# Contents

# Summary

This e-textbook presents a series of readings and laboratory exercises in R and Python that introduce advanced aspects of risk analysis and decision support. These treatments are intended for graduate students and practitioners.

## Contributor Bios

**Vivek Srikrishnan** is an Assistant Research Professor at Penn State. He is interested in how communities and regions manage climate risks. This research focuses on decision-making about climate change adaptation and mitigation under coupled social, economic, and Earth system uncertainties. Vivek received his Ph.D and master's from Penn State in energy and mineral engineering.

**Klaus Keller** is a Professor of Geosciences at Penn State, where he directs the Center for Climate Risk Management as well as the research network for Sustainable Climate Risk Management (http://scrimhub.org). Before joining Penn State, he worked as a research scientist and lecturer at Princeton University and as an engineer in Germany. Professor Keller graduated from Princeton with a Ph.D. in civil and environmental engineering. He received master's degrees from M.I.T. and Princeton as well as an engineer's degree from the Technische Universität Berlin. His research addresses two interrelated questions. First, how can we mechanistically understand past and potentially predict future changes in the climate system? Second, how can we use this information to design sustainable, scientifically sound, technologically feasible, economically efficient, and ethically defensible climate risk management strategies? He analyzes these questions by mission-oriented basic research covering a wide range of disciplines such as Earth system science, economics, engineering, philosophy, decision science, and statistics. Klaus' e-mail address is klaus@psu.edu, and his Web site is at http://www3.geosc.psu.edu/~kzk10/.

# Acknowledgements

This work was supported by the National Science Foundation through the Network for Sustainable Climate Risk Management (SCRiM) under NSF cooperative agreement GEO-1240507. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Other support was provided by the Center for Climate Risk Management and the Rock Ethics Institute.

Katerina Kostadinova designed the cover using a photograph provided by Frank Erickson.

Randy Miller created scripts for merging the parts of the book into a single `.pdf` file and created the Github repository.

**Vivek Srikrishnan** thanks his colleagues and the participants in the 2016 SCRiM Summer School for helping to shape the scope of this book.

**Klaus Keller** thanks all the students from the Risk Analysis course, the members of the Keller research group, and his SCRiM collaborators for invaluable inputs. Special thanks go to his family, many mentors, and friends for support.

## Introduction (Vivek Srikrishnan and Klaus Keller)

## Lab #1: Values and Mental Models in Decision Support (Casey Helgeson)

# Lab #2: 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 update our beliefs about the 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 (i.e., using the least squares error in the likelihood function). 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. Credible intervals can be estimated with "quantiles". In the next chapter, this process of estimating the crebiel intervals will be described. 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 $\pi$ 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 $\pi$ satisfies detailed balance with respect to $T$, then this implies $\pi$ 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 $y$ to $x$ 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 $\pi$.

## 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 (i.e., “forget the initial conditions”), 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 or in other words to forget initial conditions. This initial portion of the

Markov chain is thrown out because it is still dependent 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)
```

## 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 a class example 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 increase transparency of MCMC. It is also useful in understanding the assumptions made regarding 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.

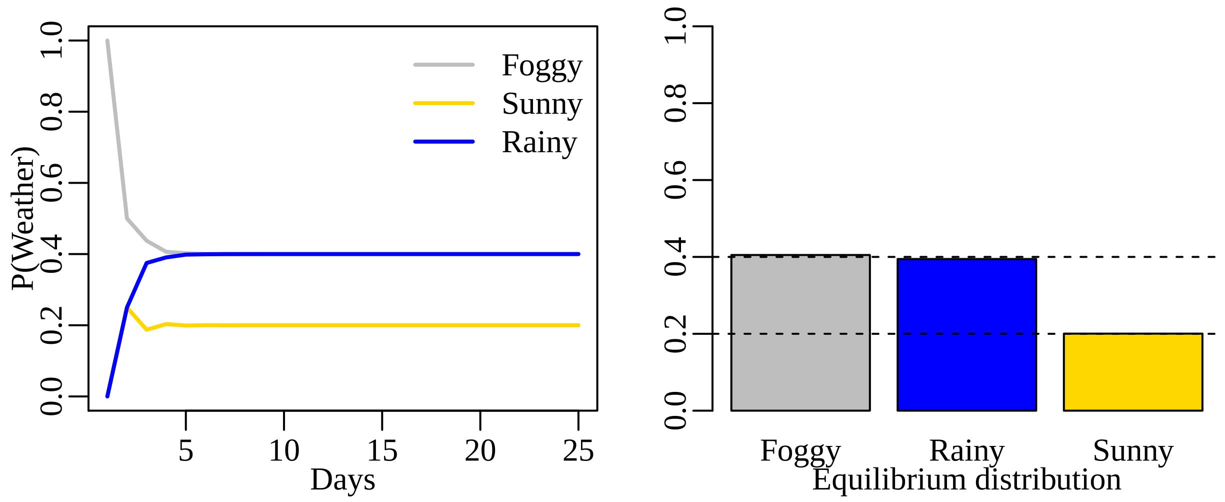


Figure 1: The probability of the weather (rainy, sunny, or foggy) predicted over time (Markov chain process; left) and the equilibrium distribution (MCMC process; right).

The following approach and assumption is taken to approximate observations,

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

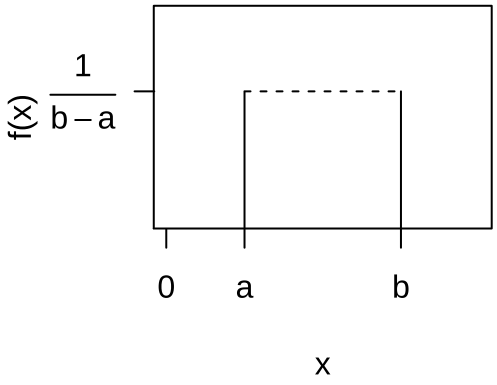
where $\theta$ denotes the unknown slope ($\alpha$) and intercept ($\beta$) parameters, and $f(\theta, t)$ denote the model output at parameter setting $\theta$ and time $t$. The observed data, $y_t$, is then equal to the physical model output plus measurement errors $\epsilon_t$.

```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 our belief about the probability that an event or observation will occur, before we have taken into account new evidence. It is based on background information. In this example, you are given very little background infromation and thus the prior in set as an improper unifrom "relatively" uninformative prior by setting the "log.prior" equal to zero. The prior is set to zero because the probability density function of a continuous unifrom distribution is

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for} \quad a \le x \le b, \\ 0 \quad \text{for} & x < a \text{ or } x > b, \end{cases}$$



where a and b are constants. We take into account new evidence using the likelihood function. The likelihood function describes the plausibility of a parameter value based on observations. Notice in the likelihood function () `log.likelihood`), we use the least squares error also known as *L2 error* rather than the least absolution deviations or *L1 error*. Using the L2 error in the likelihood function produces a "good" estimate of the mean, whereas, L1 error produces a "better" estimate of the median. 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 probability distribution on the 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 "relatively 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.

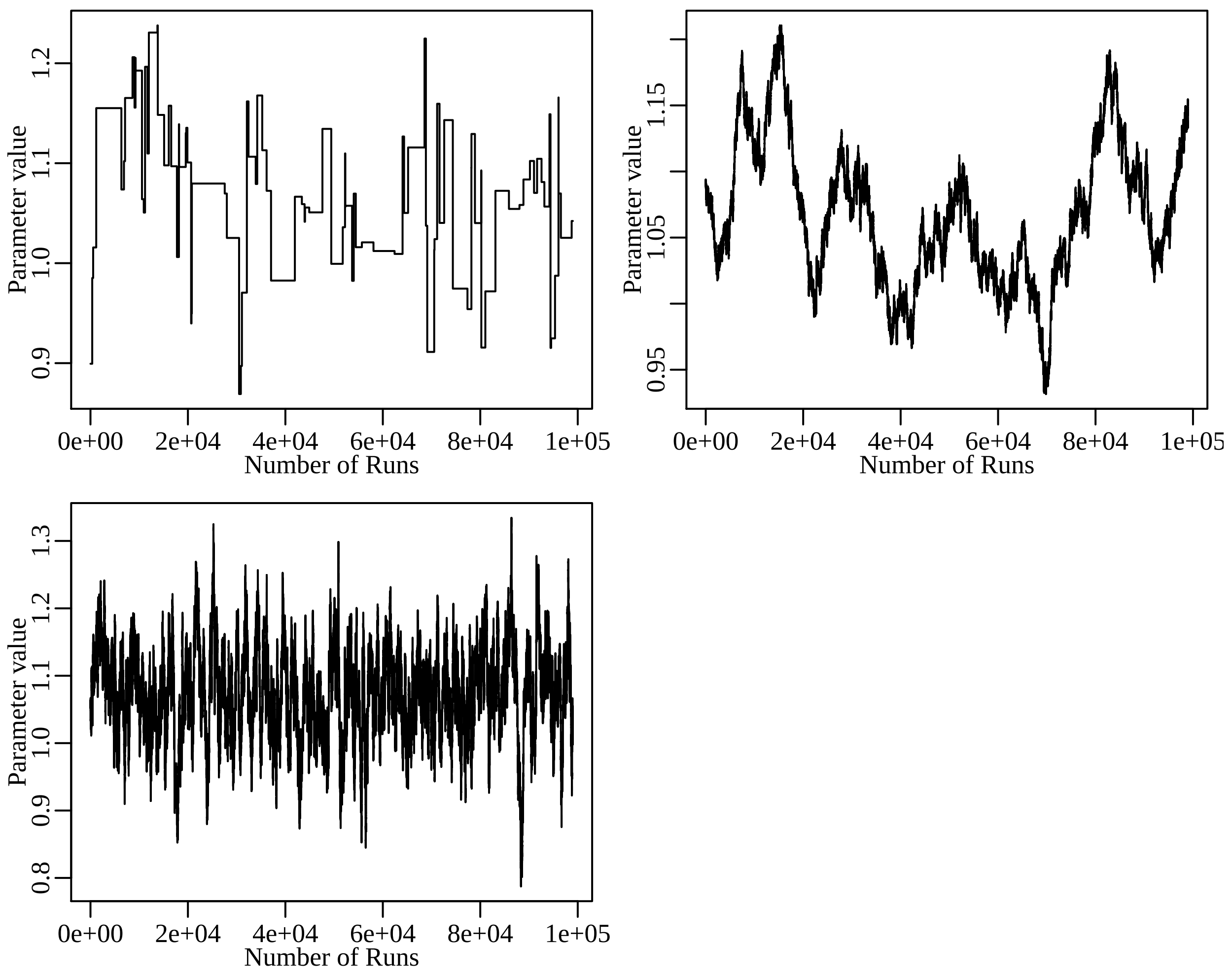


Figure 2: The chain of a single parameter that is poorly mixed with a small acceptance rate and a large step size (top left), poorly mixed with a large acceptance rate and small step size (top right), and well mixed and converged with a 'good' acceptance rate and step size (bottom left).

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. Note that in the following chapter, you will examine how to decide how many iterations is enough.

```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, $\theta^{new}$ 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). Unlike the weather example where we define the transition probability matrix, here, the transition probabilities are defined by a normal distribution centered at the current state with a covariance defined by the step sizes. 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 $p(\theta^{new} \mid y)/p(\theta^{initial} \mid y)$. In the code, this is done by first sampling the natural log of a unifromly distributed random number between one and zero. In doing so, the acceptance ratio is thus, $\min\Big(\ln(1) = 1, \quad \ln\big(p(\theta^{new} \mid y)/p(\theta^{initial} \mid y)\big) = \ln\big(p(\theta^{new} \mid y)\big) - \ln\big(p(\theta^{initial} \mid y)\big)\Big)$. 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

# Lab #3: 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)
```

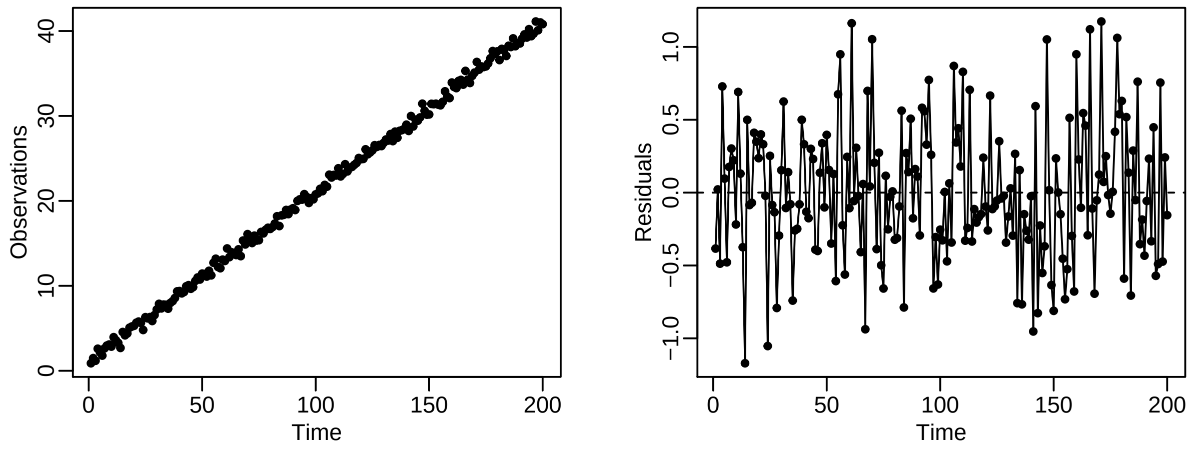


Figure 3: The original data (left) and residuals (right) as a function of time. The original data has a true value of parameter $\alpha = 0.02$, $\beta = 1$, $\sigma = 0.5$, and autocorrelation = 0.

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 $\theta$ denotes the physical parameters, $f(\theta, t)$ is the model output, $y_t$ is the observed data $y_t$, and $\epsilon_t$ 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))
}
```

### 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 ($\alpha$, the slope, and $\beta$, the intercept) while setting the statistical parameter ($\sigma$, 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 $\alpha$ and $\beta$ 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 $\alpha$ and $\beta$ to the `optim` command. These initial values can be a random guess for $\alpha$ and $\beta$ (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 $\alpha$ and $\beta$ or use the `DEoptim` command. The residuals from this optimized physical model simulation can be used to estimate a reasonable starting value for $\sigma$. 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 $\alpha$ 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 $\alpha$ is relatively small and therefore set the lower and upper bound to be -1 and 1. Similarly, the $\beta$ parameter fluctuates between 2 and 1 in the output from the optim call. From this, you can set the lower and upper bound of $\beta$ 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 $\sigma$ cannot be negative and that it is small, hence you can set the lower and upper bound of parameter $\sigma$ 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.

```
# 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 $\sigma$ (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")
```

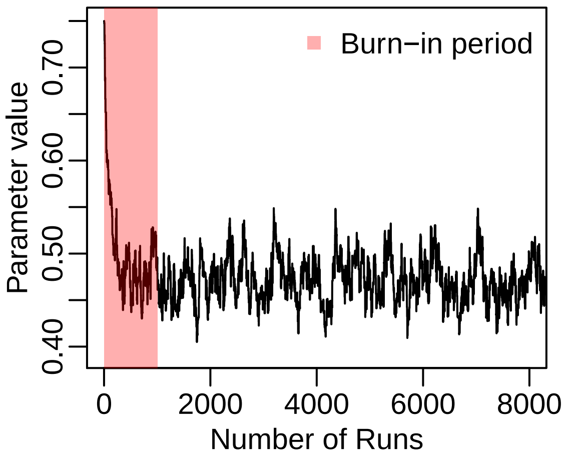


Figure 4: The chain of a single parameter showing that is well mixed and converged, but the initival values differ considerably from the converged distribution. The highlighted area shows the burn-in period that is used to remove the effects of starting values.

### 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.

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

### 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.

```
## 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 $\sigma_g^2$ 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 $\sigma$ 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 $\mu_X$ 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 $h_\alpha$ 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 ($\bar{g_n} \pm h_\alpha = \bar{g_n} \pm z(\frac{\hat{\sigma}_g}{\sqrt{n}}) \quad \text{for } n \geq 30$) 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

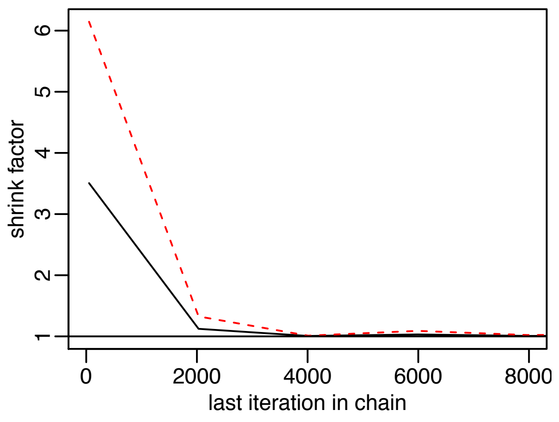
The Heidelberger and Welch diagnostic tests whether each parameter chain has stabilized. This test is split into two parts. The first part of the test examines whether the chain comes from a *stationary distribution*. A stationary distribution means that no matter the starting state, the distribution 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, $\epsilon$, 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.



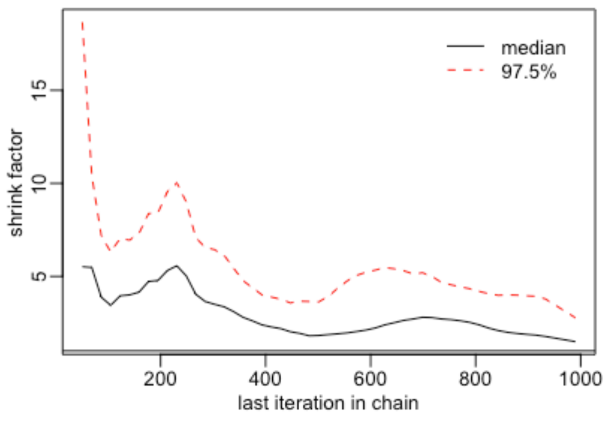


Figure 5: The potential scale reduction factors as a function of the last iteration in the chain. This parameter chain (left) flattens out to one and is hence converged. On the right, the parameter chain has not converged.

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.

```r
## 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)
```

### 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.

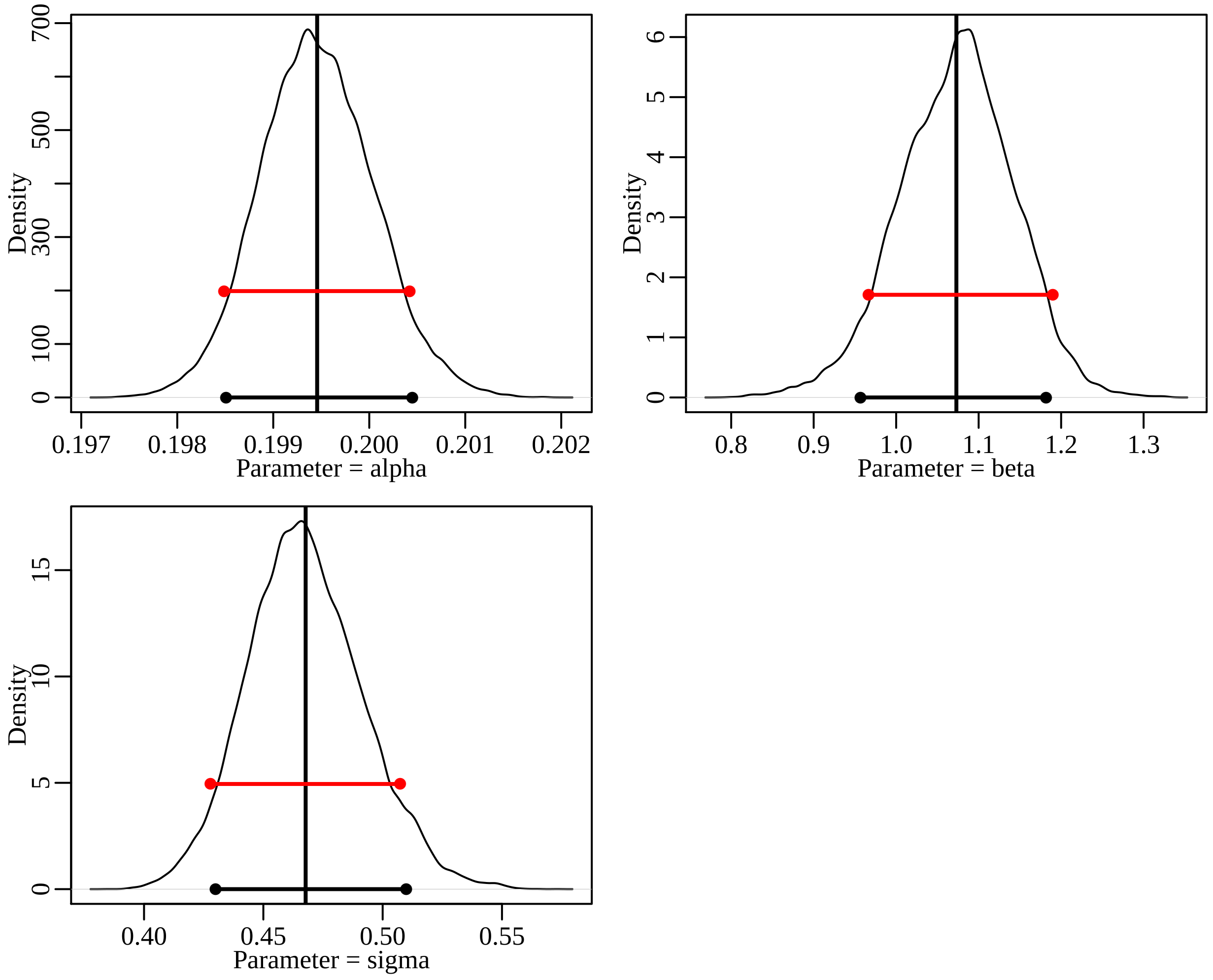


Figure 6: Probability density plot of each parameter along with the mean as a vertical line, the 90% equal-tail CI as a black horizontal line, and the 90% highest posterior density CI as a red horizontal line.

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 $\alpha$, $\beta$, and $\sigma$ 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 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

# Lab #4: Applying Markov chain Monte Carlo to sea-level data (Kelsey L. Ruckert, Tony E. Wong, Yawen Guan, and Murali Haran)

## Learning objectives

After completing this exercise, you should be able to:

- perform a Markov chain Monte Carlo analysis that accounts for autocorrelation and heteroskedasticity
- apply this approach to "real" data (sea level) and use a "real" physical model (Rahmstorf 2007)
- explain why accounting for the error structure is important

## Introduction

Suppose you have a physical model that characterizes the relationships within a system. However, this model is not perfect. There is an additional element of uncertainty. Adding a residual term is intended to account for this inaccuracy or uncertainty by accounting for model structural uncertainty, parametric uncertainty, physical process variability, and observational uncertainty not represented in the physical model. For example, sea-level data may have years when measurements are less or more certain simply because the instruments making these measurements may periodically break down, technological advancements increase the accuracy of these measurements over time, and more measurements become available. Further, deviations from the main trend of the data (residuals) could be dependent on previous observations. Testing the data for interdependent (autocorrelated) residuals and time-varying observation error (heteroskedasticity) is important because these can impact what assumptions you make about the data and the methods you choose to fit a physical model to the data.

In the previous chapters, you performed a Markov chain Monte Carlo (MCMC) calibration for a linear physical model, using data generated with non-correlated residuals. However, in many situations, observations in the Earth sciences (e.g., sea-level data) have a different and often more complicated error structure. Hence, a different approach for performing a MCMC analysis must be applied if you want to analyze data with a more complicated error structure. For instance, sea-level data is heteroskedastic, meaning that the error of the measurements varies over time. Additionally, sea-level residuals are autocorrelated, meaning that the current value is dependent upon previous values. In order to perform a MCMC analysis using sea-level data, it is necessary to check for evidence of these properties in the data. If evidence exists for this error structure, then it is important to account for autocorrelation and heteroskedasticty in the MCMC analysis that you perform.

## How do we account for the error structure?

The error structure is defined within the residual term. It is important to distinguish the difference between residuals and errors. An error is the difference of the observed value from the "unobservable" true value. On the other hand, a residual is the deviation of an observation from its estimated value or modeled value,

$$\overbrace{R_t}^{\text{residual}} = \overbrace{y_t}^{\text{observation}} - \overbrace{f(\theta, t)}^{\text{physical model}},$$

where $t$ is time and $\theta$ is the physical model parameters. The point of the residuals is to describe recognized amounts of unexplained variation in observations that are not always represented in a physical model. This variation can include a variety of uncertainties including model structure uncertainty, parametric uncertainty, physical process variability, and observational uncertainty. In this case, we approximate the residuals as the sum of the model error $\omega_t$ and observational/ measurement errors $\epsilon_t$,

$$R_t = \omega_t + \epsilon_t.$$

The model error is a time series from a multivariate normal distribution. This multivariate normal distribution is described with the variance $\sigma^2_{AR1}$ and the correlation structure given by the first-order autoregressive parameter $\rho$,

$$\omega_t = \rho \times \omega_{t-1} + \underbrace{\delta_t}_{\text{white noise}},$$

$$\omega_0 \sim N(0, \frac{\sigma^2_{AR1}}{1-\rho^2}), \qquad \delta_t \sim N(0, \sigma^2_{AR1}), \qquad \epsilon_t \sim N(0, \sigma^2_{\epsilon,t}).$$

This model is also recognized as a lag-1 autoregressive process or *AR1*. An autoregressive model is a representation of random time-varying processes in nature. In an autoregressive model, the output depends on its previous values and white noise. White noise is a sequence of continuously uncorrelated random variables with a zero mean and a constant variance $\sigma^2_{AR1}$. For AR1, only the previous term in the process, the autocorrelation coefficient, and the white noise are used to create the output. If the autocorrelation runs deeper than just an AR1 process, then it may be necessary to fit more sophisticated models (e.g., lag-2 autoregressive process or other types of autoregressive integrated moving average models).

When you account for an AR1 process, you not only modify the equation of how to approximate the residuals, but also modify the likelihood function (check out **Rahm_obs_likelihood_AR.R** and compare it to **iid_obs_likelihood_AR.R**). For instance, the modified likelihood function accounts for the observational errors $\epsilon_t$ and the autocorrelation coefficient $\rho$ so that the likelihood function is then,

$$L(y \mid \theta) = \left(\frac{1}{\sqrt{2\pi}}\right)^N \times \mid \Sigma \mid^{\frac{-1}{2}} \exp\left\{\frac{-1}{2}\big[y - f(\theta,t)\big]^{\tau}\Sigma^{-1}\big[y - f(\theta,t)\big]\right\},$$

where $\sum$ is the sum of the variance of the AR1 process and the observation errors. The variance of the AR1 process and the variance of the observation errors are both diagonal matricies of size N x N where the diagonal has the variance/ error element (see the supplementary material of Ruckert et al. 2017 for details). Additionally, $\rho$ is added as an uncertain parameter in the MCMC analysis.

In the previous chapter, we make a key assumption in the MCMC framework that the residuals are normally distributed with a mean of 0. However, in the example of calibrating changes in sea level, we fail this key assumption. Instead, the AR1 process is a Gaussian process, which is *white* (statistically uncorrelated), thereby the residuals satisfy the MCMC assumptions. Hence, in this example, we fit the AR1 error model and use the AR1 residuals. This effect of going from non-normally distributed and potentially non-mean 0 residuals to those that are normally distributed with mean 0 is called "whitening" the residuals, referring to the fact that often the signal strength of *white noise* is assumed to be normally distributed. There are actually many situations where we do not know the most appropriate error model to use, so there are many different ways we can "whiten" the residuals $(R_{t+1} - \rho R_t)$ in order to satisfy the assumptions underlying the MCMC framework. Once we whiten the residuals we can then estimate the log likelihood as the sum of the normal density of the non-correlated (whitened) residuals with a mean of 0 and a standard deviation of the square root of the variance $\sigma^2$ plus the observation errors squared.

```r
# Estimate the log likelihood of the AR1 process
logl.ar1 <- function(r, sigma1, rho1, eps1 = y.meas.err){
    n <- length(r) # r is the residuals

    logl <- 0
    if(n>1) {
      # Whiten the residuals (w)
        w <- r[2:n] - rho1*r[1:(n-1)]
        logl <- logl + sum(dnorm(w, sd=sqrt((sigma1)^2 + (eps1[c(-1)])^2), log=TRUE))
    }
    return(logl)
}
```

Here, we treat the estimation of the log prior distribution function and the log posterior distribution function the same as in the previous chapter. In this tutorial, you will use uniform distributions as priors for the physical and statistical parameters. Hence, the log prior function evaluates whether the parameter values are within the boundary and if so returns zero to represent a uniform distribution. As before, the log posterior function estimates the posterior distribution based on the log likelihood and log prior (posterior $\propto$ likelihood $\times$ prior).

## 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. This chapter builds off of the previous chapters about MCMC, so it is highly recommended that you complete those chapters before you proceed. In this exercise, you will calibrate the Rahmstorf (2007) model to sea-level data while accounting for the autocorrelated residuals and the heteroskedastic observation errors.

Before you proceed, first clear away any existing variables or figures and install packages. Install the `adaptMCMC` package. The `compiler` package should already exist in the system library and therefore does not need to be installed.

In the code block below, the command `enableJIT()` is used to increase the efficiency of R by enabling or disabling "just-in-time" compilation. Enabling "just-in-time" compilation causes the closures to be compiled before they are first used (`enableJIT(1)`), closures to be compiled before they are duplicated (`enableJIT(2)`), and all `for()`, `while()`, and `repeat()` loops to be compiled before they are executed (`enableJIT(3)`). "Just-in-time" compilation is turned off if the argument is set to 0. Test out the speed gain by tracking the time it takes to run a set of code with `proc.time()`.

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

# Install and read in packages.
# install.packages("adaptMCMC")
library(adaptMCMC)
library(compiler)
enableJIT(3)

# Set the seed.
set.seed(111)

# Make directory for this lab
dir.create("lab_X")
dir.path <- paste(getwd(), "/lab_X/", sep="")
scrim.git.path <- "https://raw.githubusercontent.com/scrim-network/"
```

Next, download and read in the sea-level observations, temperature observations, and the Rahmstorf (2007) global sea-level model. Specific details on these data sources can be found within the code block below and in previous chapters.

```r
# Global mean sea level from Church and White (2011)
# year, time in years from 1880 to 2013 at the half year mark
# slr, global mean sea level in mm
# err.obs, global mean sea level measurement error in mm
url <- paste(scrim.git.path, "BRICK/master/data/GMSL_ChurchWhite2011_yr_2015.txt", sep="")
download.file(url, paste(dir.path, "GMSL_ChurchWhite2011_yr_2015.txt", sep=""))
church.data <- read.table("lab_X/GMSL_ChurchWhite2011_yr_2015.txt")
year <- church.data[ ,1] - 0.5
```

```r
slr <- church.data[ ,2]
err.obs <- church.data[ ,3]

# Calculate sea level -/+ observation errors.
err_pos <- slr + err.obs
err_neg <- slr - err.obs

# Read in the information from the Smith (2008) temperature file.
# project.years, time in years from 1880 to 2100.
# hist.temp, historical global mean temperatures from 1880 to 2013 in C
# rcp85, merged historical + rcp 8.5 temperatures from 1880 to 2100 in C
url <- paste(scrim.git.path,
             "Ruckertetal_SLR2016/master/RFILES/Data/NOAA_IPCC_RCPtempsscenarios.csv",
             sep="")
download.file(url, paste(dir.path, "NOAA_IPCC_RCPtempsscenarios.csv", sep=""))
temp.data <- read.csv("lab_X/NOAA_IPCC_RCPtempsscenarios.csv")
project.years <- temp.data[1:match(2100, temp.data[,1]), 1]
hist.temp <- temp.data[1:match(2013, temp.data[,1]), 2]
rcp85 <- temp.data[1:match(2100, temp.data[,1]), 4]

# The sea level and temperature data have a yearly timestep.
tstep = 1

# Load the sea-level model (Rahmstorf, 2007)
url <- paste(scrim.git.path, "BRICK/master/R/gmsl_r07.R", sep="")
download.file(url, paste(dir.path, "gmsl_r07.R", sep=""))
source("lab_X/gmsl_r07.R")
```

### Characterizing the error structure

In the introduction, we discuss how scientists often must make assumptions about the data and the error structure of the data. In practice, how do we know what assumptions to make? To answer this, you can examine the observations and inform assumptions based on what information and details are known about the observations. For example, the sea-level observations `slr` start from the year 1880 and you can assume that since 1880 the measurements of sea level have become more accurate and more available. Since the data from Church and White (2011) includes error estimates, you can test that assumption of whether the data has time-varying observation error (heteroskedasticity) by plotting the errors over time. The resulting plot shows that the errors indeed vary over time.

```r
par(mfrow = c(1,1))
plot(year, err.obs, type = "l", ylab = "Observation errors [mm]", xlab = "Time")
points(year, err.obs, pch = 20)
```

Additionally, you can assume that changes in sea level this year will impact what sea level is like next year. Hence, this implies that the residuals are autocorrelated. Before testing for autocorrelation, you must calculate the residuals. Using an optimization procedure (`optim` or `DEoptim`), you can estimate the parameter values. The model parameters include the sensitivity of sea level to changes in temperature, $\alpha$; the temperature when then sea-level anomaly is zero, $T_{eq}$; and the initial sea-level anomaly value (the value in 1880), $SL_0$. These values can be plugged into the model to generate a “best fit” simulation and be later used as initial parameter estimates. Using the resulting “best-fit” simulation, you estimate the residuals by subtracting the model simulation from the observations. Taking the standard deviation of the residuals can serve as the initial estimate of $\sigma$ later on.

```
# Generate fit to sea-level observations using General-purpose Optimization.
# by calculating the root mean squared error.
fn <- function(pars, tstep, Temp, obs){
  a   <- pars[1]
  Teq <- pars[2]
  SL0 <- pars[3]

  np      <- length(Temp)
  gmsl    <- rep(NA, np)
  gmsl[1] <- SL0

  # Use Forward Euler to estimate sea level over time.
  for (i in 2:np) {
    gmsl[i] <- gmsl[i-1]  +  tstep * a * ( Temp[i-1]-Teq )
  }
  resid <- obs - gmsl
  # Return the root mean square error
  rmse <- sqrt(mean(resid^2))
  return(rmse)
}

# Plug in random values for the parameters.
parameter_guess <- c(3.4, -0.5, slr[1])

# Optimize the model to find initial starting values for parameters.
result <- optim(parameter_guess, fn, gr=NULL, tstep, Temp = hist.temp, obs = slr)
start_alpha <- result$par[1]
start_Teq <- result$par[2]
start_SL0 <- result$par[3]

# Make a plot of observations and fit.
slr.est <- gmsl_r07(a = start_alpha, Teq = start_Teq, SL0 = start_SL0, tstep,
                    Tg = hist.temp)
plot(year, slr, pch = 20, ylab = "Sea-level Anomaly [mm]", xlab = "Year")
lines(year, slr.est, col = "blue", lwd = 2)

# Calculate residuals and initial sigma value.
res <- slr - slr.est
start_sigma <- sd(res)
```

You can use the resulting residuals to test for autocorrelation with the `acf()` function. "Autocorrelation" refers to the correlation between two points within a time series, where the time difference between them is called the "lag". In the autocorrelation plot, the correlation is shown at each time lag. If the correlation exceeds the dashed blue lines (a 95% confidence interval), then the correlation of the data at that lag is considered to be "statistically significant". Values below the dashed lines are considered insignificant correlations. The correlation at lag 0 should always equal 1 because each observation should be 100% correlated to itself. The resulting plot shows that the residuals are autocorrelated with the highest correlation at the lag-1 autoregressive process or *AR1*. Accounting for the AR1 process requires adding an uncertain statistical parameter, $\rho$. $\rho$ is the first-order or lag-1 autocorrelation coefficient. For later use, you can save `rho[2]` as the initial $\rho$ estimate to set up the MCMC calibration.

```
# Apply the auto-correlation function to determine correlation coefficients.
rho <- rep(NA,3)
ac <- acf(res, lag.max = 5, plot = TRUE, main = "", xlab = "Time lag",
          ylab = "Autocorrelation function")
rho[1] <- ac$acf[1]
rho[2] <- ac$acf[2]
rho[3] <- ac$acf[3]
rho[4] <- ac$acf[4]
rho[5] <- ac$acf[5]
start_rho <- rho[2]
```

**Setting up the prior information**

In the previous blocks, you read in the data, learned about what to account for in the error structure, and estimated some good initial parameter estimates. The next goal is to learn about the distribution of the parameters ($\alpha$, $T_{eq}$, $SL_0$, $\sigma$, and $\rho$) given the observed data. First, you will specify a prior distribution for each model parameter and statistical parameter. Like the previous MCMC chapter, you will use uniform prior distributions. For the model parameters, you will use the prior distributions based on Ruckert et al. (2017; see supplementary material). Hence, the prior for $\alpha$ will be set as 0 to 20 $[mm \cdot yr^{-1} \cdot {}^{\circ}C^{-1}]$, -3 to 2 $[{}^{\circ}C]$ for $T_{eq}$, and as the sea-level observation [mm] in 1880 plus and minus the observation error in 1880 (1880 is the year of the first sea-level value) for $SL_0$. Also, set $\sigma$ as 0 to 10 [mm] because in this example the $\sigma$ cannot be negative and that it is small based on observing the residuals. $\rho$ is set to have a prior range of -0.99 to +0.99. This is because $\rho$ is a percent represented as a decimal and therefore cannot have a positive or negative correlation greater than 100%. As an added challenge (recommended in the Appendix), test out different nonuniform prior distributions for the model parameters.

The difference with the approach shown in the previous MCMC chapter is that the data have residuals that are correlated and show heteroskedasticity. To account for the differences in the error structure, we update the likelihood function to add $\rho$ as an uncertain parameter and to set the measurement errors equal to the known observational errors, `err.obs` (as stated in the introduction).

```
# Set up MCMC calibration.
# Set up priors.
bound.lower <- c( 0, -3, err_neg[1], 0, -0.99)
bound.upper <- c(20,  2, err_pos[1], 10,  0.99)

# Name the parameters and specify the number of physical model parameters.
# Sigma and rho are statistical parameters and will not be counted in the number.
parnames <- c("alpha", "Teq", "SL0", "sigma", "rho")
model.p <- 3

# Set the measurement errors.
y.meas.err <- err.obs

# Load the likelihood model accounting for the error structure.
source("Rahm_obs_likelihood_AR.R")

# Setup initial parameters.
p <- c(start_alpha, start_Teq, start_SL0, start_sigma, start_rho)

# Set p0 to a vector of values to be used in optimizing the log-posterior function.
# We optimize the negative log-posterior function because "optim" only knows how to
# minimize a function, while we would like the log-posterior to be maximized.
p0 <- c(3.4, -0.5, slr[1], 6, 0.5)
p0 <- optim(p0, function(p) -log.post(p))$par
print(round(p0, 4))
```

## Running MCMC

For this chapter, you will use a more sophisticated MCMC algorithm with an adaptive algorithm for the MCMC sampler (Vihola, 2011). This adaptive MCMC uses Metropolis-Hastings updates with joint multivariate normal proposal, which tunes the covariance matrix for these proposals to "learn" what the step sizes and directions ought to be, in order to achieve a desired acceptance. The purpose of using an adaptive MCMC over the "vanilla" MCMC version shown in the previous chapters on MCMC because 1) you will get the opportunity to learn both versions and 2) speed is a priority for this lab exercise and using adaptation can lessen the number of iterations needed to show evidence of convergence.

In `R`, the package `adaptMCMC` includes the `MCMC` function, which implements adaptive MCMC. For this function, you must specify several arguments. First, you will specify your desired acceptance rate and set adapt to `TRUE` so adaptive sampling is used. You learned from the first chapter on MCMC that "the optimal acceptance rate varies depending on the number of dimensions in the parameter space." You also learned that for a single parameter the optimal acceptance rate is about 44%, but as the number of parameters increases, the acceptance rate stabilizes around 23.4%. For the sea-level rise model, there are in total five parameters; three model parameters ($\alpha$, $T_{eq}$, $SL_0$) and two statistical parameters ($\sigma$, and $\rho$). Hence, the desired acceptance rate is likely around 23.4%. Second, set the rate of adaptation `gamma.mcmc` to equal 0.5; this is because a lower $\gamma$ leads to a faster adaptation. Next, you will set the step size based on step sizes used in the calibration of the Rahmstorf model in Ruckert et al. 2017. Even though the step sizes adapt, better initial estimates of the proposal covariance matrix yield faster convergence of the algorithm. In practice, you can play around with the "vanilla" MCMC function to observe how changing the step size for each parameter impacts the acceptance rate and use the results to inform the initial step sizes for the adaptive version. Lastly, you will set the MCMC analysis to run for $4 \times 10^5$ iterations and set the analysis to start adapting the step size after 1% of the iteration has run. Note that you do not want to start adapting too early because estimates of the parameter correlations will be weak if not enough proposals have been accepted yet. This analysis may take several minutes to complete.

```r
# Set optimal acceptance rate as # parameters->infinity
# (Gelman et al, 1996; Roberts et al, 1997)
# Set number of iterations and rate of adaptation
# (gamma.mcmc, between 0.5 and 1, lower is faster adaptation)
accept.mcmc = 0.234
gamma.mcmc = 0.5
NI <- 4e5
step <- c(0.2, 0.02, 1, 0.1, 0.01)

# Run MCMC calibration. Note: Runs for several minutes.
mcmc.out <- MCMC(log.post, NI, p0, scale = step, adapt = TRUE, acc.rate = accept.mcmc,
                 gamma = gamma.mcmc, list = TRUE, n.start = round(0.01*NI))
mcmc.chains <- mcmc.out$samples
```

## Determine the burn-in period

In the previous chapter, you applied a 1% burn-in period, but how do you decide the appropriate length of the burn-in period? Besides visual inspection, you could use the Heidelberger and Welch diagnostic or the Gelman and Rubin diagnostic. The Gelman and Rubin diagnostic technique is useful because it can also be used to check for evidence of convergence as was described in the previous chapter on MCMC. First, you will run the MCMC analysis two more times and check for convergence at different points along the chains. Once the chains show evidence for convergence, then you can determine the burn-in. For this, look at how many iterations are required for the Gelman and Rubin potential scale reduction factor to get and stay below 1.1 for all parameters. It is only after the chains are converged that you should use the parameter values as posterior draws, for analysis. Therefore, the iterations prior to the point at which the scale reduction factor falls below 1.1 is what is thrown away as the burn-in period.

```r
## Gelman and Rubin's convergence diagnostic:
set.seed(1708)
p0 <- c(1.9, -0.9, -145, 4, 0.7) # Arbitrary choice.
mcmc.out2 <- MCMC(log.post, NI, p0, scale = step, adapt = TRUE, acc.rate = accept.mcmc,
                  gamma = gamma.mcmc, list = TRUE, n.start = round(0.01*NI))
mcmc.chains2 <- mcmc.out2$samples

set.seed(1234)
p0 <- c(2.9, 0, -160, 5, 0.8) # Arbitrary choice.
mcmc.out3 <- MCMC(log.post, NI, p0, scale = step, adapt = TRUE, acc.rate = accept.mcmc,
                  gamma = gamma.mcmc, list = TRUE, n.start = round(0.01*NI))
mcmc.chains3 <- mcmc.out3$samples

# Turn the chains into an mcmc object.
# This is a requirement for the gelman.diag and gelman.plot commmand.
mcmc1 <- as.mcmc(mcmc.chains)
mcmc2 <- as.mcmc(mcmc.chains2)
mcmc3 <- as.mcmc(mcmc.chains3)

# Test for convergence.
set.seed(111) # revert back to original seed
mcmc_chain_list <- mcmc.list(list(mcmc1, mcmc2, mcmc3))
gelman.diag(mcmc_chain_list)
gelman.plot(mcmc_chain_list)

# Create a vector to test the statistic at several spots throughout the chain.
# Test from 5000 to 4e5 in increments of 5000.
niter.test <- seq(from = 5000, to = NI, by = 5000)
gr.test <- rep(NA, length(niter.test))

# Once the statistic is close to 1 (adopting the standard of < 1.1), the between-chain
# variability is indistinguishable from the within-chain variability, and hence converged.
for (i in 1:length(niter.test)){
mcmc1 = as.mcmc(mcmc.chains[1:niter.test[i],])
mcmc2 = as.mcmc(mcmc.chains2[1:niter.test[i],])
mcmc3 = as.mcmc(mcmc.chains3[1:niter.test[i],])
mcmc_chain_list = mcmc.list(list(mcmc1, mcmc2, mcmc3))
gr.test[i] = gelman.diag(mcmc_chain_list)[2]
}

# Plot Gelman and Rubin statistics as a function of iterations. The iterations prior
# to the point in which convergence happens is what we set as the burn-in.
plot(niter.test, gr.test, pch=20)
abline(h = 1.1, col = "red")
gr.test < 1.1; first.stat = max(which((gr.test < 1.1) == FALSE))+1; print(first.stat)

# Set the burn-in to the 1st statistic that remains under 1.1 and remove it.
burnin <- seq(1, niter.test[first.stat], 1)
slr.burnin.chain <- mcmc.chains[-burnin, ]
```

### Hindcasting and projecting sea-level rise

The parameters from the chains (burn-in removed) can be plugged into the model to generate multiple simulations of the data, called an "ensemble". The residuals can be added to the simulations of the data to generate hindcasts and projections for further analysis. In the exercise below, you will generate hindcasts and projections with this process.

If you insert the following lines at the bottom of your R script, then you will make sea-level rise projections as a matrix (`SLR.projections`), using the sea-level model and multiple `for` loops. In `SLR.projections`, each row represents a state of the world while the columns represent time. Because you are using all $4 \times 10^5$ iterations, this code block will take several minutes to fully generate the projections. For an additional exercise, rather than using multiple `for` loops try to improve on the code by using the `apply` or `sapply` function in R, or vectorize the internal `for` loop in the `SLR.residuals` calculation. Does using any of these other options increase the speed of the code?

```r
# Extract parameter vectors from the chain to enhance code readability.
alpha.chain <- slr.burnin.chain[ ,1]
Teq.chain <-   slr.burnin.chain[ ,2]
SL_0.chain <-  slr.burnin.chain[ ,3]
sigma.chain <- slr.burnin.chain[ ,4]
rho.chain <-   slr.burnin.chain[ ,5]

# Set up empty matrices for sea-level output.
# Loop over the sea level model to generate a distribution of sea level simulations.
NI_length <- length(slr.burnin.chain[ ,1])
SLR.model.sim <- mat.or.vec(NI_length, length(project.years))
for(n in 1:NI_length) {
  SLR.model.sim[n, ] = gmsl_r07(a = alpha.chain[n], Teq = Teq.chain[n],
                                SL0 = SL_0.chain[n], tstep, Tg = rcp85)
}

# Estimate the residuals with the AR(1) coefficient and sigma.
SLR.residuals <- mat.or.vec(NI_length, length(project.years)) #(nr,nc)
for(n in 1:NI_length) {
  for(i in 2:length(project.years)) {
    SLR.residuals[n,i] <- rho.chain[n]*SLR.residuals[n,i-1] +
      rnorm(1, mean = 0, sd = sigma.chain[n])
  }
}

# Estimate the hindcasts and projections: add the residuals onto the model simulations.
SLR.projections <- mat.or.vec(NI_length, length(project.years)) #(nr,nc)
for(i in 1:NI_length) {
  SLR.projections[i,] <- SLR.model.sim[i,] + SLR.residuals[i,]
}
```

## Exercise

*Part 1.* Using the diagnostics specified in the previous chapter, check the analysis for evidence of convergence. In your lab report, provide evidence to support or reject whether the chains are converged. If the chains are not converged, then explain how you would go about getting the chains to converge. The goal of this chapter is to understand how to apply a Markov chain Monte Carlo analysis to "real" data and use a "real" model, so getting the chains to converge—while extremely important in a physical application—is not a top priority in this chapter due to time constraints.

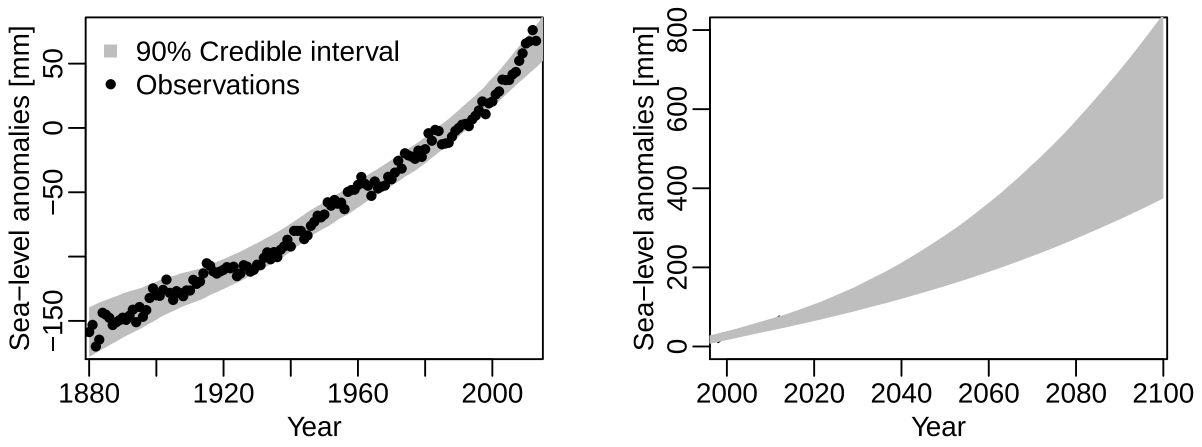


Figure 7: The sea level anomaly data from Church and White (2011; left) with its associated 90% credible interval, and the 90% credible interval projections of sea level anomalies from 2000 to 2100 in mm (right).

*Part 2.* Modify your script from Part 1 to:

- calculate and plot the 90% credible interval of the sea-level projections (to the year 2100) using the RCP8.5 temperature scenario (hint: consider using `quantile()` and `polygon()`),
- and to make density plots of the parameters $\alpha$, $T_{eq}$, $SL_0$, and $\rho$.

It is highly recommended that you do not plot all $4 \times 10^5$ projections as individual lines at once, but rather plot credible intervals. Ultimately, you should end up with a plot similar to Figure 1. If not, check your script against the codes listed through the tutorial to make sure your script is working properly.

*Part 3.* Save a copy of your script from Part 2 using a different file name. In this script, modify the MCMC analysis to assume the errors are homoskedastic (constant through time) by changing `y.meas.err <- err.obs` to a vector where the values equal 0, `y.meas.err <- rep(0, length(slr))`. Setting the measurement errors to 0 combines the model error and observational error into one term.

Your scripts should make the following plots:

1. a two-panel plot displaying the autocorrelation function and the time dependent measurement error of the sea-level data.
2. a plot comparing the 90% credible interval of projections made from considering heteroskedasticity (part 2) versus assuming homoskedasticity (part 3).
3. a four-panel plot comparing the densities of parameters $\alpha$, $T_0$, $H_0$, and $\rho$ made from considering heteroskedasticity (part 2) versus assuming homoskedasticity (part 3).

## Questions

1. Compare the sea-level projections from *part 2* and *part 3*. How do the results differ? How much larger or smaller is the 90% credible interval in *part 1* from *part 2*? How might the assumptions made about the data impact "real-life" decisions regarding flood protection?
2. Compare the parameter distributions from *part 2* and *part 3*. How do the results differ? Do both methods produce the same mode and the same width in distribution?
3. Refer to the paper Ruckert et al. (2017) (link) and the results from *part 2* and *part 3*. Does it matter how you account for the error structure of data? Explain your reasoning.

## Appendix

####Comparing un-whitened raw residuals to the whitened residuals In the exercise above, you whiten the residuals from the sea-level data in order to satisfy the MCMC assumptions. This process changes the residuals to be normally distributed. Using the initial residuals and *rho* calculate the whitened residuals and compare them to the un-whitened raw residuals and to a normal distribution. For this case, does whitening the residuals make a difference?

```r
# Whiten the residuals
n <- length(res)
w_res <- res[2:n] - start_rho*res[1:(n-1)]

par(mfrow = c(1,1))
plot(density(w_res), col="black", xlab="Residuals", yaxt="n", main="")
lines(density(res), col="red")
lines(density(rnorm(1e5, sd=5)), col="gray", lty=2)
abline(v=0, col="gray", lty=2)
legend("topleft", c("Whitened", "Un-whitened", "Normal distribution\nwith mean 0"),
       lty=c(1,1,2), col=c("black", "red", "gray"), bty="n")
```

####Using different prior distributions Modify the prior parameter distributions (priors do not have to be uniform distributions). Explain your reasoning behind the modifications and rerun the MCMC calibration. How do the results differ?

####Thinning the chain *Thinning* the chain is the process of taking a subsample of the chain instead of all of the chain. For example, you can take every "n'th" observation from the chain. Note that thinning the chain is optional and often not necessary. Here, we give an example of how it is done in practice.

Thinning can be useful for two reasons: (1) to reduce storage burden and (2) to increase computational speed. A good way to inform the choice of thinning is to look at the autocorrelation function of the Markov chains. We adopt the standard that the chains should be no smaller in length than the lag at which the autocorrelation function is below 0.05 or (5%), so the thinned parameters are not substantially autocorrelated (Link and Eaton, 2012). The plots show the parameters are relatively uncorrelated by roughly the 5,000th lag (this number may vary because the chains are not converged). This means that to properly thin the chain only every 5,000th iteration (or more) should be saved and that the samples are in general highly correlated with one another.

```r
# The diagonal ACF plots are the plots of interest
par(mfrow = c(3,2))
for(i in 1:5){
  acf(slr.burnin.chain[,i], lag.max = 20000,
      main = paste('Parameter = ', parnames[i], sep = ''))
  abline(h = 0.05, col = "red")
}
```

When you thin the chains, visually check to make sure the full chain and the subset have almost identical posterior parameter distributions, so no important information is lost. The plot created below shows that the distributions are similar in shape, however, they do not visually match in shape.

```r
# Thin the chain to a subset by taking every 5000th iteration (this number
# may vary if the chains are not converged). Another option is to
# use the sample() command.
subset_length <- round(nrow(slr.burnin.chain)/5000, 0)
sub_chain = slr.burnin.chain[seq(1, nrow(slr.burnin.chain), 5000), ]
# sub_chain2 <- slr.burnin.chain[sample(nrow(slr.burnin.chain), size = subset_length,
#                                     replace = FALSE), ]
# Check for similarities between full chain and the subset.
par(mfrow = c(3,2))
for(i in 1:5){
  plot(density(slr.burnin.chain[ ,i]), type="l",
       xlab = paste('Parameter =',' ', parnames[i], sep=''), ylab="PDF", main="")
  lines(density(sub_chain[ ,i]), col="red")
}
```

In this chapter, the output was not thinned. Why was thinning not necessary as well as inappropriate for this example?

####Applying MCMC to other data and models In previous exercises, you used the sea-level data from Jevrejeva et al (2008). Apply MCMC analysis accounting for autocorrelation and heteroskedastic residuals on the Jevrejeva et al (2008) data. For this, use the second order polynomial from earlier chapters as the model to fit to the data. You can extract the third vector from `sl.data` as your measurement error, `err.sl <- sl.data[ ,3]` and do not forget to set `y.meas.err <- err.sl`. Examine your sea-level rise estimate in 2100. How do your estimates differ from Exercise #7? How do your model parameter distributions differ from Exercise #7?

####Bootstrapping the sea-level model Apply the Bootstrap method to the Rahmstorf (2007) model with the data used in this chapter. How would one account for autocorrelation? Can one account for heteroskedasticity using the Bootstrap method?

## 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

Church, J. A. and White, N. J. 2011. Sea-Level Rise from the Late 19th to the Early 21st Century. Surv Geophys, 32, 585-602. Available online at https://link.springer.com/article/10.1007/s10712-011-9119-1.

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

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. Available online at https://www.jstor.org/stable/2334940?seq=1#page_scan_tab_contents

Jevrejeva, S., Moore, J. C., Grinsted, A., and Woodworth, P. L., 2008. Recent global sea level acceleration started over 200 years ago? Geophysical Research Letters 35, L08715. Available online at http://onlinelibrary.wiley.com/doi/10.1029/2008GL033611/full

Link, W. A. and Eaton, M. J. 2012. On thinning of chains in MCMC. Methods in Ecology and Evolution 3, 112-115. Available online at http://onlinelibrary.wiley.com/doi/10.1111/j.2041-210X.2011.00131.x/abstract

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

Rahmstorf, S. 2007. A Semi-empirical approach to projecting future sea-level rise. Science 315, 368–370. Available online at http://science.sciencemag.org/content/315/5810/368

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

Smith, T. M., Reynolds, R. W., Peterson, T. C., and Lawrimore, J. 2008. Improvements to NOAA’s historical merged land-ocean surface temperature analysis (1880-2006). J. Clim. 21, 2283–2296. Available online at http://journals.ametsoc.org/doi/abs/10.1175/2007JCLI2100.1

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

Zellner, A. and Tiao, G. C. 1964. Bayesian analysis of the regression model with autocorrelated errors. J. Am. Stat. Assoc. 59, 763–778. available online at http://www.sciencedirect.com/science/article/pii/S0167947301000846

# Lab #5: Global sensitivity analyses (Tony E. Wong, Vivek Srikrishnan and Klaus Keller)

## Lab #6: Decision-making under uncertainty (Vivek Srikrishnan and Klaus Keller)

## Lab #7: Multi-objective decision analysis with a lake management problem (Vivek Srikrishnan, Caitlin Spence, and Giacomo Marangoni)

## Lab #8: Introduction to multi-objective evolutionary algorithms and multi-objective optimization (Vivek Srikrishnan, Caitlin Spence, Giacomo Marangoni, and Patrick Applegate)

# Lab #9: Vulnerability analysis and robust decision-making (Vivek Srikrishnan, Caitlin Spence, and Giacomo Marangoni)

# Lab #10: Principles of data visualization (Vivek Srikrishnan, Kelsey L. Ruckert, and Klaus Keller)

# Lab #11: Visualizing uncertainty using sea-level rise and storm surge (Vivek Srikrishnan, Kelsey L. Ruckert, and Klaus Keller)

on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

```
a) Convey the object code in, or embodied in, a physical product
(including a physical distribution medium), accompanied by the
Corresponding Source fixed on a durable physical medium
customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product
(including a physical distribution medium), accompanied by a
written offer, valid for at least three years and valid for as
long as you offer spare parts or customer support for that product
model, to give anyone who possesses the object code either (1) a
copy of the Corresponding Source for all the software in the
product that is covered by this License, on a durable physical
medium customarily used for software interchange, for a price no
more than your reasonable cost of physically performing this
conveying of source, or (2) access to copy the
Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the
written offer to provide the Corresponding Source.  This
alternative is allowed only occasionally and noncommercially, and
only if you received the object code with such an offer, in accord
with subsection 6b.

d) Convey the object code by offering access from a designated
place (gratis or for a charge), and offer equivalent access to the
Corresponding Source in the same way through the same place at no
further charge.  You need not require recipients to copy the
Corresponding Source along with the object code.  If the place to
copy the object code is a network server, the Corresponding Source
may be on a different server (operated by you or a third party)
that supports equivalent copying facilities, provided you maintain
clear directions next to the object code saying where to find the
Corresponding Source.  Regardless of what server hosts the
Corresponding Source, you remain obligated to ensure that it is
available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided
you inform other peers where the object code and Corresponding
Source of the work are being offered to the general public at no
charge under subsection 6d.
```

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation