Markov Chain Monte Carlo

Felipe José Bravo Márquez

July 15, 2021

Markov Chain Monte Carlo

- This class introduces estimation of posterior probability distributions using a stochastic process known as **Markov chain Monte Carlo** (MCMC).
- Here we'll produce samples from the joint posterior without maximizing anything.
- We will be able to sample directly from the posterior without assuming a Gaussian, or any other, shape.
- The cost of this power is that it may take much longer for our estimation to complete.
- But the benefit is escaping multivariate normality assumption of the Laplace approximation.
- More advanced models such as the generalized linear and multilevel models tend to produce non-Gaussian posterior distributions.
- In most cases they cannot be estimated at all with the techniques of earlier classes.
- This class is based on Chapter 9 of [McElreath, 2020] and Chapter 7 of [Kruschke, 2014].

Markov Chain Monte Carlo

- The essence of MCMC is to produce samples from the posterior $f(\theta|d)$ by only accessing a function that is proportial to it.
- This proportial function is the product of the likelihood and the prior $f(d|\theta) * f(\theta)$, which is always available in a Bayesian model.
- So, merely by evaluating $f(d|\theta) * f(\theta)$, without normalizing it by $f(d)$, MCMC allows us to generate random representative values from the posterior distribution.
- This property is wonderful because the method obviates direct computation of the evidence $f(d)$, which, as you'll recall, is one of the most difficult aspects of Bayesian inference.
- It has only been with the development of MCMC algorithms an software that Bayesian inference is applicable to complex data analysis.
- And it has only been with the production of fast and cheap computer hardware that Bayesian inference is accessible to a wide audience.
- The question then becomes this: How does MCMC work? For an answer, let's ask a politician.

A politician stumbles upon the Metropolis algorithm

- Suppose an elected politician lives on a long chain of islands.
- He is constantly traveling from island to island, wanting to stay in the public eye.
- At the end of a day he has to decide whether to:
  1. stay on the current island
  2. move to the adjacent island to the left
  3. move to the adjacent island to right
- His goal is to visit all the islands **proportionally** to their **relative population**.
- But, he doesn't know the total population of all the islands.
- He only knows the population of the current island where he is located.
- He can also ask about the population of an adjacent island to which he plans to move.

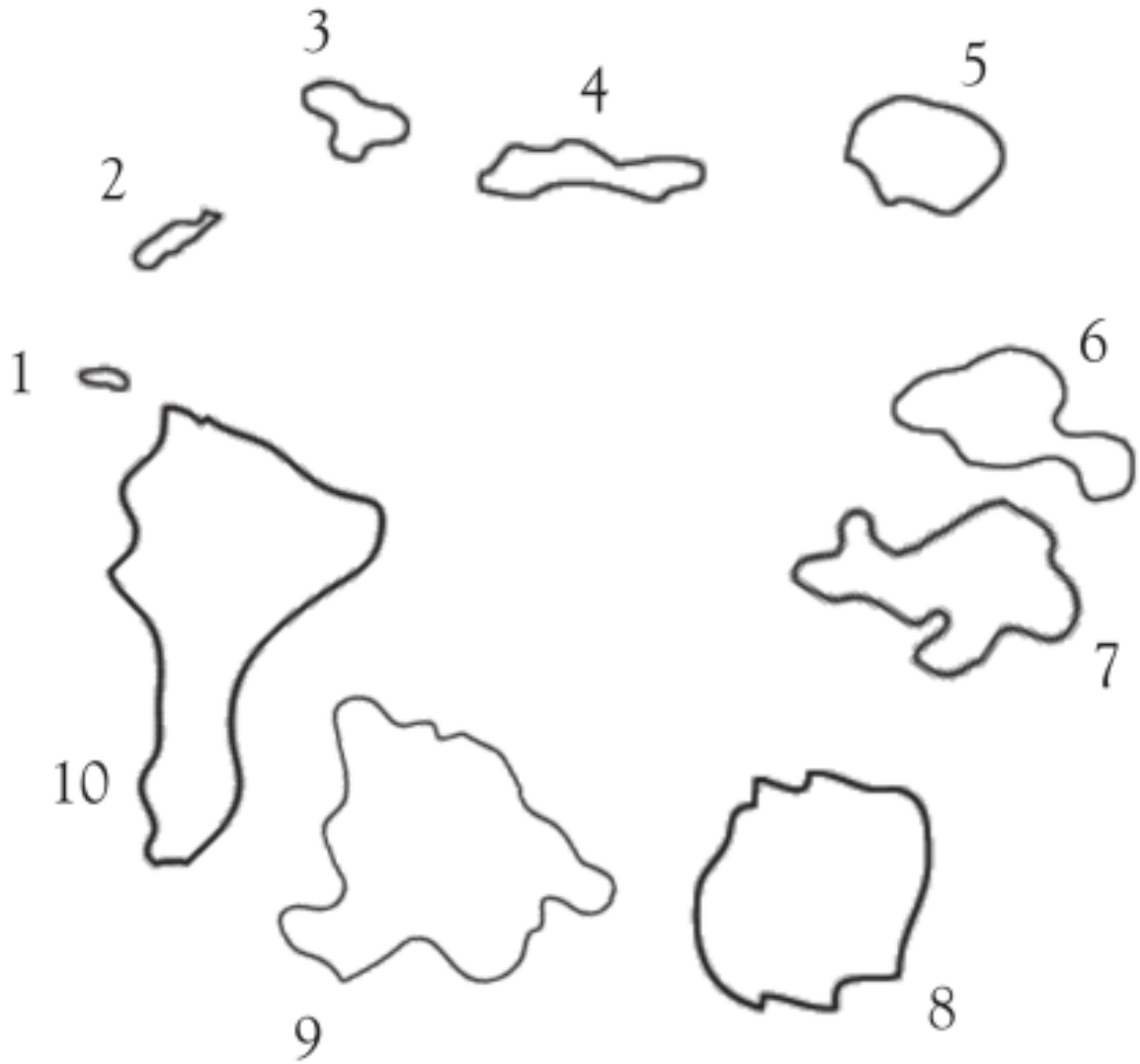
The Metropolis algorithm

- The politician has a simple heuristic for travelling accross the islands called the **Metropolis** algorithm [Metropolis et al., 1953].
- First, he flips a (fair) coin to decide whether to propose the adjacent island to the left or the adjacent island to the right.
- If the proposed island has a larger population than the current island ($P_{proposed} > P_{current}$), then he goes to the proposed island.
- If the proposed island has a smaller population than the current island ($P_{proposed} < P_{current}$), then he goes to the proposed island with probability $p_{move} = P_{proposed}/P_{current}$.
- In the long run, the probability that the politician is on any one of the islands exactly matches the relative population of the island!

The Metropolis algorithm

- Let's analyze the Metropolis algorithm in more detail.
- Suppose there are 10 islands in total.
- Each island is neighbored by two others, and the entire archipelago forms a ring.
- The islands are of different sizes, and so had different sized populations living on them.
- The second island is twice as populous as the first, the third three times as populous as the first.
- And so on, up to the largest island, which is 10 times as populous as the smallest.

The Metropolis algorithm



The Metropolis algorithm

- We are going to show an implementation of this algorithm in R.
- But before that, we will combine combine the two possibilities for the probability of moving into a single expression: the proposed island having a 1) higher or 2) lower population than the current island.

$$p_{move} = \min(1, P_{proposed}/P_{current}). \quad (1)$$

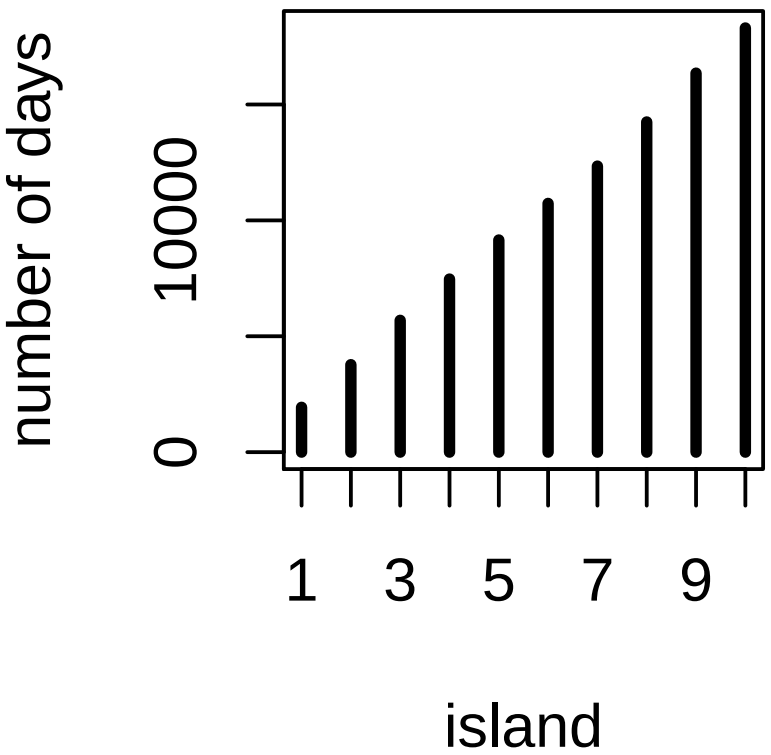
- So, if $P_{proposed} > P_{current}$, $P_{proposed}/P_{current} > 1$ and $p_{move} = 1$.
- For example, $current = 4$ and $proposed = 5$, $5/4 > 1$ so we move to the proposed island (with probability 1).
- On the other hand, if $P_{proposed} < P_{current}$, $P_{proposed}/P_{current} < 1$, and $p_{move} = P_{proposed}/P_{current}$.
- For example, $current = 4$ and $proposed = 3$, $3/4 < 1$ so we move to the proposed island with probability $3/4$.

The Metropolis algorithm

```
num_days <- 1e5
positions <- rep(0,num_days)
current <- 10
for ( i in 1:num_days ) {
  # record current position
  positions[i] <- current
  # flip coin to generate proposal
  proposal <- current + sample( c(-1,1) , size=1 )
  # now make sure he loops around the archipelago
  if ( proposal < 1 ) proposal <- 10
  if ( proposal > 10 ) proposal <- 1
  # move?
  prob_move <- min(proposal/current,1)
  decision <- rbinom(1,1,prob_move)
  current <- ifelse( decision == 1 , proposal , current )
}

library(rethinking)
simplehist(positions,xlab="island",ylab="number of days")
```

The Metropolis algorithm



The time spent on each island is proportional to its population size.

The Metropolis algorithm

- The first three lines of the method just define the number of days to simulate, an empty history vector, and a starting island position (the biggest island, number 10).
- Then the for loop steps through the days.
- Each day, it records the politician's current position.
- Then it simulates a coin flip to nominate a proposal island.
- The only trick here lies in making sure that a proposal of "11" loops around to island 1 and a proposal of "0" loops around to island 10.
- Finally, a random binary number is generated with a Bernoulli distribution (Binomial with 1 trial) with probability of success (or moving)$= \min(1, P_{proposed}/P_{current})$.
- If this random number is 1 we move, otherwise we stay.

The Metropolis algorithm

- In real applications, the goal is not to help a politician, but instead to draw samples from an unknown and usually complex posterior probability distribution.
- The “islands” in our objective are parameter values θ, and they need not be discrete, but can instead take on a continuous range of values as usual.
- The “population sizes” in our objective are the posterior probabilities (or densities) at each parameter value: $f(\theta|d)$
- The “days” in our objective are samples taken from the posterior distribution.
- The Metropolis algorithm will eventually give us a collection of samples from the posterior.
- We can then use these samples just like all the samples we have already used in this course.

Why it works

- Now, let's try to understand why the algorithm works.
- Consider two adjacent positions and the probabilities of moving from one to the other.
- We'll see that the relative transition probabilities, between adjacent positions, exactly match the relative values of the target distribution.
- Extrapolate that result across all the positions, and you can see that, in the long run, each position will be visited proportionally to its target value.
- Suppose we are at position θ.
- The probability of moving to $\theta + 1$, denoted $P(\theta \rightarrow \theta + 1)$, is the probability of proposing that move times the probability of accepting it if proposed, which is:

$$P(\theta \rightarrow \theta + 1) = 0.5 \times \min(P(\theta + 1)/P(\theta), 1)$$

Why it works

- On the other hand, if we are presently at position $\theta + 1$, the probability of moving to θ is:

$$P(\theta + 1 \rightarrow \theta) = 0.5 \times \min(P(\theta)/P(\theta + 1), 1)$$

- The ratio of the transition probabilities is:

$$\frac{p(\theta \rightarrow \theta+1)}{p(\theta+1 \rightarrow \theta)} = \frac{0.5 \min\left(P(\theta+1)/P(\theta), 1\right)}{0.5 \min\left(P(\theta)/P(\theta+1), 1\right)}$$

$$= \begin{cases} \frac{1}{P(\theta)/P(\theta+1)} & \text{if } P(\theta+1) > P(\theta) \\ \frac{P(\theta+1)/P(\theta)}{1} & \text{if } P(\theta+1) < P(\theta) \end{cases}$$

$$= \frac{P(\theta+1)}{P(\theta)}$$

Why it works

- The last equation tells us that during transitions back and forth between adjacent positions, the relative probability of the transitions exactly matches the relative values of the target distribution.
- That might be enough to get the intuition that, in the long run, adjacent positions will be visited proportionally to their relative values in the target distribution.
- If that's true for adjacent positions, then, by extrapolating from one position to the next, it must be true for the whole range of positions.
- In more mathematical terms, this means that the transition probabilities form a Markov chain that has the target distribution as its equilibrium or stationary distribution. [Wikipedia, 2021]
- Hence, one can obtain a sample of the desired distribution by recording states from the chain.

The Metropolis Algorithm more Generally

- So far, we have only considered the case with a single discrete parameter θ that can only move to the left or right.
- The general Metropolis algorithm allows working with multiple continuous parameters $\theta_1, \theta_2, \ldots, \theta_n$ and more general proposal distributions.
- The essentials of the general method are the same as for the simple case.
- First, we have some target distribution $P(\theta)$ (θ can be a vector of parameters) from which we would like to generate representative sample values.
- We must be able to compute the value of $P(\theta)$ for any candidate value of θ.
- The distribution, $P(\theta)$, does not have to be normalized, however.
- Just needs needs to be nonnegative.

The Metropolis Algorithm more Generally

- In our Bayesian inference application $P(\theta)$ is the unnormalized posterior distribution on θ, which is the product of the likelihood and the prior: $f(d|\theta) * f(\theta)$.
- This is a very important property of MCMC, as it allows us to draw samples from the posterior without having to calculate the evidence $f(d)$.
- Sample values from the target distribution are generated by taking a random walk through the parameter space.
- Proposal distributions can take many different forms, the goal being to use a proposal distribution that efficiently explores the regions of the parameter space where $P(\theta)$ has most of its probability area.
- The generic case is using a Gaussian distribution centered at the current position.
- So the proposed move will typically be near the current position, with the probability of proposing a more distant position dropping off according to the normal curve.
- For multivariate target distributions, we can use a Multi-variate Gaussian to propose multi-dimensional points in each step.

Gibbs Sampling

- The Metropolis algorithm works whenever the probability of proposing a jump to B from A is equal to the probability of proposing A from B, when the proposal distribution is symmetric (such as a Gaussian distribution).
- There is a more general method, known as Metropolis-Hastings, that allows asymmetric proposals.
- This would mean, that the politician's coin were biased to lead him clockwise on average.
- Asymmetric proposal distributions allows us to explore the posterior distribution more efficiently (i.e., acquire a good image of the posterior distribution in fewer steps).
- Gibbs sampling is a variant of the Metropolis-Hastings algorithm that uses clever proposals and is therefore more efficient.
- The improvement arises from adaptive proposals in which the distribution of proposed parameter values adjusts itself intelligently, depending upon the parameter values at the moment.

Gibbs Sampling

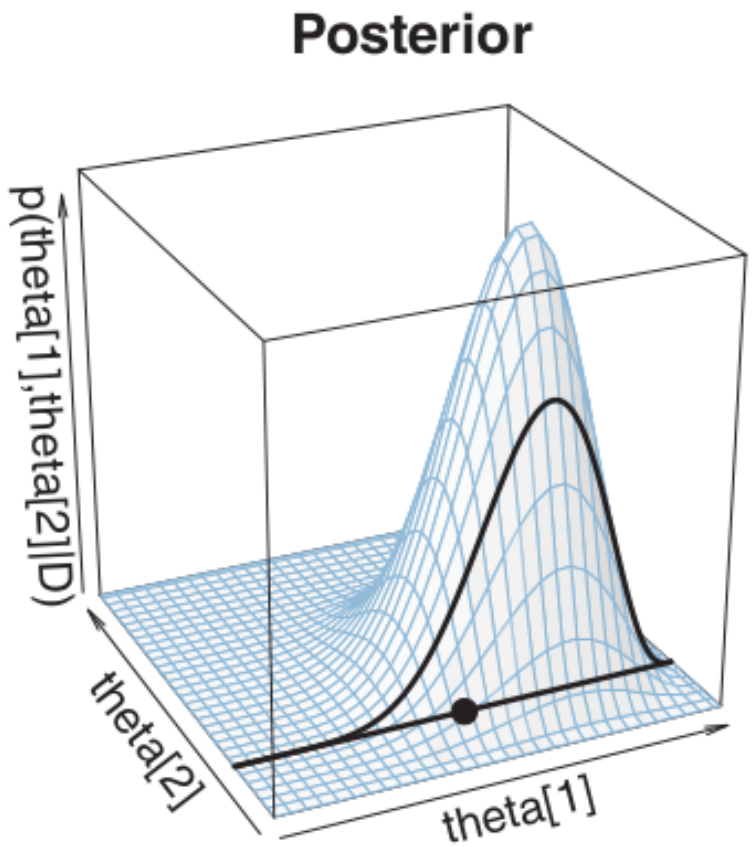
- How Gibbs sampling computes these adaptive proposals depends upon using conjugate combinations of priors and likelihoods (such as the Beta and the Binomial).
- As previously presented, conjugate combinations have analytical solutions for the posterior distribution of an individual parameter.
- And these solutions are what allow Gibbs sampling to make smart jumps around the joint posterior distribution of all parameters.
- The algorithm works as follows:
- At each point in the walk, the parameters are selected in an iterative cycle: $\theta_1, \theta_2, \theta_3, \ldots \theta_1, \theta_2, \theta_3, \ldots$.
- Suppose that parameter θ_i has been selected.
- Gibbs sampling then chooses a new value for that parameter by generating a random value directly from the conditional probability distribution of that parameter given all the others and d:

$$f(\theta_i|\theta_1, \ldots, \theta_{i-1}, \theta_{i+1}, \ldots, \theta_n, d)$$

Gibbs Sampling

- Since we are using conjugate combinations, this conditional distribution has a closed form that facilitates the sampling of random numbers from it.
- The new value for θ_i , combined with the unchanged values of $\theta_1, \ldots, \theta_{i-1}, \theta_{i+1}, \ldots, \theta_n$, constitutes the new position in the random walk.
- The process then repeats: select the next parameter θ_{i+1} and select a new value for that parameter from its conditional posterior distribution.
- Let's illustrate this process for a two-parameter example: θ_1, θ_2.
- In the first step, we want to select a new value for θ_1 .
- We conditionalize on the values of all the other parameters from the previous step in the chain.
- In this example, there is only one other parameter, namely θ_2.

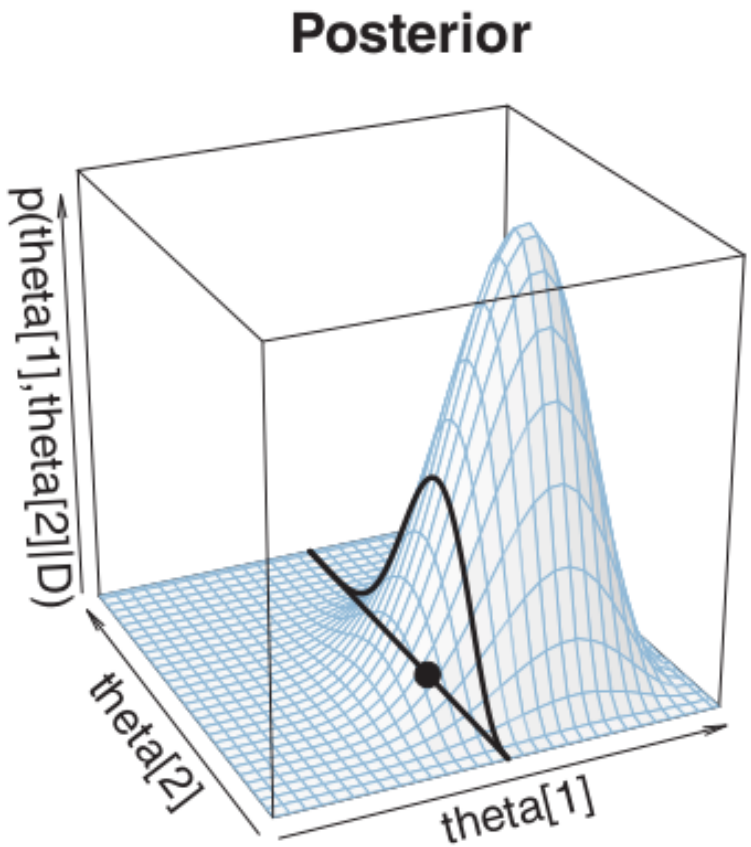
Gibbs Sampling



- The figure shows a slice through the joint distribution at the current value of θ_2.
- The heavy curve is the posterior distribution conditional on this value of θ_2, which is $f(\theta_1|\theta_2, d)$ in this case because there is only one other parameter.

Gibbs Sampling

- Because we are using conjugate distributions a computer can directly generate a random value of θ_1 from $f(\theta_1|\theta_2, d)$.
- Having generated a new value for θ_1, we then conditionalize on it and determine the conditional distribution of the next parameter, θ_2 using $f(\theta_2|\theta_1, d)$ as shown below:



- We generate a new value of θ_2 , and the cycle repeats.

Gibbs Sampling

- Because the proposal distribution exactly mirrors the posterior probability for that parameter, the proposed move is always accepted.
- Hence, the algorithm is more efficient than the standard Metropolis algorithm in which proposals are rejected in many cases.
- But there are some limitations to Gibbs sampling.
- First, there are cases when we don't want to use conjugate priors.
- Second, it can become inefficient with complex models containing hundreds, thousands or tens of thousands of parameters.

Hamiltonian Monte Carlo

- The Metropolis algorithm and Gibbs sampling are both highly random procedures.
- They try out new parameter values and see how good they are, compared to the current values.
- But Gibbs sampling gains efficiency by reducing this randomness and exploiting knowledge of the target distribution.
- Hamiltonian Monte Carlo (or Hybrid Monte Carlo, HMC) is another sampling method that is much more computationally costly than the others but its proposals are much more efficient.
- As a result, it doesn't need as many samples to describe the posterior distribution.
- And as models become more complex (thousands or tens of thousands of parameters) HMC can really outshine other algorithms.

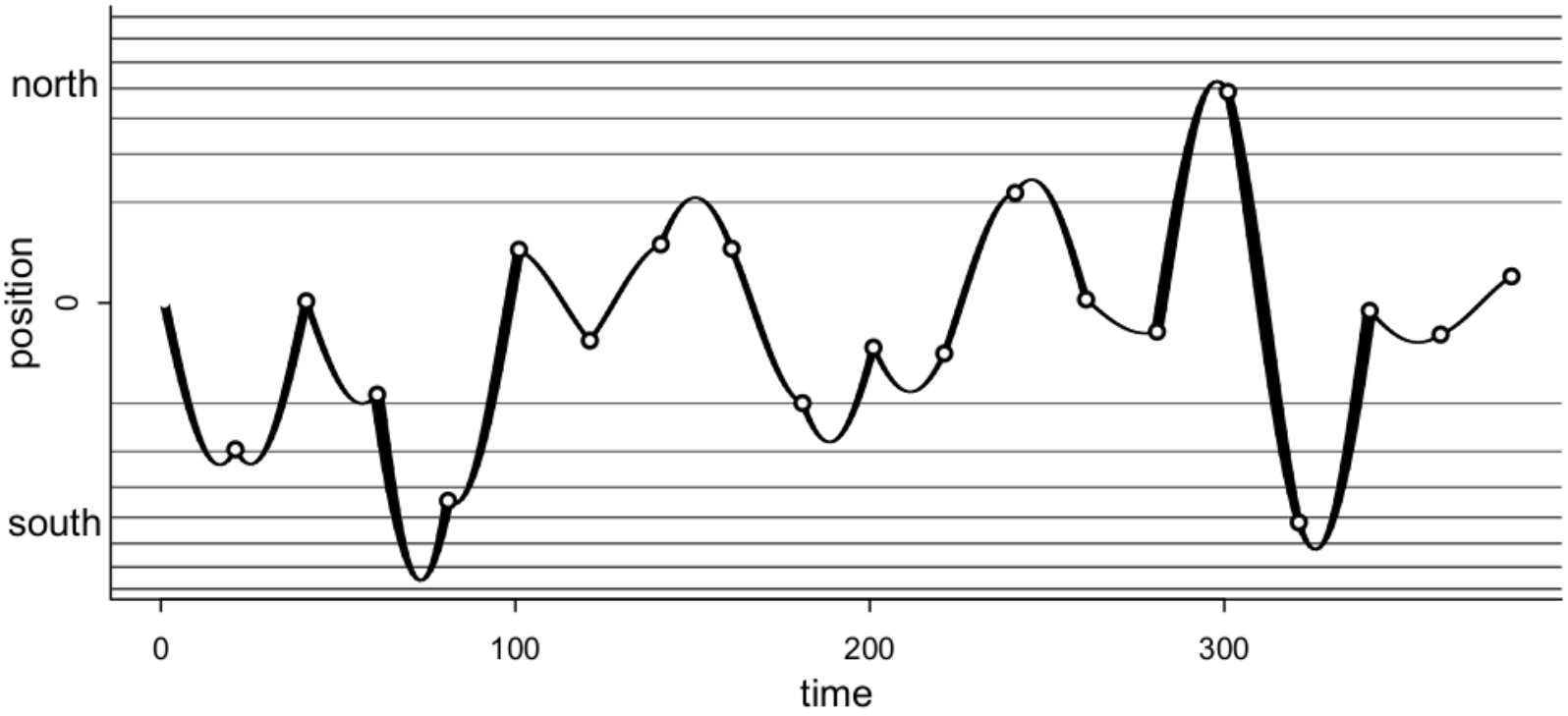
Hamiltonian Monte Carlo

- HMC is a very complex algorithm and we won't get into the details of its inner workings
- Let's try to understand it in a very superficial way by using again the politician's tale.
- Suppose the politician has moved to the mainland now.
- Now, instead of moving over a set of discrete islands, it has to move through a continuous territory stretched out along a narrow valley, running north-south.
- The obligations are the same: to visit his citizens in proportion to their local density.
- And again, the politician doesn't know the population of each area in advance.

Hamiltonian Monte Carlo

- The strategy of the politician is the following:
- He drives his car across the narrow valley back and forth along its length.
- In order to spend more time in densely settled areas, he slows down his vehicle when houses grow more dense.
- Likewise, he speeds up when houses grow more sparse.
- This strategy requires knowing how quickly population density is changing, at their current location.
- But it doesn't require remembering where they've been or knowing the population distribution anyplace else.
- This story is analogous to how Hamiltonian Monte Carlo works.

Hamiltonian Monte Carlo



Hamiltonian Monte Carlo

- In statistical applications, the politician's vehicle is the current vector of parameter values.
- HMC really does run a physics simulation, pretending the vector of parameters gives the position of a little frictionless particle.
- The log-posterior provides a surface for this particle to glide across.
- Then the job is to sweep across this surface, adjusting speed in proportion to how high up we are.
- When the log-posterior is very flat, then the particle can glide for a long time before the slope (gradient) makes it turn around.
- When instead the log-posterior is very steep, then the particle doesn't get far before turning around.

Hamiltonian Monte Carlo

- A big limitation of HMC is that it needs to be tuned to a particular model and its data.
- Stan[1] is a very popular platform for statistical modeling and high-performance statistical computation.
- Stan automates much of that tuning.
- Next, we will see how to use Stan to fit a linear model with an interaction.
- Make sure that the **rstan** package is installed to access Stan from within R.

[1] https://mc-stan.org/

Easy HMC: ulam

- Stan provides its own language to delcare models.
- The rethinking package provides a convenient interface, **ulam**, to compile lists of formulas, like the lists we've been using so far to construct quap estimates, into Stan HMC code.
- All that ulam does is translate formulas into Stan models, and then Stan defines the sampler and does the hard part.
- Stan models look very similar, but require some more explicit definitions.
- We will show them later.
- Before using ulam we must preprocess any variable transformations (log, exp, etc).
- We also must construct a clean data list with only the variables we will use.

A model of ruggedness

- We are are going to work with the **rugged** dataset from the rethinking package.
- Each row in these data is a country, and the various columns are economic, geographic, and historical features.
- The variable rugged is a Terrain Ruggedness Index that quantifies the topographic heterogeneity of a landscape.
- The outcome variable of our regression model will be the logarithm of real gross domestic product per capita, from the year 2000, **rgdppc_2000**.
- Ruggedness is usually associated with poorer countries, in most of the world.
- Rugged terrain usually means transport is difficult, which means market access is hampered, which means reduced gross domestic product.
- However, this associated gets inverted for African countries.

A model of ruggedness

- Let's load and explore the data:

```
library(rethinking)
data(rugged)
d <- rugged
d$log_gdp <- log(d$rgdppc_2000)
#remove rows with missing values
dd <- d[ complete.cases(d$rgdppc_2000) , ]
# discard columns we are not going to use
dd.trim <- dd[ , c("log_gdp","rugged","cont_africa") ]
summary(dd.trim)
```

```
> summary(dd.trim)
    log_gdp          rugged        cont_africa
 Min.   : 6.146   Min.   :0.0030   Min.   :0.0000
 1st Qu.: 7.539   1st Qu.:0.4422   1st Qu.:0.0000
 Median : 8.578   Median :0.9795   Median :0.0000
 Mean   : 8.517   Mean   :1.3332   Mean   :0.2882
 3rd Qu.: 9.480   3rd Qu.:1.9572   3rd Qu.:1.0000
 Max.   :10.965   Max.   :6.2020   Max.   :1.0000
```

- Note that we do not convert cont_africa into a factor because that produces numerical problems with the rethinking methods.

A model of ruggedness

- We have a dataset with countries as rows and 3 columns:
  1. log_gdp: the log GDP of the country in year 2000.
  2. rugged: the Terrain Ruggedness Index of the country
  3. cont_africa: a binary variable (1 for African countries and 0 for non-African ones).
- Let's compute the correlation between log_gdp and rugged:

```
> cor(dd.trim$rugged,dd.trim$log_gdp)
[1] 0.002833496
```

- It is very low. Now let's calculate them separately for African and non-African countries:

```
> dd.A<-dd.trim[dd.trim$cont_africa==1,]
> cor(dd.A$rugged,dd.A$log_gdp)
[1] 0.2607532
>
> dd.NA<-dd.trim[dd.trim$cont_africa==0,]
> cor(dd.NA$rugged,dd.NA$log_gdp)
[1] -0.2307945
```

- We now observe a stronger relationship between the two variables that is reversed for African and non-African countries.

A model of ruggedness

- We will now construct a Bayesian Linear Regression between log GDP (y) and terrain ruggedness (x) capable of incorporating the different relationships for these two groups of countries encoded by the variable A.
- As we learned in a previous lecture, interactions are a convenient way to include different slopes for different categories in our data.
- Our model will be as follows:

$$
\begin{aligned}
y_i &\sim N(\mu_i, \sigma) && \text{[likelihood]} \\
\mu_i &= \beta_0 + \beta_1 x_i + \beta_2 A_i + \beta_3 * x_i * A_i && \text{[linear model]} \\
\beta_0 &\sim N(0, 100) && [\beta_0 \text{ prior}] \\
\beta_1 &\sim N(0, 10) && [\beta_1 \text{ prior}] \\
\beta_2 &\sim N(0, 10) && [\beta_2 \text{ prior}] \\
\beta_3 &\sim N(0, 10) && [\beta_3 \text{ prior}] \\
\sigma &\sim \text{Cauchy}(0, 2) && [\sigma \text{ prior}]
\end{aligned}
$$

- We put Gaussian priors on all β coefficients and a Cauchy prior for σ (a t-student with 1 degree of freedom).
- The Cauchy distribution is heavy-tailed and has proven to be a useful prior for standard deviations (especially when restrained to postive values, taking the name of half-Cauchy).

A model of ruggedness

- The R code for the model would be:

```
model<-alist(
  log_gdp ~ dnorm( mu , sigma ) ,
  mu <- b0 + b1*rugged + b2*cont_africa
  + b3*rugged*cont_africa,
  b0 ~ dnorm(0,100),
  b1 ~ dnorm(0,10),
  b2 ~ dnorm(0,10),
  b3 ~ dnorm(0,10),
  sigma ~ dcauchy(0,2)
)
```

- Let's try to fit this model using Laplace Approximation:

```
> b.reg3<-quap(model,data=dd.trim)
Error in quap(model, data = dd.trim) :
  initial value in 'vmmin' is not finite
The start values for the parameters were invalid.
```

- The optimizer is having a hard time finding the MAP estimates for this model!
- This can sometimes be corrected by using start values (declared with the **start** parameter), but in many models even this does not help.

A model of ruggedness

- After trying to fit b.reg3 with quap many times, it worked!

```
> precis(b.reg3)
       mean   sd  2.5% 97.5%
b0     9.22 0.14  8.95  9.49
b1    -0.20 0.08 -0.35 -0.05
b2    -1.95 0.22 -2.39 -1.51
b3     0.39 0.13  0.14  0.65
sigma  0.93 0.05  0.83  1.03
```

- Let's interpret the mean values for the posterior marginals of each β coefficient:
  - mean(b0) = 9.22: this is the intercept for non-African countries.
  - mean(b1)=-0.20: this is the slope for non-African countries, which is clearly negative.
  - mean(b2)=-1.95: this is the difference between the intercept for African and non-African countries, which indicates that African countries have a lower GDP than non-African countries when rugedness is zero.
  - mean(b3)=0.39: this is the difference between the slopes for African and non-African countries, which indicates that the slope for non-African countries is positive (0.39-0.2).

Ulam

- Now, let's fit the same model using Stan HMC with **ulam**:

```
m.reg1 <- ulam(model ,data=dd.trim)
```

- After messages about compiling, and sampling, ulam returns an object that contains a bunch of summary information, as well as samples from the posterior distribution.
- This process takes much longer than our previous quap estimates.
- We can summarize just like with quap:

```
> precis(m.reg1, prob=0.95 )
       mean   sd  2.5% 97.5% n_eff Rhat4
b0     9.21 0.15  8.89  9.51   169  1.03
b1    -0.20 0.09 -0.37 -0.01   163  1.04
b2    -1.95 0.23 -2.37 -1.48   241  1.01
b3     0.39 0.13  0.10  0.66   243  1.03
sigma  0.95 0.05  0.85  1.06   348  1.00
```

- These estimates are very similar to the Laplace approximation.

Ulam

- A report of the chain can be obtained with command **show**:

```
> show(m.reg1)
Hamiltonian Monte Carlo approximation
500 samples from 1 chain

Sampling durations (seconds):
        warmup sample total
chain:1    0.2   0.17  0.37

Formula:
log_gdp ~ dnorm(mu, sigma)
mu <- b0 + b1 * rugged + b2 * cont_africa +
b3 * rugged * cont_africa
b0 ~ dnorm(0, 100)
b1 ~ dnorm(0, 10)
b2 ~ dnorm(0, 10)
b3 ~ dnorm(0, 10)
sigma ~ dcauchy(0, 2)
```

Stan Code

We can also get the Stan code with: `stancode(m.reg1)`:

```
data{
    vector[170] log_gdp;
    int cont_africa[170];
    vector[170] rugged;
}
parameters{
    real b0;
    real b1;
    real b2;
    real b3;
    real sigma;
}
```

Ulam

```
model{
    vector[170] mu;
    sigma ~ cauchy( 0 , 2 );
    b3 ~ normal( 0 , 10 );
    b2 ~ normal( 0 , 10 );
    b1 ~ normal( 0 , 10 );
    b0 ~ normal( 0 , 100 );
    for ( i in 1:170 ) {
        mu[i] = b0 + b1 * rugged[i] + b2 * cont_africa[i]
        + b3 * rugged[i] * cont_africa[i];
    }
    log_gdp ~ normal( mu , sigma );
}
```

This is Stan code, not R code. It is essentially the formula list we provided to ulam , but in reverse order.

Additional Ulam Parameters

- **iter**: the number of samples from the chain. The default is 1000.
- **warmup**: the number of tuning samples. These samples are used to adapt sampling, and so are not actually part of the target posterior distribution. The default value is iter/2 , which gives us 500 warmup samples and 500 real samples to use for inference.
- **chains**: the number of independent Markov chains to sample from. All of the non-warmup samples from each chain will be automatically combined in the resulting inferences.
- **cores**: the number of procesors over which the chains will be distributed.

Ulam

- Let's fit the same model sampling 4 different chains distributed over 4 CPU cores with 3000 iterarions and 1000 warmups.

```
> m.reg2 <- ulam(model ,data=dd.trim, iter=3000,
                 warmup =1000, chains=4, cores=4)
> show(m.reg2)
Hamiltonian Monte Carlo approximation
8000 samples from 4 chains

Sampling durations (seconds):
         warmup sample total
chain:1   0.62   1.08  1.70
chain:2   0.54   1.06  1.60
chain:3   0.60   0.89  1.49
chain:4   0.37   0.82  1.19
```

Diagnostic criteria: n_eff and Rhat

- Note that **precis** reported two new columns: n_eff and Rhat, which provide MCMC diagnostic criteria.
- They help us tell how well estimation worked.
- The column n_eff is a crude estimate of the number of independent samples we managed to get.
- When n_eff is much lower than the actual number of iterations (minus warmup) of your chains, it means the chains are inefficient, but possibly still okay.
- Rhat is the Gelman Rubin convergence diagnostic, which estimates the convergence of the Markov chains to the target distribution.
- It should approach 1.00 from above, when all is well.
- When Rhat is above 1.00, it usually indicates that the chain has not yet converged, and probably we shouldn't trust the samples.
- If we draw more iterations, it could be fine, or it could never converge.

- We can extract samples from the posterior in the usual way:

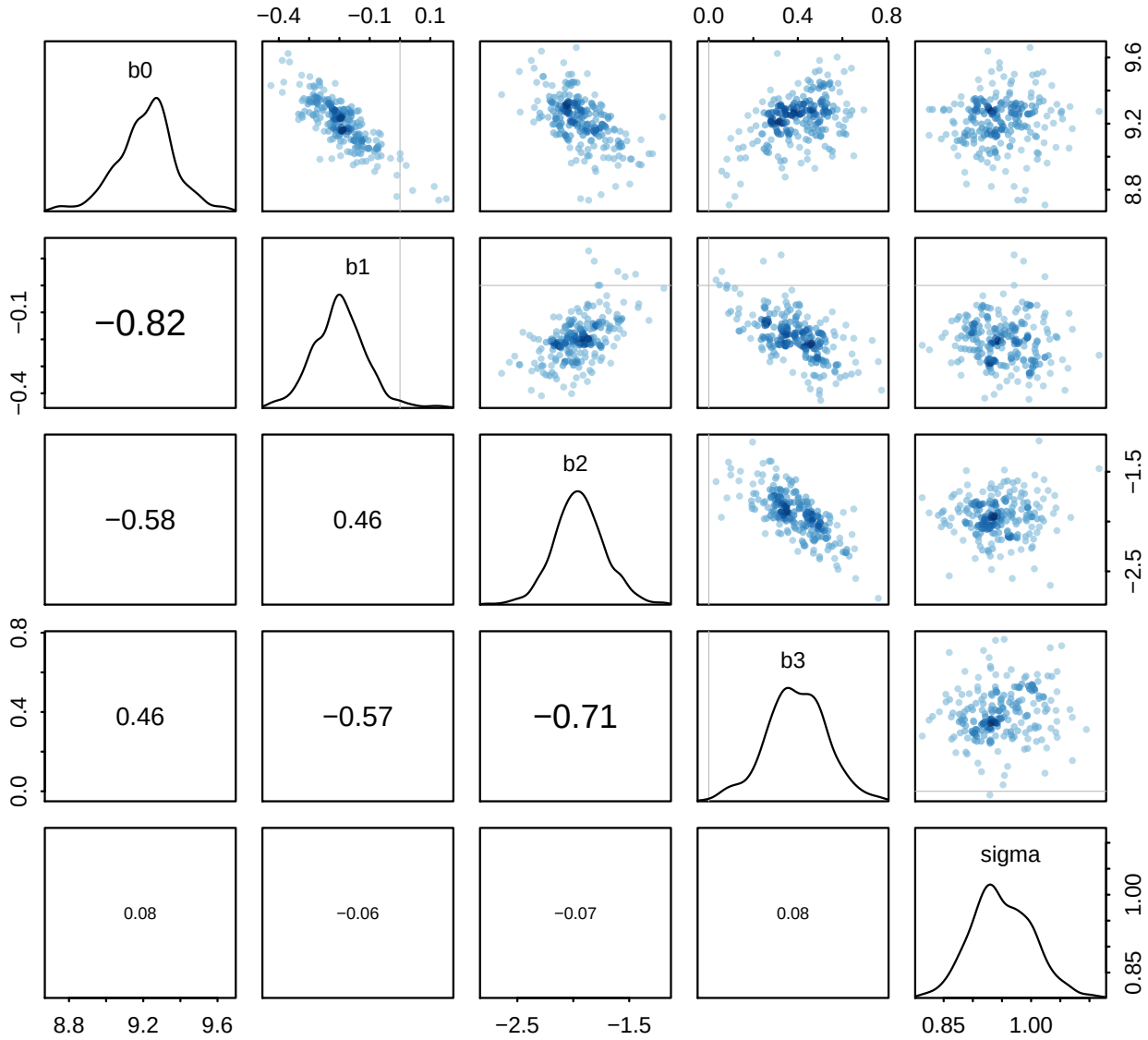
```
post <- extract.samples( m.reg1 )
```

- We also use **link** and **sim** functions to get posterior predictions.
- We can plot the correlations between samples using the function **pairs**:

```
pairs( m.reg1)
```

- This will show a pairs plot: a matrix of bivariate scatter plots.
- Along the diagonal the smoothed histogram of each parameter is shown, along with its name.
- In in the lower triangle of the matrix, below the diagonal, the correlation between each pair of parameters is shown, with stronger correlations indicated by relative size.

Ulam Pairs


b0
b1
b2
b3
sigma
−0.82
−0.58
0.46
0.46
−0.57
−0.71
0.08
−0.06
−0.07
0.08
−0.4
−0.1
0.1
0.0
0.4
0.8
9.6
9.2
8.8
−1.5
−2.5
0.8
0.4
0.0
1.00
0.85
8.8
9.2
9.6
−2.5
−1.5
0.85
1.00

Ulam Pairs

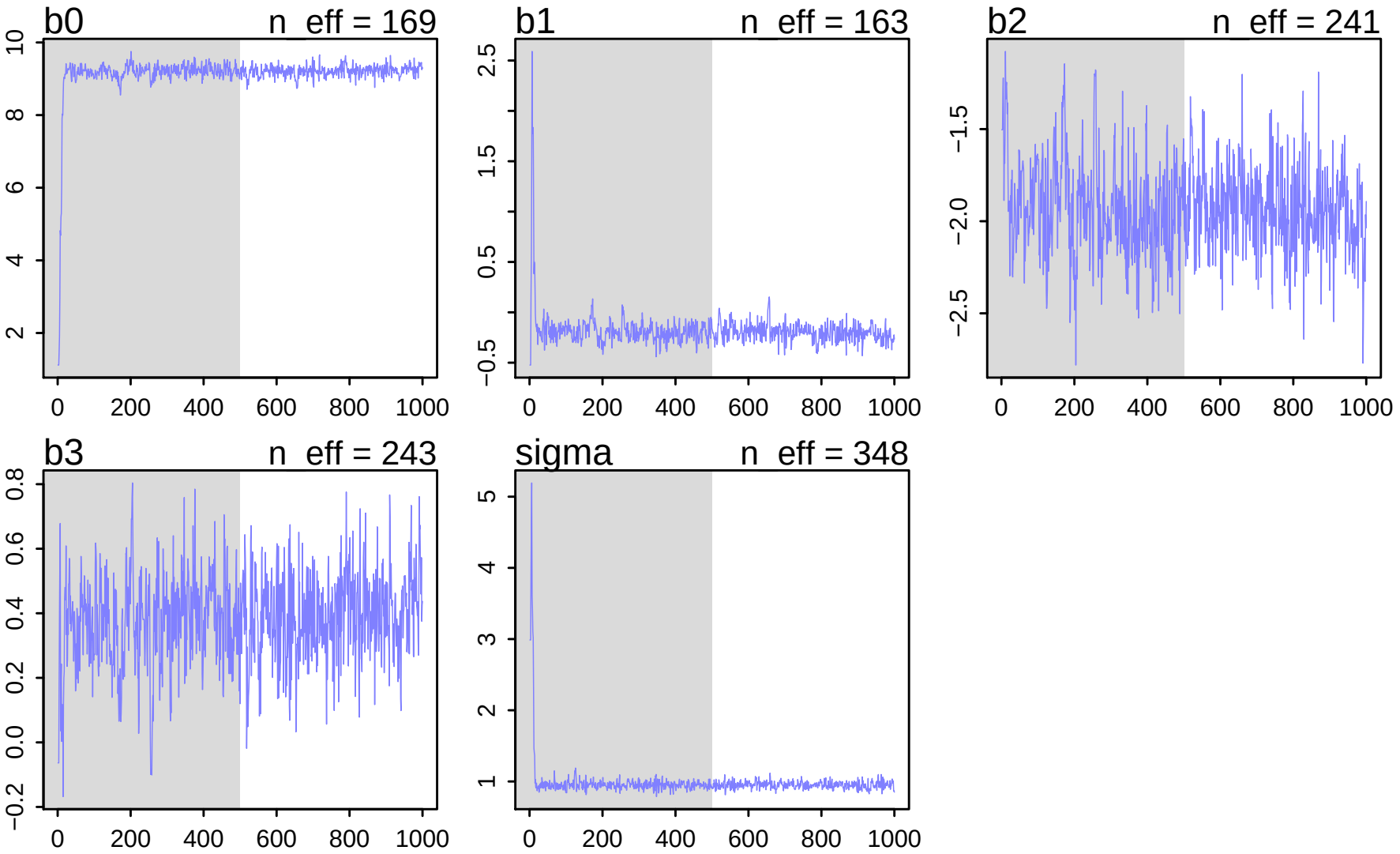
- For this model and these data, the resulting posterior distribution is quite nearly multivariate Gaussian.
- The density for sigma is certainly skewed in the expected direction.
- But otherwise the Laplace approximation does almost as well as Hamiltonian Monte Carlo.
- This is a very simple kind of model structure of course, with Gaussian priors, so an approximately Gaussian posterior should be no surprise.
- For more complex models, posterior distributions can take more exotic shapes.

Checking the Chain

- If the Markov chain is defined correctly then it is guaranteed to converge in the long run to the right posterior distribution.
- But the machine can sometimes fail.
- We won't study these cases in detail.
- A trace plot is a very useful tool for diagnosing malfunction.
- It basically plots the samples in sequential order, joined by a line.
- In the terrain ruggedness example, the trace plot shows a very healthy chain.
- We can obtain it with the following command:

```
traceplot( m.reg1 )
```

Trace Plot



Checking the Chain

- The plot shows the zig-zagging trace of each parameter as the path the chain took through each dimension of parameter space.
- The gray region in each plot, the first 500 samples, marks the warmups samples.
- During adaptation, the Markov chain is learning to more efficiently sample from the posterior distribution.
- So these samples are not necessarily reliable to use for inference.
- They are automatically discarded by **extract.samples** , which returns only the samples shown in the white regions.
- Typically we look for two things in these trace plots: stationarity and good mixing.

Checking the Chain

- Stationarity refers to the path staying within the posterior distribution.
- Notice that these traces, all stick around a very stable central tendency.
- Another way to think of this is that the mean value of the chain is quite stable from beginning to end.
- A well-mixing chain means that each successive sample within each parameter is not highly correlated with the sample before it, which is known as **autocorrelation**.
- Visually, we can see this by the rapid zig-zag motion of each path, as the trace traverses the posterior distribution without getting mired anyplace.

Conclusions

- This class has been an introduction to Markov chain Monte Carlo (MCMC) estimation.
- The goal has been to introduce the purpose and approach MCMC algorithms.
- The major algorithms introduced were the Metropolis, Gibbs sampling, and Hamiltonian Monte Carlo algorithms.
- Each has its advantages and disadvantages.
- A function in the rethinking package, ulam , was introduced that uses the Stan Hamiltonian Monte Carlo engine to a linear model with an interaction.

References I


Kruschke, J. (2014).
Doing bayesian data analysis: A tutorial with r, jags, and stan.


McElreath, R. (2020).
Statistical rethinking: A Bayesian course with examples in R and Stan.
CRC press.


Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953).
Equation of state calculations by fast computing machines.
The journal of chemical physics, 21(6):1087–1092.


Wikipedia (2021).
Markov chain Monte Carlo — Wikipedia, the free encyclopedia.
http://en.wikipedia.org/w/index.php?title=Markov%20chain%20Monte%20Carlo&oldid=1027048003.
[Online; accessed 01-July-2021].