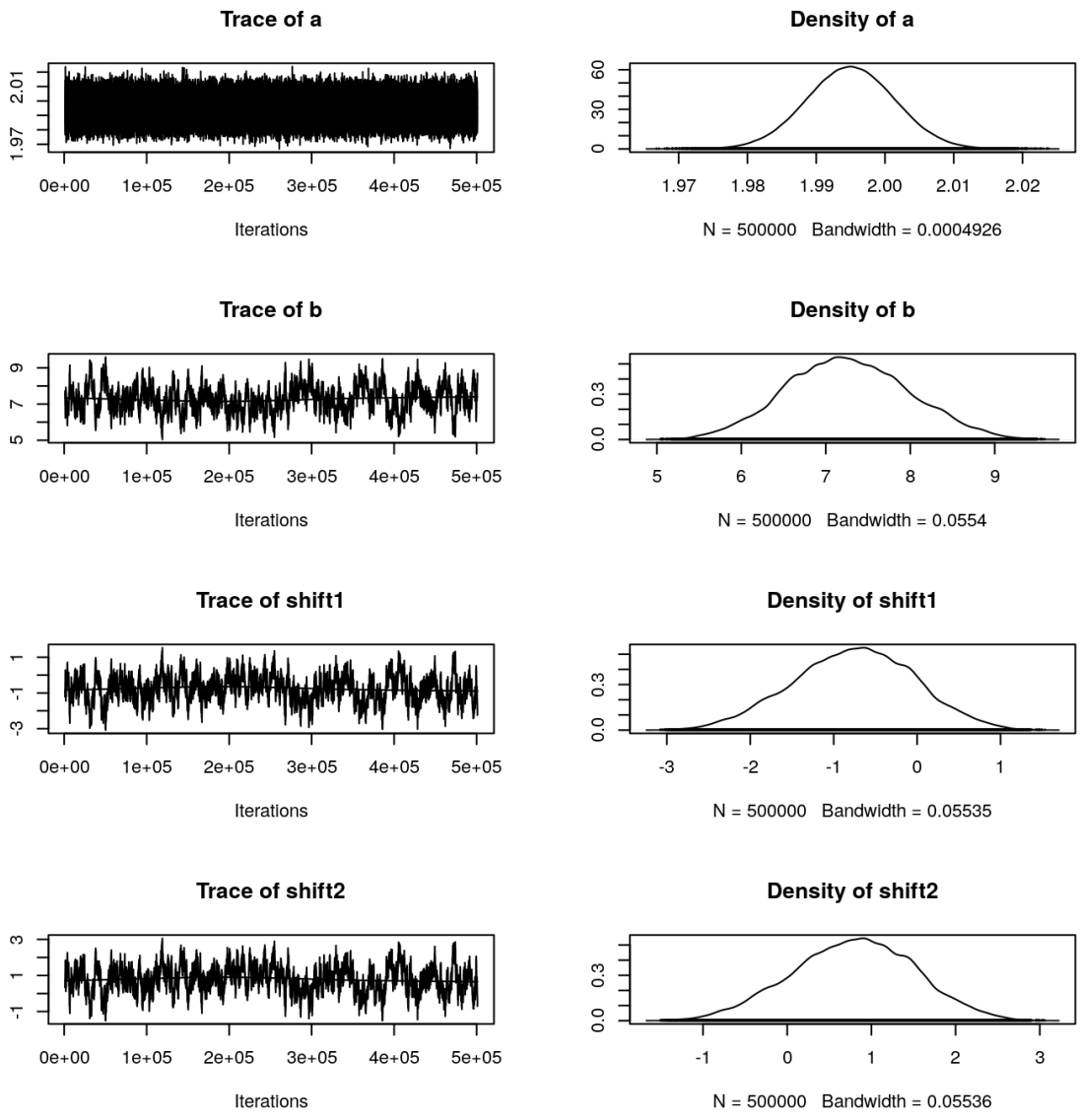

The model will be similar to before:

```
model_string = "
  model {
    ## priors:
    a ~ dunif(0,5)
    b ~ dunif(-10,10)
    shift1 ~ dnorm(0, 1)
    shift2 ~ dnorm(0, 1)

    ## structure:
```

```
    for (i in 1:length(obs_x1)) {
        y1[i] = shift1 + (a*obs_x1[i] + b)
        obs_y1[i] ~ dnorm(y1[i], pow(err_y1[i],-2))
    }
    for (i in 1:length(obs_x2)) {
        y2[i] = shift2 + (a*obs_x2[i] + b)
        obs_y2[i] ~ dnorm(y2[i], pow(err_y2[i],-2))
    }
  }
"

model = jags.model(file = textConnection(model_string),
                   data = list('obs_x1' = obs_x1,
                               'obs_x2' = obs_x2,
                               'obs_y1' = obs_y1,
                               'obs_y2' = obs_y2,
                               'err_y1' = err_y1,
                               'err_y2' = err_y2),
                   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", "shift1", "shift2"),
                      n.iter = 500000,
                      thin=1)
```

Again the number of iterations has been increased to 500,000. The posterior distributions are amazingly accurate:

```
plot(output)
```




This time, the results are rather disappointing. The systematic error posteriors are not centered near the true values at all. The intercept posterior has been shifted as well.

how to fix additive systematic errors?

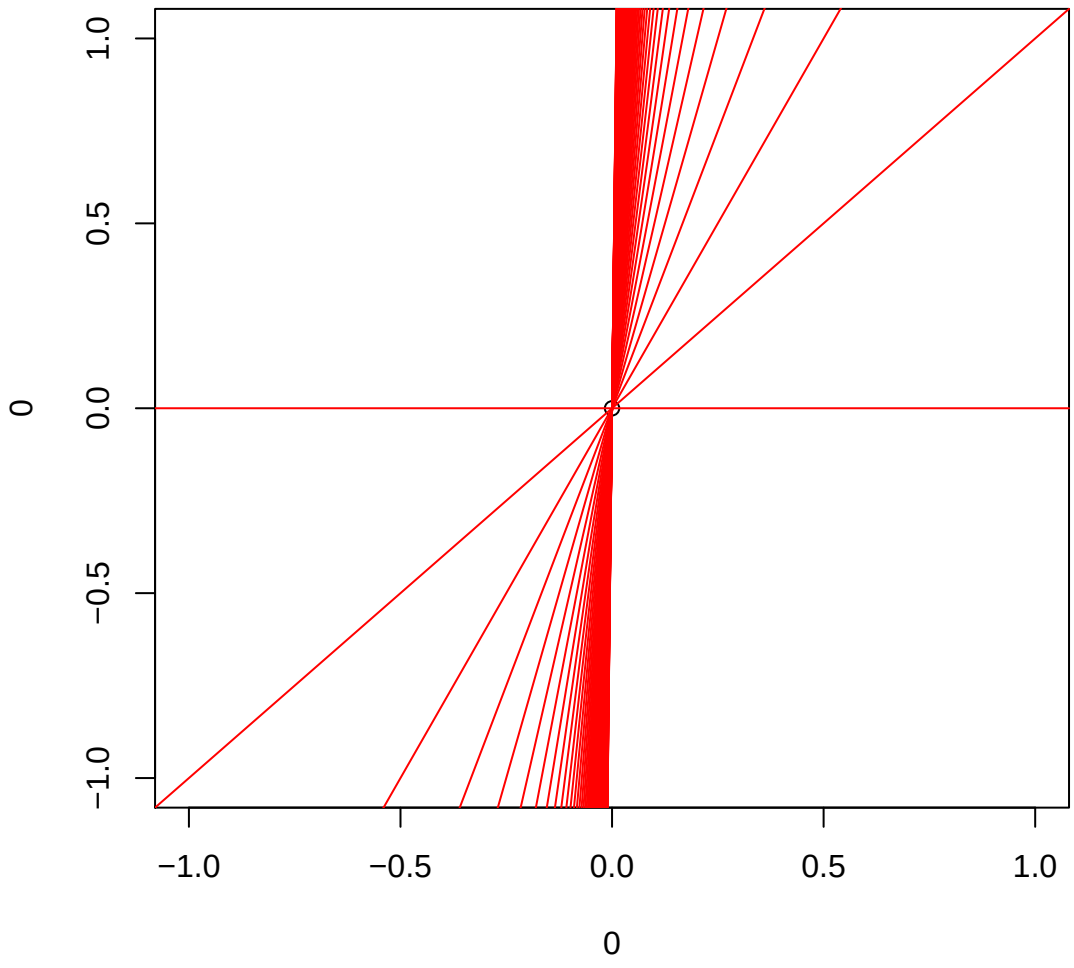
# 4 Choosing a Prior

The choice of prior to use for parameters can be somewhat tricky. If the parameters in question are well-studied, there may be reasonably precise bounds around the parameter value. However, previous information doesn't exist about all parameters, so sometimes "vague priors" and "uninformative priors" are used to let the model have more freedom in determining parameter values. We'll focus first on this kind of prior.

## 4.1 Uninformative Priors

A common choice of uninformative prior is the "flat" or uniform prior. In rjags code, a flat prior between two numbers $a$ and $b$ is represented by `dunif(a,b)`. This is easy to understand and weights each value in the interval $[a, b]$ with equal value. However, the "fairness" of the uniform prior does need some careful consideration. Consider the case when the parameter of interest is the slope of a line. Using a uniform prior on $[0, 100]$ means that lines with slope greater than unity are 99% more likely than lines with slope less than unity. This is easy to see graphically: if we plot lines with slope $0, 1, 2, 3, ...$, we get

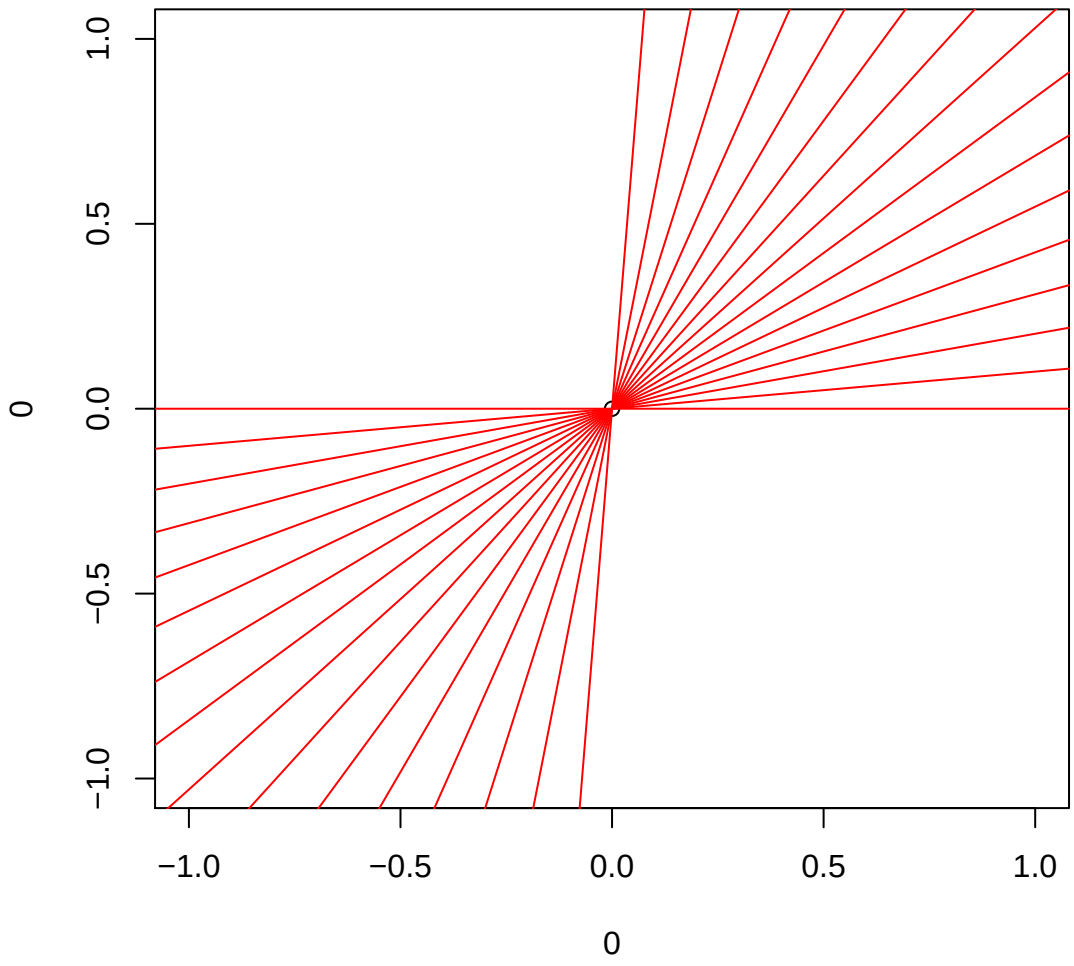
```
plot(0,0)
for(i in 0:100){
  abline(0,i,col='red');
}
```




Clearly, more steep lines exist than shallow lines, so this prior is probably less "fair" than you might

have thought. A more fair alternative might be a uniform prior on $\phi$, the angle between the line and the x-axis. Then the slope corresponding to each $\phi$ is $b = \tan\phi$ (because slope is $\Delta y/\Delta x = \sin\phi/\cos\phi$). To see this graphically:

```
plot(0,0)
phi_array = seq(from = 0, to = pi/2, by = 0.1)
for(phi in phi_array){
  abline(0,tan(phi),col='red');
}
```




So, if a uniformly-weighted prior is desired, sometimes it takes a bit of thought to achieve. It is a good idea to think about whether you would rather be totally ignorant of the scale of the parameter rather than the value of the parameter – if you are ignorant of the parameter's order of magnitude, then it is as likely to be in $[1, 10]$ as it is to be in $[100, 1000]$. This corresponds to a logarithmic decaying prior, which could be created as a uniform prior on the power of ten and then taking the logarithm for the parameter value.

In the past, people have put a lot of effort into find the "best" non-informative prior to use. The current concensus is that there is not one single uninformative prior that is the best to use for all models.

Vague priors are usually very broad in the sense that they cover a large amount of the parameter's possible values. Examples of this include a wide uniform distribution and a Gaussian with large variance. A vague prior may still have some information; for instance, using a wide lognormal distribution will require the posterior to be non-negative.

## 4.2 Informative Priors

If some information about the parameters is already known before the data analysis, this can (and should) be encoded in the prior. For example, if previous experiments have estimated the value of the parameter of interest, you might use a prior of a Gaussian centered on the existing estimate. Choosing an informative prior is somewhat unsettling to some people – shouldn't statistical inference be independent of "expert opinion"? The arbitrary choice of prior distributions has been a hot debate between statisticians for decades. That won't be covered here, but instead we will sidestep the issue and examine the sensitivity of an analysis to the choice of prior.