Lab #X: Bayesian Inference and Markov chain Monte Carlo Basics (Kelsey L. Ruckert, Tony E. Wong, Benjamin Seiyon Lee, Yawen Guan, and Murali Haran)

Learning objectives

After completing this exercise, you should be able to

- describe what the detailed balance equation is
- describe what are the three conditions
- explain why the Metropolis-Hastings algorithm fulfills the conditions and works

Introduction

In earlier chapters, we examined how a statistical technique called the bootstrap can be used to determine how confident we are about our estimates of uncertain values. In particular, we looked at how to determine whether a coin is fair and how sure we can be about our estimates of future sea-level change.

The bootstrap is an example of what is called a *frequentist* statistical technique. Frequentist techniques assume that the probability of an event is the proportion of the times the event will occur (if it was an infinite number of trials and a probability $0 < p < 1$, then we would see an infinite number of "successful" outcomes). For example, we know that the probability that a fair coin will come up heads on any individual flip is 0.5. In a frequentist interpretation, that probability implies that we can observe a very large number of coin flips, in which we count the number of times the coin comes up heads. That number of times, divided by the number of flips, is the *frequency* with which the coin comes up heads.

Of course, in many situations, we cannot perform lots of random trials to determine the probabilities of different outcomes. The sea-level rise problem is one example. We want to know how much sea level will rise in the future. Because the data and physical models are imperfect, we cannot be sure of the exact answer. In a frequentist framework, we would try to ascertain how sea-level rise would vary under multiple hypothetical replications of the state of the world (multiple alternative worlds). This is how we would determine the probabilities of different outcomes.

Bayesian techniques provide us with an alternative way of viewing this problem. In a Bayesian framework, we have a preexisting estimate of the probability of different outcomes that is based on our past experiences and our beliefs about the situation in question. We then make observations and update the probability estimates based on those observations. This procedure, which is called *Bayesian updating*, is perhaps similar to how people often make decisions. New information leads to changed opinions. The entire process of defining the prior probability distribution of different outcomes or physical model parameters and then using Bayesian updating to change these outcomes is called *Bayesian model calibration.*

It is important to point out that the use of the word "model" can refer to two different things, that is, the statistical model (which is Bayesian in our case) and the physical model of the system of interest (e.g., a global sea-level model). For simplicity and ease is understanding, we will use "statistical" or "physical" when referring to both kinds of models.

In Bayesian model calibration, physical model parameters are considered to be random variables. Our knowledge of the parameters (before any data are observed) is represented by a *prior* probability distribution. Observations may be used to inform estimates of which parameter values are more or less likely. The probability model for observations provides a distribution on observations for a particular parameter setting, that is, as we vary the value of the parameters, the probability distribution changes. To fix ideas, think of the mean and variance parameters of a normal distribution — as we vary the values of the mean and the variance, the normal distribution for the observation changes. The probability model therefore provides a probability distribution for random variables for a particular parameter value. A *likelihood function* helps solve the inverse problem — it is useful for providing information about the parameters *given the observations*. It is obtained from the probability distribution by plugging in the observations into the function. The likelihood function is therefore a function of just the parameters — this includes both the statistical and physical model parameters. Direct sampling from the posterior distribution is typically impossible because the posterior is only known up to a constant, and in many cases this distribution is intractable. In order to obtain samples from the posterior, one approach is to employ a *Markov chain Monte Carlo* (MCMC) sampling technique.

MCMC is an algorithm used to simulate random variables or "draw samples" from a given probability distribution by constructing a *Markov chain* based on the distribution. For Bayesian inference, the distribution of interest is the posterior distribution. A Markov chain is a sequence of random variables where each successive value in the sequence depends on the current value of the sequence. The *Metropolis-Hastings* algorithm is used to construct the Markov chain so its "stationary distribution" is the distribution of interest. What this means is that the Markov chain satisfies a theoretical property which allows us to use sample means of the Markov chain to approximate the mean of the posterior distribution. For instance, if we want to approximate the mean of the posterior distribution, we just need to take an average of the Markov chain samples. If we want to approximate the correlation between two random variables in a joint distribution, we can take the sample correlation between the two samples in the Markov chain.

MCMC's popularity and prominence are due to its generality as it may be used to draw samples from high-dimensional, complicated probability distributions for which sampling algorithms may not generally exist. MCMC is useful where the goal is to use posterior distributions of parameters in a physical model to summarize the information about the physical model parameters that are contained in a given data set. Because posterior distributions tend to be complicated, MCMC is often one of the few generally applicable approaches that may be used to approximate various characteristics of the posterior distribution. Once samples from the posterior distribution are generated, one can calculate "best" ("point") estimates for the parameters by using attributes like posterior means and also represent uncertainties for the parameters of interest by using "credible intervals" (Bayesian analogues to frequentist confidence intervals) based on the samples. Bayesian methods are useful in climate science because they can easily integrate multiple sources of information (e.g., physical models, multiple data sets, and complex sources of error) into a single framework. MCMC then provides a standard algorithm for approximating the resulting posterior distribution, thereby conveniently incorporating multiple sources of uncertainty, say from various data and assumptions, when providing conclusions about the model (physical and statistical) parameters.

MCMC is a complex concept, so we broke up the section on MCMC into three chapters. Each chapter will build off of the previous one. In this chapter, the goal is to describe how MCMC works and why it works. To do this you will code your own MCMC with the Metropolis-Hastings algorithm. The overall goal of these three chapters are to:

1. understand the basics of MCMC
2. understand the output and what MCMC convergence means
3. how to apply and problem solve applying MCMC to a "real" model and "real" data

Why does MCMC work?

Here we briefly describe some of the basic concepts of MCMC and why it works. For more details, we recommend consulting chapter 6 from Wood (2015), the YouTube video called, "(ML 18.6) Detailed balance (a.k.a. Reversibility)", Detailed balance (2017), and Gilks (1997). This section was adapted from those references.

Detailed balance equation

To understand how and why MCMC works, one must learn about *detailed balance equations*. Detailed balance or reversibility is when a process—say the probability density distribution π on some set of states Y—satisfies the detailed balance equations with respect to a *transition probability* distribution T. The transition probability is the probability of changing from one parameter value to another value in a single move. If π satisfies detailed balance with respect to T, then it implies π has a stationary distribution such that

$$\pi(x)T(x,y) = \pi(y)T(y,x),$$

where $T(x,y)$ is the transition probability of moving from state x (current value) to state y (proposed value), and $\pi(x)$ and $\pi(y)$ are the equilibrium probabilities of being in states x and y, respectively. In other words, for all y,

$$\pi(y) = \sum^{x} \pi(x)T(x,y) = \sum^{x} \pi(y)T(y,x) = \pi(y)\sum^{x} T(y,x) = \pi(y).$$

Additionally, detailed balance also implies that the process must be reversible. In other words, the Markov chain looks the same moving forward as it does moving backwards. For instance, a Markov chain, MC, satisfies detailed balance when, $(Y_0, ..., Y_n) \sim MC(\pi, T) \Rightarrow (Y_n, ..., Y_0) \sim MC(\pi, T)$. Therefore, detailed balance implies that there is no net flow of probability such that

$$T(x,y)T(y,z)T(z,x) = T(x,z)T(z,y)T(y,x).$$

Metropolis-Hastings Algorithm

Next, we show how the Metropolis-Hastings algorithm satisfies the detailed balance equation. Given that we have two values of the Markov chain (x and y), $k(x,y)$ is the *transition kernel* of moving from x to y, and $k(y,x)$ is the transition kernel of moving from y to x, we want to show that

$$(1) \qquad \pi(x)k(x,y) = \pi(y)k(y,x).$$

In the Metropolis-Hastings algorithm, the transition kernel, $k(y,x)$ of moving from y (current value) to x (proposed value) is defined as

$$(2) \qquad k(y,x) = \alpha(y,x)q(y,x),$$

where $q(x,y)$ is the proposal distribution and $\alpha(y,x)$ is the acceptance probability for the proposal. The acceptance probability is defined as

$$(3) \qquad \alpha(y,x) = min\left\{1, \frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right\}.$$

In equation (1), if $x = y$, then the detailed balance equation is satisfied. If $x \neq y$, then the left hand side of the equation (LHS) gives us

$$(4) \qquad \pi(x)k(x,y) = \pi(x)q(x,y)\alpha(x,y).$$

Without loss of generality, we assume that $\pi(y)q(y,x) > \pi(x)q(x,y)$. Then, by equation (3), the LHS becomes:

$$(5) \qquad \pi(x)k(x,y) = \pi(x)q(x,y)\alpha(x,y) = \pi(x)q(x,y) \times 1.$$

The right hand side (RHS) of equation (1) becomes

$$\pi(y)k(y,x) = \pi(y)q(y,x)\alpha(y,x) = \pi(y)q(y,x) \times \frac{\pi(x)q(x,y)}{\pi(y)q(y,x)} = \pi(x)q(x,y) = \text{LHS}.$$

Hence, the Markov chain defined by the all-at-once Metropolis-Hastings algorithm satisfies the detailed balance equation with respect to the stationary distribution π.

How does MCMC work?

A Markov chain is determined by three key elements: the *parameter space*, the initial distribution, and the transition probability distribution. Parameter space is the set of all possible parameter value combinations that satisfy the specified constraints. For MCMC, where Monte Carlo simply means random samples, we generate a random parameter value from the initial distribution, then move to the next parameter value iteratively based on the transition probability. Under certain conditions, the draws will eventually reach an equilibrium probability distribution. That is, statistical properties of the distribution do not change as new random values are generated, thus showing evidence of *convergence*.

For example, suppose we are interested in the weather (state) on any day (this example is adapted from Example 11.1 in Grinstead and Snell (2006)). Suppose the weather can be either rainy, foggy, or sunny; thus, the parameter space contains three possible values, “rainy”, “foggy”, and “sunny”. Based on past experience, we know in this example that there are never two sunny days in a row and only half of the time a sunny day will occur after a foggy or rainy day. We also know there is an even chance of having two foggy days in a row and two rainy days in a row. This information can be represented as a *transition matrix*,

$$P = \begin{array}{c c} & \begin{array}{ccc} \text{FOGGY} & \text{SUNNY} & \text{RAINY} \end{array} \\ \begin{array}{c} \text{FOGGY} \\ \text{SUNNY} \\ \text{RAINY} \end{array} & \left[\begin{array}{ccc} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{array} \right] \end{array},$$

where each row represents the current state of the weather; a foggy day (first row), sunny day (second row), and rainy day (third row). The columns represent the next state of the weather; a foggy day (first column), sunny day (second column), and rainy day (third column). For example, if today is foggy, then the probability of the event that tomorrow is also foggy is $P_{1,1}$ or 0.5. Note that given the state of the weather today, the sum of the probabilities of the parameter values tomorrow must necessarily equal one.

We are going to formulate this example in R. The transition matrix can be set up by first creating vectors of the transition probabilities followed by using the matrix command. Typing `help(matrix)` in the Console window will describe the arguments of the function.

```r
# Set up the known probabilities based on prior knowledge.
# Define the transition matrix.
# Rows represent the current state.
# Columns are the next state.
#        FOGGY   SUNNY  RAINY
FOGGY <- c( 0.5, 0.25,  0.25)
SUNNY <- c( 0.5,    0,   0.5)
RAINY <- c(0.25, 0.25,   0.5)

# Define the parameter space.
parameter <- c("Foggy", "Sunny", "Rainy")

P <- matrix(c(FOGGY, SUNNY, RAINY), nrow=3, ncol=3, byrow = TRUE,
            dimnames = list(parameter, parameter))
print(P)
```

Using what we already know, we can answer questions about weather in the future via calculation. For example, suppose today is foggy. What then is the probability that the weather will be rainy two days from now? One way is for tomorrow to be foggy, and two days from now to be rainy. Since today is foggy, the probability that tomorrow is also foggy is 0.5. And if tomorrow is foggy, then the probability that two days from now is rainy is 0.25. Thus, the probability of it being foggy then rainy is the product of these probabilities (0.5 x 0.25 = 0.125). A different way is for tomorrow to be rainy (a 0.25 probability) and then rainy two days from now (a 0.5 probability), which we can determine to have a 0.125 probability of occurring based on the product of the probabilities. The last way is for tomorrow to be sunny followed by a rainy day. Since today is foggy, the probability of tomorrow being sunny is 0.25. If tomorrow is sunny, then the probability that it will be rainy two days from now is 0.50. Hence, if today is foggy, then the probability of it being sunny then rainy is the product (0.25 x 0.5 = 0.125). These three events are independent (tomorrow cannot be foggy, rainy, and sunny all at the same time), so combining these three probabilities gives a total probability of 0.125 + 0.125 + 0.125 = 0.375 that it will be rainy two days from now. Thus, there is a 0.375 probability that the weather will be rainy two days from now given today is foggy.

The information that we already know can answer more than just the probability that the weather will be rainy two days from now given today is foggy. Similar to the example above, we could determine the probability that it will be foggy or sunny two days from now given it is foggy today. We could also determine the same weather probabilities given today was sunny or rainy instead of foggy. Additionally, this information could be used to forecast further than two days into the future. Using the steps above, try to calculate the probability that it will be foggy or sunny three days from now given that it is foggy today. You should calculate a 0.41 probability that it will be foggy and a 0.20 probability that it will be sunny three days from now.

If you continue to forecast the weather further into the future, eventually your probabilities will remain the same no matter what the weather is like today. In this example, the probabilities for the three types of weather are; Probability(rainy) = 0.4, Probability(sunny) = 0.2, and Probability(foggy) = 0.4. Once the probabilities remain the same no matter the initial weather, the system has reached an equilibrium probability distribution of the parameters. The code block below runs this process.

In the code block below, the command `mat.or.vec()` produces a vector or matrix with a number of rows equal to the first argument and a number of columns equal to the second argument. Initially, all of the elements of this new vector or matrix have the value 0. Based on the code blocks below, what should be the dimensions of `CurrentStateProb`? Confirm your guess using the command `dim(CurrentStateProb)` after running the code block.

```r
# PREDICTING THE WEATHER WITH A MARKOV CHAIN EXAMPLE
#==================================================
# Set the prediction length and the initial parameter value.
Prediction_Days <- 25
CurrentStateProb <- mat.or.vec(Prediction_Days, length(P[1,]))
CurrentStateProb[1,] <- c(1, 0, 0) # Today is foggy

# Run the Markov chain.
for(i in 2:Prediction_Days){
  # Current parameter value times probability
  Prob <- CurrentStateProb[i-1, ] * P
  CurrentStateProb[i, ] <- c(sum(Prob[,1]), sum(Prob[,2]), sum(Prob[,3]))
}
colnames(CurrentStateProb) <- parameter
print(CurrentStateProb)

# Print weather predictions.
print(paste("P(", parameter, ") in 1 day = ", round(CurrentStateProb[2, ], 2)))
print(paste("P(", parameter, ") in 5 days = ", round(CurrentStateProb[6, ], 2)))
print(paste("P(", parameter, ") in 24 days = ", round(CurrentStateProb[25, ], 2)))
```

Alternatively, we can achieve the same goal via MCMC. We may draw today's weather using the probabilities for the various kinds of weather following a foggy day. The initial value is not too important at this point because we typically include a *burn-in period* to remove the effects of starting values. The burn-in period is the process of throwing away some initial portion of the Markov chain so as to remove dependence of the results on the initial conditions. Then we iteratively draw the weather of next day based on the transition probability. After a large enough number of draws, the distribution of the samples will eventually converge to the equilibrium distribution; Probability(rainy) = 0.4, Probability(sunny) = 0.2, and Probability(foggy) = 0.4. The code block below runs this process and displays the output from both the Markov chain and MCMC. Note that the difference is subtle. The difference stems from the fact that the Markov chain calculates the exact probability while the MCMC simulates the probability.

```r
# SAMPLING THE PROBABILITY DISTRIBUTION; A MARKOV CHAIN MONTE CARLO EXAMPLE
#=================================================
# Set the initial parameter value probability.
CurrentState <- sample(parameter, 1, prob = c(0.5, 0.25, 0.25)) # Today is foggy

# Start sampling (iteratively) from the distribution.
weather <- c()
for (i in 1:1e4){
  NextState <- sample(parameter, 1, prob = P[CurrentState,])
  weather[i] <- NextState
  CurrentState <- NextState
}
# Throw away the first 1% of the data (Burn-in)
burnin <- seq(from = 1, to = 1e4*0.1, by = 1)
weatherDraws <- weather[-burnin]

# DISPLAY
#=================================================
par(mfrow = c(1,2))
# Display the results from the Markov Chain
plot(1:Prediction_Days, CurrentStateProb[ ,1], xlab = "Days", ylab = "P(Weather)",
     ylim = c(0,1), type = "l", lwd = 2, col = "gray")
lines(1:Prediction_Days, CurrentStateProb[ ,2], lwd = 2, col = "gold")
lines(1:Prediction_Days, CurrentStateProb[ ,3], lwd = 2, col = "blue")
legend("right", c("Foggy", "Sunny", "Rainy"), lwd = 2, bty = "n", col = c("gray", "gold",
                                                                        "blue"))

# Display the results from MCMC
barplot(prop.table(table(weatherDraws)), ylim = c(0,1),
        sub = "Equilibrium distribution\nof the weather", col = c("gray", "blue", "gold"))
abline(h = 0.4, lty = 2); abline(h = 0.2, lty = 2)
```

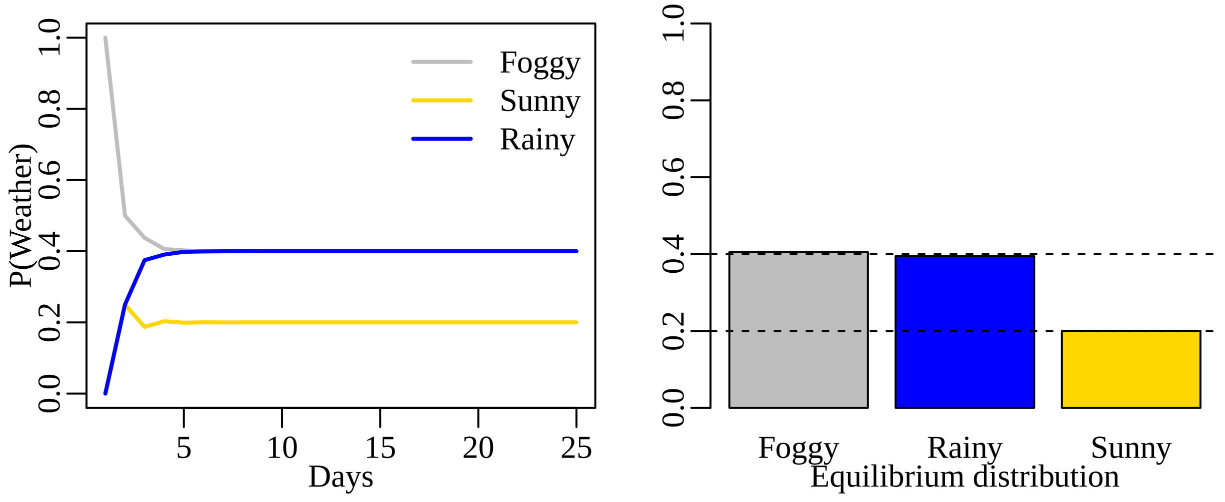

Figure 1: The probability of the weather (rainy, sunny, or foggy) predicted over time (Markov chain process; left) and the equilibrium distribution (MCMC process; right).

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. The tutorial example will show exactly what is coded within MCMC by having the user code MCMC rather than use a MCMC package. This code has been adapted from its original lay out in a class taught by Dr. Klaus Keller (specific details are given with labX_sample.R).

In a real world application, we would have some observations of a system. In this tutorial, we will instead fit a two parameter model to some pseudo observations that we generate. We choose this approach to reduce the complexity of learning and understanding MCMC. It is also useful in understanding the assumptions made according to the structure of the observations. In the following chapters on MCMC, we will increase the complexity, so in the end you will be able to calibrate a model to observations using MCMC.

The following approach and assumption is taken to approximate observations,

$$\begin{aligned}
\overbrace{y_t}^{\text{obsevations}} &= \overbrace{f(\theta,t)}^{\text{physical model}} + \overbrace{\epsilon_t}^{\text{measurement errors}}, \\
f(\theta,t) &= \alpha \times t + \beta, \\
\epsilon_t &\sim N(0,\sigma^2), \\
\underbrace{\theta}_{\text{parameters}} &= (\alpha,\beta),
\end{aligned}$$

where θ denotes the unknown slope (α) and intercept (β) parameters, and $f(\theta,t)$ denote the model output at parameter setting θ and time t. The observed data, y_t, is then equal to the physical model output plus measurement errors ϵ_t.

```r
# Clear away any existing variables or figures and set the seed for random sampling.
rm(list = ls())
graphics.off()
set.seed(1234)

# Define the true model parameters and set up the time scale.
alpha.true <- 2 # Arbitrary choices.
beta.true <- -5
time <- 1:10

# Generate some observations with noise (measurement error) and the model.
y.true = alpha.true*time + beta.true
sigma = 1
meas_err <- rnorm(length(time), mean = 0, sd = sigma)
y = y.true + meas_err

# Plot the true model and observations.
par(mfrow = c(1,1))
plot(time, y.true, type="l", main="True observations and model",
     xlab = "Time", ylab = "Observations")
points(time, y, pch = 20)
```

Define the log posterior

In Bayesian inference, we approximate the *posterior probability* by defining the *prior probability* and the *likelihood function*. The posterior probability is the probability that an event or observation will occur **after** taking into account all evidence and background information. The prior probability differs because it is the probability that an event or observation will occur **before** taking into account new evidence. It is based on background information. We take into account new evidence using the likelihood function. The likelihood function describes the plausibility of a parameter value based on observations. Generally, we seek parameter values that maximize the likelihood function, in light of the uncertainties in both the parameters and the observations. Bayes' theorem defines the posterior probability as proportional to the likelihood of the observations given the parameters times the prior probability of the parameters (Bayes, 1764),

$$\overbrace{P(parameters \mid observations)}^{\text{Posterior}} \propto \overbrace{L(observations \mid parameters)}^{\text{Likelihood}} \times \overbrace{P(parameters)}^{\text{Prior}}.$$

The posterior distribution therefore summarizes information about the parameters based on the prior distribution and what the likelihood function says about more "likely" parameter values. The posterior therefore provides a range of parameter values, and says which values are more probable than others.

In this analysis, you will work with the log-probability distributions for numerical stability reasons. That is, the probabilities involved may be very small, and computers may not be able to distinguish them from 0 in many cases.

```r
# Define the unnormalized log posterior.
logp = function(theta){
  N = length(time)

  # Calulate model simulations.
  alpha = theta[1]; beta = theta[2]
  model = alpha*time + beta

  # Estimate the residuals (i.e., the deviation of the observations from the
  # model simulation).
  resid =   y - model

  # Get the log of the likelihood function.
  log.likelihood = -N/2*log(2*pi) - N*log(sigma) - 1/2*sum(resid^2)/sigma^2

  # Use an improper uniform uninformative prior.
  log.prior = 0 # log(1)

  # Bayesian updating: update the probability estimates based on the observations.
  log.posterior = log.likelihood + log.prior

  # Return the unnormalized log posterior value.
  return(log.posterior)
}
```

Code example with Metropolis-Hastings

The algorithm used for MCMC in this tutorial is called Metropolis-Hastings where the method implemented here uses a *random walk*. A random walk is a wandering movement of an object away from where it started, following no recognizable pattern. The Metropolis-Hastings algorithm randomly samples the probability distribution and creates a Markov chain, in which the next step in the random walk only depends on the current state, and is determined by the transition probabilities and the *step size* (the proposed deviation between the current state and the next). The step size is important because if the jumps are "too small", then the Markov chain explores the parameter space at a slower rate with a high *acceptance rate* and hence will increase the number of iterations needed for convergence. If the jumps are "too large", then the chain could get stuck in places and possibly reject many values. The acceptance rate is the percentage of candidates that were accepted or the ratio of the number of unique values in the Markov chain to the total number of values in the Markov chain.

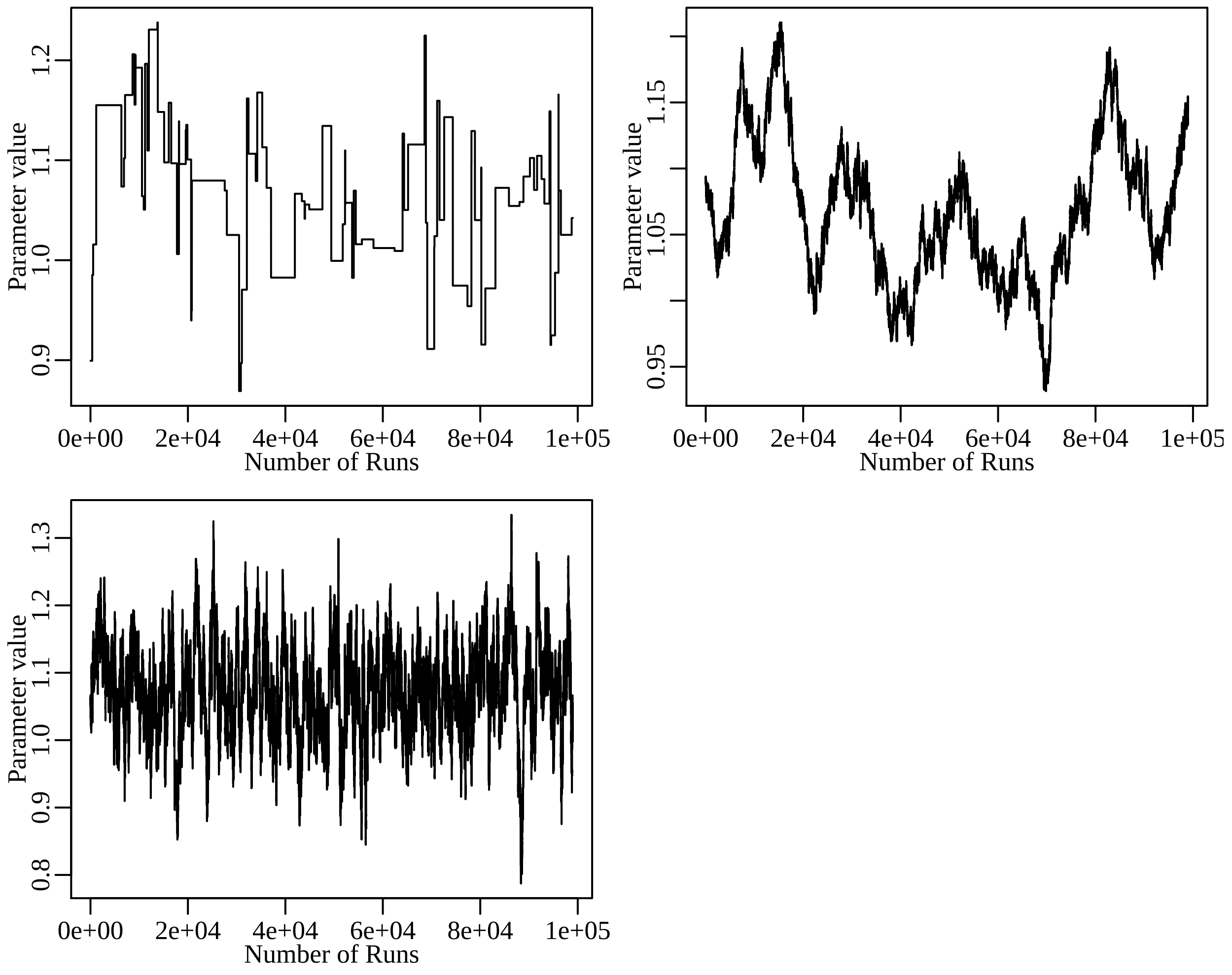

Figure 2: The chain of a single parameter that is poorly mixed with a small acceptance rate and a large step size (top left), poorly mixed with a large acceptance rate and small step size (top right), and well mixed and converged with a 'good' acceptance rate and step size (bottom left).

The acceptance rate is not to be confused with the *acceptance ratio*, which is used to decide whether to accept or reject a proposed value and is how the Markov chain is constructed. In general, the step size should be chosen so that the acceptance rate is far from 0 and far from 1 (Rosenthal, 2010). The optimal acceptance rate varies depending on the number of dimensions in the parameter space. As the number of dimensions in the parameter space increases, the optimal acceptance rate goes to about 23.4% (Roberts et al., 1997; Rosenthal, 2010). But for a single parameter, or if you implement a Metropolis-within-Gibbs (or similar) sampling approach, it is about 44% (Rosenthal, 2010). For 2 parameters, it is likely higher than 23.4%. Note that while the step size is set in this tutorial, there are adaptive algorithms for the MCMC sampler to "learn" what the step sizes ought to be, in order to achieve a desired acceptance rate. For example, there is a package called `adaptMCMC` where the `MCMC` function has arguments to specify a desired acceptance rate as well as the option to adapt the Metropolis sampler to achieve this desired acceptance rate (Vihola, 2011).

Constructing a Markov Chain

To construct the chains, you need to decide how long (how many iterations) to run MCMC and start with some initial parameter values, $\theta^{initial}$. In the example below, the number of iterations is set to 30,000 and the initial parameter values is randomly generated. In a "real" situation you will likely have more informed parameter values either based on expert opinion or based on an optimization algorithm.

```r
# Set the number of MCMC iterations.
NI = 30000

# Start with some initial state of parameter estimates, $theta^initial$
alpha.init = runif(1, -50, 50) # Arbitrary choices.
beta.init = runif(1, -50, 50)
theta = c(alpha.init, beta.init)
```

Using the initial parameter values $\theta^{initial}$, you then evaluate the unnormalized posterior of that value, $p(\theta^{initial} \mid y)$, using the likelihood function $(L(y \mid \theta))$ and the prior distribution.

```r
# Evaluate the unnormalized posterior of the parameter values
# P(theta^initial | y)
lp = logp(theta)
```

From here, a new parameter value, θ^{new} is proposed by being randomly drawn based on the current parameter value, transition probability $(p(\theta^{new} \mid \theta^{initial}))$, and *step size* (the proposed deviation between the current state and the next). You then evaluate the new unnormalized posterior of that value, $p(\theta^{new} \mid y)$, using the likelihood function. Comparing the new value to the old one, the code evaluates whether or not to accept the new value. You accept the new value with a probability based on the ratio of $p(\theta^{new} \mid y)/p(\theta^{initial} \mid y)$. To put this in simpler terms, say you are at an arbitrary parameter value within the prior probability distribution. You then pick a new randomly selected parameter value. If this proposed new parameter value is "better" (higher posterior probability) than the current state, then you accept with a probability of 1; if the proposal is worse, you accept with some probability less than 1. Keeping a list of the parameter values in the model simulation will create a vector of possible values that is the Markov chain.

```r
# Setup some variables and arrays to keep track of:
theta.best = theta         # the best parameter estimates
lp.max = lp                # the maximum of the log posterior
theta.new = rep(NA,2)      # proposed new parameters (theta^new)
accepts = 0                # how many times the proposed new parameters are accepted
mcmc.chains = array(dim=c(NI,2)) # and a chain of accepted parameters

# Set the step size for the MCMC.
step = c(0.1, 1)

# Metropolis-Hastings algorithm MCMC; the proposal distribution proposes the next
# point to which the random walk might move. For this algorithm, this proposal
# distribution is symmetric, that is P(x to x`) = P(x` to x).
for(i in 1:NI) {
  # Propose a new state (theta^new) based on the current parameter values
  # theta and the transition probability / step size
  theta.new = rnorm(2, theta, sd = step)

  # Evaluate the new unnormalized posterior of the parameter values
  # and compare the proposed value to the current state
  lp.new = logp(theta.new)
  lq = lp.new - lp

  # Metropolis test; compute the acceptance ratio
  # Draw some uniformly distributed random number 'lr' from [0,1];
  lr = log(runif(1))

  # If lr < the new proposed value, then accept the parameters setting the
  # proposed new theta (theta^new) to the current state (theta).
  if(lr < lq) {
    # Update the current theta and log posterior to the new state.
    theta = theta.new
    lp = lp.new

    # If this proposed new parameter value is "better" (higher posterior probability)
    # than the current state, then accept with a probability of 1. Hence, increase
    # the number of acceptions by 1.
    accepts = accepts + 1

    # Check if the current state is the best, thus far and save if so.
    if(lp > lp.max) {
      theta.best = theta
      lp.max = lp
    }
  }
  # Append the parameter estimate to the chain. This will generate a series of parameter
  # values (theta_0, theta_1, ...).
  mcmc.chains[i,] = theta
}
```

Checking the acceptance rate

After running MCMC, check the acceptance rate to determine whether the calibration appropriately explored the parameter space. That is, the chain didn't reject or accept too many values. For two parameters, the acceptance rate should be higher than 23.4%.

```
# Calculate the parameter acceptance rate; it should be higher than 23.4%.
accept.rate <- (accepts/NI) * 100
print(accept.rate)
```

Exercise

Save labX_sample.R to a new file. Now, using what you've learned throughout the previous chapters, modify the new file so that it produces a .pdf file with 6 panels that plots the results from the MCMC Metropolis-Hastings algorithm. In the following chapter, we will go in depth in analyzing the output. Specifically, you'll need to plot:

1. the true observations, true model fit, and estimated best fit over time
2. a joint scatter plot of the alpha chain `chain[ ,1]` versus the beta chain `chain[ ,2]`
3. the alpha chain and the beta chain versus the iteration number
4. histograms of the the alpha chain and the beta chain

Make sure that your plots have descriptive labels for the axes and a legend if necessary.

Questions

1. What is the purpose of defining a likelihood function?
2. In the exercise, you plotted the parameter chains, what do these chains mean/show?
3. Explain in your own words what the detailed balance equation is and why MCMC—specifically the Metropolis-Hasting algorithm—works?

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

Detailed balance. 2017. In Wikipedia, The Free Encyclopedia. Available online at https://en.wikipedia.org/w/index.php?title=Detailed_balance&oldid=810481659

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

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

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

(ML 18.6) Detailed balance (a.k.a. Reversibility). *YouTube*. Posted by mathematicalmonk, July 25, 2011. Available online at https://www.youtube.com/watch?v=xxDkdwQdGvs

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

Vihola, M. 2011. Robust adaptive Metropolis algorithm with coerced acceptance rate. Statistics and Computing. Available online at http://www.springerlink.com/content/672270222w79h431/

Wood, S. N. 2015. Core Statistics, Volume 6 of Institute of Mathematical Statistics Textbooks. Cambridge University Press. available online at https://people.maths.bris.ac.uk/~sw15190/core-statistics.pdf