Lab #X: A Calibration Problem and Markov chain Monte Carlo (Kelsey L. Ruckert, Tony E. Wong, Yawen Guan, Murali Haran, and Patrick J. Applegate)

Learning objectives

After completing this exercise, you should be able to

- explain what convergence means in terms of a Markov chain
- describe several methods on how to test for convergence

Introduction

In the last chapter, we introduced Bayesian inference and Markov chain Monte Carlo (MCMC). In particular, we looked at why the Metropolis-Hastings algorithm works by satisfying detailed balance and calibrated a linear model to observations using MCMC. However, how do we know whether we can trust the results? When making inferences about probability distributions using MCMC, it is necessary to make sure that the chains (vectors of parameter values) show evidence of convergence. A converged chain is one in which the statistical properties of the chain do not change as new random values are generated. Typically, chains are not converged when they are begun and reach convergence only after many random numbers ("iterations") have been generated.

In this exercise, you will examine multiple ways on how to test for evidence of convergence using an application of Bayesian inference with MCMC to data with non-correlated residuals. You will also see how the amount of data and how long you run the calibration impact the results and evidence of convergence. If you have not already completed the previous chapter on MCMC, go back and complete that exercise as the concepts in this chapter build off of the previous one.

Tutorial

If you have not already done so, download the .zip file containing the scripts associated with this book from www.scrimhub.org/raes. Put the file labX_sample.R in an empty directory. Open the R script labX_sample.R and examine its contents.

In this tutorial, you will estimate parameters for a linear physical model using Bayesian inference with MCMC and test for evidence of convergence. Before you proceed, first clear away any existing variables or figures and install the `mcmc` package and a package for testing convergence, `coda`.

```r
# Clear away any existing variables or figures.
rm(list = ls())
graphics.off()

# Install and read in packages.
# install.packages("coda")
# install.packages("mcmc")
# install.packages("batchmeans")
library(coda)
library(mcmc)
library(batchmeans)

# Set the seed for random sampling.
# All seeds in this tutorial are arbitrary.
set.seed(1)
```

For simplicity, the data have a linearly increasing trend over time, have independent measurement errors, and you will use the same simple model as before,

$$y_t = f(\theta, t) + \epsilon_t,$$
$$f(\theta, t) = \alpha \times t + \beta,$$
$$\epsilon_t \sim N(0, \sigma^2), \quad \theta = (\alpha, \beta),$$

where θ denotes the physical parameters, $f(\theta, t)$ is the model output, y_t is the observed data y_t, and ϵ_t are the measurement errors.

```r
# Read in some observations with non-correlated measurement error
# (independent and identically distributed assumption).
data <- read.table("observations.txt", header=TRUE)
t <- data$time
observations <- data$observations

# Plot data.
par(mfrow = c(1,1))
plot(t, observations, pch = 20, xlab = "Time", ylab = "Observations")

# Set up a simple linear equation as a physical model.
model <- function(parm,t){ # Inputs are parameters and length of data
  model.p <- length(parm) # number of parameters in the physical model
  alpha <- parm[1]
  beta <- parm[2]
  y.mod <- alpha*t + beta # This linear equation represents a simple physical model
  return(list(mod.obs = y.mod, model.p = model.p))
}
```

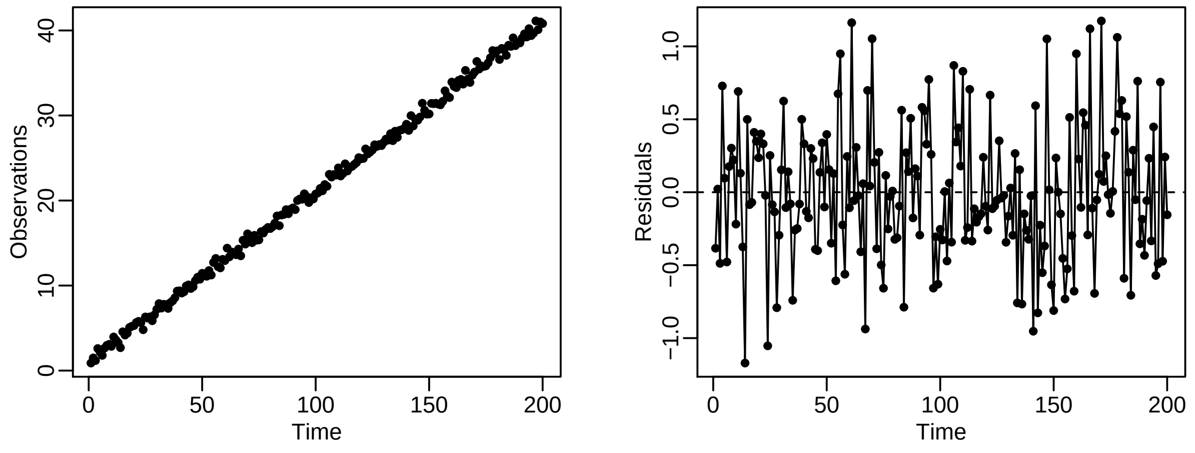

Figure 1: The original data (left) and residuals (right) as a function of time. The original data has a true value of parameter $\alpha = 0.02$, $\beta = 1$, $\sigma = 0.5$, and autocorrelation = 0.

Setting up the prior information and the likelihood function

Now that you have some data, you can obtain estimates of the parameters as well as their uncertainties. Before running the MCMC calibration, it is necessary to set up some information on what you know about the physical model and data. For instance, you must specify initial parameter values for the Markov chains, a prior distribution for the parameters, and set up how to calculate the log posterior.

In the previous chapter, you estimated the physical model parameters (α, the slope, and β, the intercept) while setting the statistical parameter (σ, our estimate of the standard deviation) to a constant of 1. Additionally, you used initial parameter values that were randomly generated. In physical applications, it is likely better to use more informative initial parameter values that are based on, for example, a mechanistic understanding of the modeled system, previous published estimates, or by using an optimization technique. In this example, you will use the `optim` command to obtain values for α and β that minimize the root mean squared error, and use these as the Markov chain initial values, but you must, in turn, provide initial estimates for α and β to the `optim` command. These initial values can be a random guess for α and β (try $\alpha = 0.5$ and $\beta = 2$). Try using several different initial values. If the Markov chains are converged, then their distributions of parameters should converge to the same posterior distribution, regardless of the initial values specified for each chain. Note that the `optim` command will fail if there is a multimodal or complicated distribution, so try a few starting values for α and β or use the `DEoptim` command. The residuals from this optimized physical model simulation can be used to estimate a reasonable starting value for σ. Later on, you will apply the likelihood function to update these initial starting values to account for the prior parameter distributions.

```r
# Sample function for calculating the root mean squared error given a set of
# parameters, a vector of time values, and a vector of observations.
fn <- function(parameters, t, obs){
  alpha <- parameters[1]
  beta <- parameters[2]
  data <- alpha*t + beta
  resid <- obs - data
  rmse <- sqrt(mean(resid^2))
  # return the root mean square error
  return(rmse)
}

# Plug in random values for the parameters.
parameter_guess <- c(0.5, 2)

# Optimize the physical model to find initial starting values for parameters.
# For optim to print more information add the arguments:
# method = "L-BFGS-B", control=list(trace=6))
result <- optim(parameter_guess, fn, gr=NULL, t, observations)
start_alpha <- result$par[1]
start_beta <- result$par[2]
parameter <- c(start_alpha, start_beta)

# Use the optimized parameters to generate a fit to the data and
# calculate the residuals.
y.obs <- model(parameter,t)
res <- observations - y.obs$mod.obs
start_sigma <- sd(res)

par(mfrow = c(1,1))
plot(res, type = "l", ylab = "Residuals", xlab = "Time")
points(res, pch = 20)
abline(h = 0, lty = 2)
```

Previously, you used priors as uniform distributions with no bounds. In physical applications, you should use a mechanistic understanding of the modeled system to set up prior parameter distributions. Assuming that there is relatively little information known about the data and physical model, how would you come up with prior distributions? Maybe test out several values, calculate the root mean square error for several parameter sets, or look at the output from the `optim` call (try setting the method to "L-BFGS-B" and trace to 6). For this tutorial, you will set uniform prior distributions by specifying a lower and upper bound for each parameter. Note that in a physical application, using non-uniform priors may be more informative. Looking at the output from the `optim` call, it shows that α did not vary much from 0.19 and that the final root mean square error is roughly 0.46. Based on this information, you can infer that α is relatively small and therefore set the lower and upper bound to be -1 and 1. Similarly, the β parameter fluctuates between 2 and 1 in the output from the optim call. From this, you can set the lower and upper bound of β to be -1 and 3. Now look at the original data in Figure 2, notice how the data are closely grouped together and the residuals display deviations of less than 2 from the trend (0). Using this information, you can imply that the σ cannot be negative and that it is small, hence you can set the lower and upper bound of parameter σ to be 0 to 1. In this application, the $/sigma$ is converted to variance $/sigma^2$ in the likelihood function, however calibrating the variance $/sigma^2$ directly is a viable option. Calibrating the variance directly should be done if using a conjugate prior such as an inverse gamma distribution.

```r
# Set up priors.
bound.lower <- c(-1, -1, 0)
bound.upper <- c( 1,  3, 1)
```

In this tutorial, we provide the likelihood function, $L(y \mid \theta)$, as a separate script to be sourced. Open up **iid_obs_likelihood.R** and note the differences in how we estimate the log posterior from how it was coded in the previous chapter. Here, we split the log likelihood function, log prior, and log posterior into separate R functions. The likelihood is now the product of potentially many probabilities (between 0 and 1), so with lots of data will give underflow (Figure 1). Underflow happens when an operation performed on a value smaller than the smallest magitude non-zero number, which is often rounded to zero. Again, we work with the log-probability distributions to solve this problem.

The log likelihood function, `log.lik`, reads in the parameters (physical and statistical) and produces model predictions. These predictions are then evaluated against the observations to estimate the residuals. The residuals along with the statistical parameter σ (the standard deviation) are used to estimate the *likelihood* of the parameter values based on the observations. Note that in the previous chapter, the log likelihood was coded as an equation,

$$L(y \mid \theta) = -\frac{N}{2}\ln(2\pi) - N\ln(\sigma) - \frac{1}{2}\frac{\sum(y - f(\theta, t))}{\sigma^2},$$

while here it uses the `sum` and `dnorm` commands. Both versions are the same and should produce the same value when interchanged.

In the log prior function, `log.pri`, the parameters are set as uniform distributions with an upper and lower bound. When the parameters are read in the function checks whether the parameter values are within in upper and lower bounds. If they are within the distribution, then the prior is set to 0 to represent a uniform distribution because the natural log of 1, ln(1), is 0. If the parameter values fall outside of the boundary, then the prior is set to `-Inf`. The function returns `-Inf` because there is 0 probability associated with parameters outside of the range.

Lastly, the log posterior function, `log.post`, estimates the posterior probability based on the likelihood and the prior. In this function, both the prior and likelihood functions described above are called. The parameters are first evaluated to check whether they satisfy the prior distribution. If the parameters pass with a nonzero prior probability, then the log likelihood function is called to estimate the likelihood of the parameter values based on the observations. The function then estimates the posterior probability by combining the likelihood and the prior probability. If the parameters have a prior probability of zero, then the likelihood function is not run and the posterior probability is set to `-Inf` (a probability of 0). It is important to point out that this process of not evaluating the model at parameter values outside the prior range (that will not be accepted anyway) can save valuable time. For instance, if someone was to code the log posterior function and the MCMC function by scratch, this process might be overlooked, and more expensive model (in terms of computer storage and time) can be a huge burden.

```r
# Name the parameters and specify the number of physical model parameters (alpha and beta).
# sigma is a statistical parameter and will not be counted in the number.
parnames <- c("alpha", "beta", "sigma")
model.p <- 2

# Load the likelihood model for measurement errors
source("iid_obs_likelihood.R")

# Optimize the likelihood function to estimate initial starting values
p <- c(start_alpha, start_beta, start_sigma)
p0 <- c(0.3, 1, 0.6) # random guesses
p0 <- optim(p0, function(p) -log.post(p))$par
print(round(p0,4))
```

Bayesian Inference using MCMC

For the MCMC calibration, this tutorial uses the `metrop` function from the package called `mcmc`. In the console, open up the `metrop` function documentation by typing `help(metrop)`. This function runs the MCMC calibration using the Metropolis-Hasting algorithm where a random walk is implemented. Both the `metrop` function and the function we coded in the previous chapter should be the same. By using the `metrop` function, you increase the efficiency since it has already been optimized and generalized.

The `metrop` function calls for multiple input arguments including `obj`, `initial`, `nbatch`, and `scale`. The `obj` input calls for a function that estimates the unnormalized log posterior. This has been previously coded as the `log.post` function in **iid__obs__likelihood.R**, where the function estimates the posterior probability by combining the likelihood and the prior probability, as described in the previous section. The next input is `initial`, which is the initial parameter values `p0` that you calculated using the `optim` command and the likelihood function. The argument `nbatch` sets the number of iterations to run the MCMC calibration. In exercise one, you are asked to vary the number of iterations, but for the first pass set the number of iterations to 1000. Lastly, set the `scale`. The `scale` controls the step size. Remember that the step size is important because it has an impact on how well the Markov chain explores the parameter space. A step size that is "too small" will require a larger number of iterations to reach convergence, while one that is "too large" could possibly reject many acceptable values; the previous chapter displays a figure with an example depicting this issue. Here, the step sizes are set to values that are reasonable for this tutorial. How would you come up with step sizes if they were not given? Maybe test out several step size values, check out the `proposal.matrix` function in **iid__obs__likelihood.R**, or look at the `MCMC` function in `adaptMCMC`.

```r
# Set the step size and number of iterations.
step <- c(0.001, 0.01, 0.01)
NI <- 1E3

# Run MCMC calibration.
mcmc.out <- metrop(log.post, p0, nbatch = NI, scale = step)
prechain <- mcmc.out$batch
```

Checking the acceptance rate

Check that the acceptance rate didn't reject or accept too many values. For instance, the acceptance rate should be higher than 23.4% because it explores only three parameters.

```r
# Print the acceptance rate as a percent.
acceptrate <- mcmc.out$accept * 100
cat("Accept rate =", acceptrate, "%\n")
```

Testing for convergence

We seek an answer to that initial question from the introduction: How do we know when the physical model is calibrated? This question is answered by diagnosing when the MCMC iterates have "converged" to sampling from the posterior distribution. There are many ways to test for convergence: visual inspection, Monte Carlo Standard Error, Gelman and Rubin diagnostic, Geweke diagnostic, Raftery and Lewis diagnostic, Heidelberg and Welch diagnostic, and many others. It is a good practice to use multiple tests to check for convergence of the Markov chains (the vector output for each parameter). The example here, will focus on four tests: (1) visual inspection, (2) Monte Carlo standard error, (3) the Heidelberg and Welch diagnostic, and (4) the Gelman and Rubin diagnostic (the potential scale reduction factor).

Before you begin testing for convergence or doing further analysis, subtract the *burn-in*. Even if the chains are converged, the first values in the sample may differ considerably from the converged distribution. Here, we throw away the first 1% of the sample for the *burn-in period*. Even though a 1% burn-in is used here, other applications may require different lengths or may not need a burn-in period.

```r
# Identify the burn-in period and subtract it from the chains.
burnin <- seq(1, 0.01*NI, 1)
mcmc.chains <- prechain[-burnin, ]
```

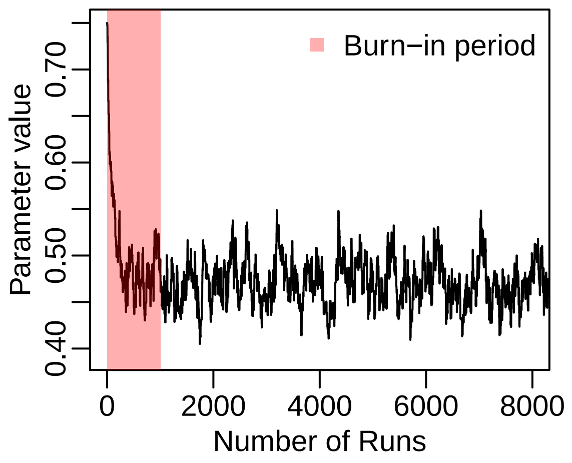

Figure 2: The chain of a single parameter showing that is well mixed and converged, but the initival values differ considerably from the converged distribution. The highlighted area shows the burn-in period that is used to remove the effects of starting values.

Visual inspection

One way to check for convergence is by looking at how the chain jumps around in the parameter space or how well it mixes. You can visualize this using a plot of the iteration number versus the parameter value (a trace plot, also known as a history plot). If the Markov chain becomes stuck in certain areas of the parameter space or if the chain is taking a long time to explore the parameter space, then the chain has not converged and indicates that either the chain needs to be run longer or there is some other kind of issue.

```r
## Check #1: Trace Plots:
par(mfrow = c(2,2))
for(i in 1:3){
  plot(mcmc.chains[ ,i], type="l", main = "",
       ylab = paste('Parameter = ', parnames[i], sep = ''), xlab = "Number of Runs")
}
```

Monte Carlo standard error

The point of using MCMC is to approximate expectations. These approximations are not exact, but are off by some amount known as the *Monte Carlo standard error* (MCSE). A way to calculate the MCSE is through the *batch means method.* The batch means method estimates uncertainty by calculating means and standard errors (as square root of sample variances) of *batch means* of the markov chain (Jones et al. 2006; Flegal et al., 2008). To perform this method, suppose you have a markov chain $X = \{X_1, X_2, X_3, ...\}$. The Monte carlo estimate or the sample mean is

$$\bar{g_n} := \frac{1}{n}\sum_{i=1}^{n} \bar{X_i} \quad \text{as } n \to \infty,$$

where n is the number of iterations in the markov chain and $\bar{X_i}$ is a sample mean for a batch. In batch means, the output is broken into a number of blocks or batches of b size. By dividing the simulation into batches, the batch means estimate of σ_g^2 becomes

$$\hat{\sigma}_g^2 = \frac{b}{a-1}\sum_{j=1}^{a}(\bar{Y_j} - \bar{g_n})^2$$

$$\bar{Y_j} := \frac{1}{b}\sum_{i=(j-1)b+1}^{a} \bar{X_i} \quad \text{for } j = 1, ..., a.$$

The σ is denoted as $\hat{\sigma}$ because it is unkown and is estimated from the data. Using consistent batch means, you can estimate the MCSE of $\bar{g_n}$ as

$$\text{MCSE} = \frac{\hat{\sigma}_g}{\sqrt{n}}$$

This method is performed in R via the `batchmeans` package. The command `bm()` performs consistant batch means estimation on a Markov chain returning the mean and the MCSE. Below, `bmmat()` is used because it performs consistant batch means estimation on a matrix of Markov chains and in this case you have three chains.

```
## Check #2: Monte Carlo Standard Error:
# est: approximation of the mean
# se: estimates the MCMC standard error
bm_est <- bmmat(mcmc.chains)
print(bm_est)
```

In theory, if you run the chain infinitely, then the standard errors of the approximations will be virtually 0. In practice, since the chain is not run infinitely, you can calculate the standard error of the approximations and decide if it is "small enough" to stop the chain. You can calculate if the estimate is small enough by assessing whether we trust the amount of significant figures in the standard error. The first step after calculating the MCSE and the Monte Carlo estimate is to estimate the z-statistic,

$$z = \frac{(\mu_X - \bar{g_n})}{\frac{\hat{\sigma}_g}{\sqrt{n}}},$$

where μ_X is the mean of the chain, $\bar{g_n}$ is the Monte Carlo estimate, and $\frac{\hat{\sigma}_g}{\sqrt{n}}$ is the MCSE. Here, we use the z-distribution instead of the t-distribution because a t-distribution with a degrees of freedom $(\sqrt{n})$ equal to or greater than 30 is very close to the z-distribution. Typically, MCMC is run with a large number of iterations (tens of thousands), which exceeds a degrees of freedom of 30. Unless the sample is highly correlated or has a *low* degrees of freedom, the z-distribution can be used. Next, is to calculate the half-width h_α of the interval by multiplying the z-statistic and the MCSE,

$$h_\alpha = z(\frac{\hat{\sigma}_g}{\sqrt{n}}).$$

If the lower and upper bound of the confidence intervals ($\bar{g_n} \pm h_\alpha = \bar{g_n} \pm z(\frac{\hat{\sigma}_g}{\sqrt{n}})$ for $n \geq 30$) can be rounded to equal the Monte Carlo estimate, then the significant figure in the MCSE can be trusted. If the error is small enough, then no more samples are needed and the chain is considered converged. Additionally, reporting the MCSE provides a measure of the accuracy and quality of the estimates (Flegal et al., 2008).

```r
# Evaluate the number of significant figures
z <-
half_width <- rep(NA, length(parnames))
interval <- matrix(data = NA, nrow = 3, ncol = 2,
                   dimnames = list(c(1:3), c("lower_bound", "upper_bound")))
for(i in 1:length(parnames)){
z[i] <- (mean(mcmc.chains[,i]) - bm_est[i ,"est"])/bm_est[i ,"se"]
half_width[i] <- z[i] * bm_est[i ,"se"]
interval[i,1] <- bm_est[i ,"est"] - half_width[i]
interval[i,2] <- bm_est[i ,"est"] + half_width[i]
}
print(interval)
```

Heidelberger and Welch diagnostic

The Heidelberger and Welch diagnostic tests whether each parameter chain has stabilized. This test is split into two parts. The first part of the test examines whether the chain comes from a *stationary distribution*. A stationary distribution means that no matter the starting state, the distribution is unchanged and stable over time. If the chain does not come from a stationary distribution, then the first 10% of the chain will be thrown away and the test runs again. The test will continue to repeat this process until either the chain passes the test or half of the chain has been thrown away and still failed. If the first part is passed, then the test will move on to the second part using the portion of the chain that passed the stationarity test. In part two, a 95% confidence interval is calculated for the mean of the remaining chain(s). *Half of the width* of the 95% confidence interval is compared to the mean. If the ratio between the half-width number and the mean is less than the user-defined tolerance value, ϵ, then the chain passes the test. If the test fails in either part, this suggests that the chains have not converged and need to be run longer. In R, the Heidelberger and Welch diagnostic is carried out in the `heidel.diag` function. Type `help(heidel.diag)` and check out the function arguments.

```r
## Check #3: Heidelberger and Welch's convergence diagnostic:
heidel.diag(mcmc.chains, eps = 0.1, pvalue = 0.05)
```

Gelman and Rubin diagnostic

The Gelman and Rubin diagnostic evaluates the *potential scale reduction factor*. The potential scale reduction factor monitors the variance within the chain and compares it against the variance between the chains and if they look similar, then this is evidence for convergence. It is necessary to estimate the "between-chain variance", so you need to run at least one additional MCMC calibration to estimate the potential scale reduction factor. Each additional MCMC calibration needs to be run with a different random seed and a different set of starting values. All seeds in this tutorial are arbitrary and the different initial values are to ensure that posteriors are independent from the initial values.

In this case, run the MCMC calibration three more times. Each set of chains must be converted into MCMC objects and combined into a list in order to use the R routine `gelman.diag` to calculate these diagnostics. The Gelman and Rubin diagnostic then estimates the median and 97.5% quantile of the potential scale reduction factor. We adopt the standard that the chains are considered to be converged if the point estimate and the upper limit potential scale reduction factor is no larger than 1.1. The results can also be visualized using the `gelman.plot()` function. Run the MCMC calibration longer, if the test fails.

```
## Check #4: Gelman and Rubin's convergence diagnostic:
set.seed(111)
p0 <- c(0.05, 1.5, 0.6) # Arbitrary choice.
mcmc.out2 <- metrop(log.post, p0, nbatch=NI, scale=step)
prechain2 <- mcmc.out2$batch

set.seed(1708)
p0 <- c(0.1, 0.9, 0.3) # Arbitrary choice.
mcmc.out3 <- metrop(log.post, p0, nbatch=NI, scale=step)
prechain3 <- mcmc.out3$batch

set.seed(1234)
p0 <- c(0.3, 1.1, 0.5) # Arbitrary choice.
mcmc.out4 <- metrop(log.post, p0, nbatch=NI, scale=step)
prechain4 <- mcmc.out4$batch

# The burn-in has already been subtracted from the first chain.
# Thus, the burn-in only needs to be subtracted from the three other
# chains at this point.
mcmc1 <- as.mcmc(mcmc.chains)
mcmc2 <- as.mcmc(prechain2[-burnin, ])
mcmc3 <- as.mcmc(prechain3[-burnin, ])
mcmc4 <- as.mcmc(prechain4[-burnin, ])

set.seed(1) # revert back to original seed
mcmc_chain_list <- mcmc.list(list(mcmc1, mcmc2, mcmc3, mcmc4))
gelman.diag(mcmc_chain_list)
gelman.plot(mcmc_chain_list)
```

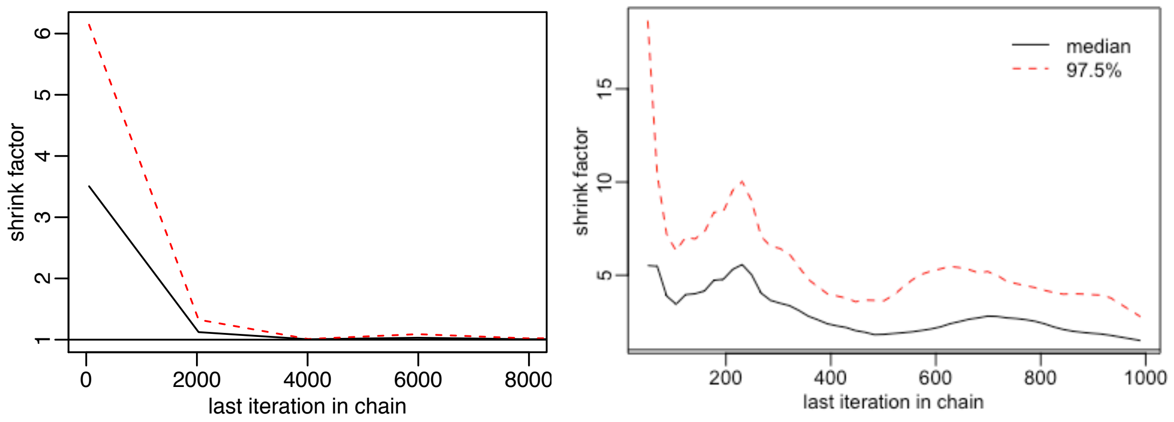

Figure 3: The potential scale reduction factors as a function of the last iteration in the chain. This parameter chain (left) flattens out to one and is hence converged. On the right, the parameter chain has not converged.

Analzying MCMC output

Once there is evidence that the MCMC chains are converged, the output can be analyzed to obtain information about the physical and statistical model parameters such as how confident we are about the estimates of uncertain values. For instance, you can look at the probability density of each parameter using the `density()` function. Estimates of the mean, mode, median, and credible intervals can be determined as well. The *equal-tail* credible interval can be computed using the `quantile()` function or the *highest posterior density* credible interval can be estimated using the `HPDinterval()` function. The equal-tail credible interval excludes the same percentage from each tail of the distribution. By definition, a 90% equal-tail credible interval would exclude 5% from the left and right tail so it would be calculated with the 5% estimate and the 95% estimate. The highest posterior density credible interval differs by estimating the shortest interval in the parameter space containing the specified percentage (e.g. 90%) of the posterior probability.

```r
# Calculate the 90% highest posterior density CI.
# HPDinterval() requires an mcmc object; this was done in the code block above.
hpdi = HPDinterval(mcmc1, prob = 0.90)

# Create density plot of each parameter.
par(mfrow = c(2,2))
for(i in 1:3){
  # Create density plot.
  p.dens = density(mcmc.chains[,i])
  plot(p.dens, xlab = paste('Parameter =',' ', parnames[i], sep = ''), main="")

  # Add mean estimate.
  abline(v = bm(mcmc.chains[,i])$est, lwd = 2)

  # Add 90% equal-tail CI.
  CI = quantile(mcmc.chains[,i], prob = c(0.05, 0.95))
  lines(x = CI, y = rep(0, 2), lwd = 2)
  points(x = CI, y = rep(0, 2), pch = 16)

  # Add 90% highest posterior density CI.
  lines(x = hpdi[i, ], y = rep(mean(p.dens$y), 2), lwd = 2, col = "red")
  points(x = hpdi[i, ], y = rep(mean(p.dens$y), 2), pch = 16, col = "red")
}
```

Exercise

Before you proceed, **make sure that you have a working version of the code blocks above.** Open your script in RStudio and `source()` it. Examine the convergence tests. You should see that the chains are visually poorly mixed and the Gelman and Rubin diagnostic failed, but the Heidelberger and Welch diagnostic passed. If not, check your script against the instructions above. Once you are satisfied that your script is working properly, save the file.

Part 1: Converging the MCMC results. Modify your new script by changing the number of iterations *NI* in block 5. The goal is to get the chains to converge. Each time you modify the number of iterations save each chain. Remember to store the parameter results (the chains) under different names each time you change the iteration number so the values are not rewritten, perhaps *chains.1thous, chains.10thous, chains.100thous*, and so on. Repeat this process until you have evidence that the MCMC chains are converged.

Your script should generate the following plots:

1. a three-panel figure with a trace plot of each parameter showing that the chains are well-mixed and converged.
2. a three-panel figure of the potential scale reduction factor for each parameter showing that the chains have converged.
3. plots of the posterior probability densities: density plots of the α, β, and σ values. Each plot should include the true parameter value from the original data as a vertical line and the densities produced

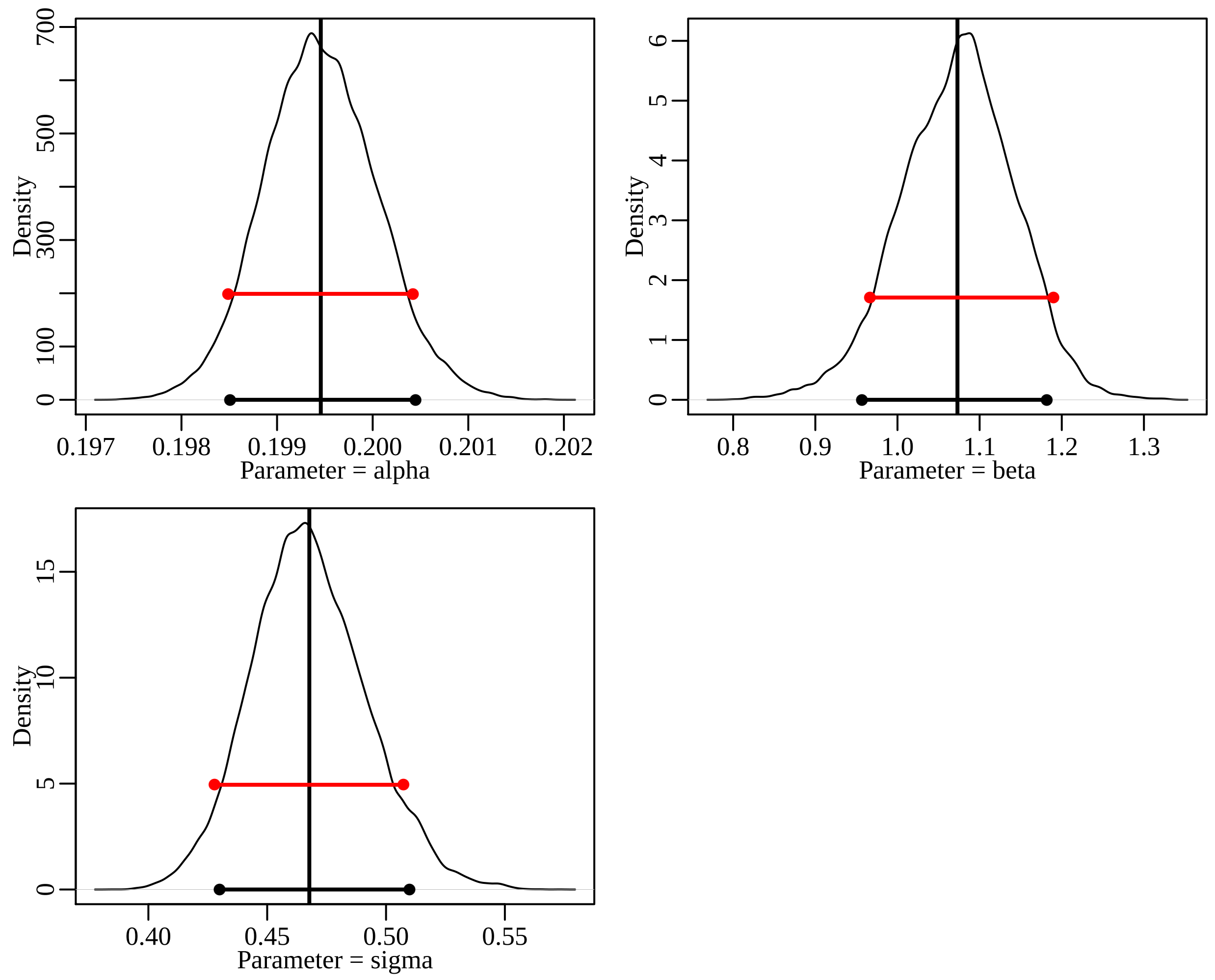

Figure 4: Probability density plot of each parameter along with the mean as a vertical line, the 90% equal-tail CI as a black horizontal line, and the 90% highest posterior density CI as a red horizontal line.

from each time you changed the iteration number. Make sure to label the axes of your plots in a sensible way and include a legend.

Part 2: Varying length of data. Suppose you did not have 200 years worth of data and instead only had 20 years of data. Modify your new script by changing *num.of.obs* to 20, but keep the iteration number at the number when the results converged with 200 data points. If the results are not converged, then increase the number of iterations used for the calibration. Once satisfied, store the parameter results. Repeat this process with 50 and 100 data points and store those results.

Your script should generate the following plots:

1. plots of the posterior probability density functions of the parameter values. Each plot should include the true parameter value from the original data as a vertical line and the densities produced from each time you changed the number of data points. Make sure to label the axes of your plots and include a legend.

Questions

1. What is the minimum chain length needed in order for the MCMC calibration in this problem to converge (to the nearest 100 thousand)?
2. Compare the mean and median from each converged chain. Are the estimates close to the true parameters ($\alpha = 0.02$, $\beta = 1$, $\sigma = 0.5$)?
3. Compare the resulting parameter densities produced from different numbers of iterations. Does it matter if the calibration has converged and why?
4. How do the results change by having less data? Does the number of iterations to run until convergence change? Does the uncertainty surrounding each parameter change?
5. When you first source the code above both the trace plots and Gelman and Rubin diagnostic fail, yet the Heidelberger and Welch diagnostic suggest the chains have converged. Why does this happen and how would you rectify this issue?

Appendix

These questions are intended for students with advanced backgrounds in Statistics or the Earth Sciences and R programming.

In the tutorial, you used uniform priors. In other applications non-uniform priors maybe more appropriate. How would one go about setting up a non-uniform prior?

When you incorporate uncertainty, the range is called a confidence interval in frequentist statistics and is a credible interval in Bayesian statistics. The difference between a frequentist 90% (5-95%) confidence interval and a Bayesian 90% (5-95%) credible interval is that the confidence interval states if you do this many times, the estimate will be in there 90% of the time, whereas the Bayesian credible interval states that the physical model, data, and the statistical approach indicate that there is a 90% chance of capturing the true value. Does the credible interval capture the true values?

In this chapter, you fit a linear model to data that have non-correlated (independent and identically distributed) residuals. In the Earth sciences, the data being evaluated may have correlated residuals. How could one test whether the residuals are correlated? If the data have correlated residuals, how might one modify the MCMC calibration to account for this correlation?

References

Bayes, T. 1764. An essay toward solving a problem in the doctrine of chances. Philosophical Transactions of the Royal Society on London 53, 370-418. Available online at http://rstl.royalsocietypublishing.org/content/

53/370.full.pdf+html

Brooks, S. P. and Gelman, A. 1998. General methods for monitoring convergence of iterative simulations. Journal of Computational and Graphical Statistics, 7, 434-455. Available online at http://www.tandfonline.com/doi/abs/10.1080/10618600.1998.10474787

Flegal, J. M., Haran, M., and Jones, G. L. 2008. Markov chain Monte Carlo: Can we trust the third significant figure? Statistical Science, 23, 250-260. Available online at http://sites.stat.psu.edu/~mharan/mcse_rev2.pdf

Gelman, A. and Rubin, D. B. 1992. Inference from iterative simulation using multiple sequences. Statistical Science, 7, 457-511. Available online at https://projecteuclid.org/euclid.ss/1177011136

Gilks, W. R. 1997. Markov chain Monte Carlo in practice. Chapman & Hall/CRC Press, London, UK

Grinstead, C. M. and Snell, J. L. 2006. Introduction to Probability. American Mathematical Society, 2006. Available online at http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/amsbook.mac.pdf

Hastings, W. K. 1970. Monte Carlo sampling methods using Markov chains and their applications. Biometrika 57(1):97–109. doi:10.1093/biomet/57.1.97. Avialable online at https://www.jstor.org/stable/2334940?seq=1#page_scan_tab_contents

Heidelberger, P. and Welch, P. D. 1981. A spectral method for confidence interval generation and run length control in simulations. Comm. ACM. 24, 233-245. Available online at http://dl.acm.org/citation.cfm?id=358630

Heidelberger, P. and Welch, P. D. 1983. Simulation run length control in the presence of an initial transient. Opns Res. 31, 1109-1144. Available online at http://dl.acm.org/citation.cfm?id=2771121

Howard, R. A. 2007. Dynamic Probabilistic Systems, Volume I: Markov Models. Dover Publications, Inc., Mineola, NY. Available online at https://books.google.com/books?id=DU06AwAAQBAJ

Jones, G. L., Haran, M., Caffo, B. S. and Neath, R. 2006. Fixed-width output analysis for Markov chain Monte Carlo. Journal of the American Statistical Association, 101, 1537–1547. Available online at https://www.jstor.org/stable/27639771?seq=1#page_scan_tab_contents

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. Equation of state calculations by fast computing machines. J. Chem. Phys. 21(6), 1087-1092. Available online at http://aip.scitation.org/doi/abs/10.1063/1.1699114

Roberts, G. O., Gelman, A., and Gilks W. R. 1997. Weak convergence and optimal scaling of random walk Metropolis algorithms. Ann. Appl. Prob. 7, 110–120. Available online at http://projecteuclid.org/download/pdf_1/euclid.aoap/1034625254

Rosenthal, J. S. 2010. “Optimal Proposal Distributions and Adaptive MCMC.” Handbook of Markov chain Monte Carlo. Eds., Brooks, S., Gelman, A., Jones, G. L., and Meng, X.-L. Chapman & Hall/CRC Press. Available online at https://pdfs.semanticscholar.org/3576/ee874e983908f9214318abb8ca425316c9ed.pdf

Ruckert, K. L., Guan, Y., Bakker, A. M. R., Forest, C. E., and Keller, K. The effects of non-stationary observation errors on semi-empirical sea-level projections. Climatic Change 140(3), 349-360. Available online at http://dx.doi.org/10.1007/s10584-016-1858-z

Schruben, L. W. 1982. Detecting initialization bias in simulation experiments. Opns. Res., 30, 569-590. Available online at http://dl.acm.org/citation.cfm?id=2754246