9. Chapter 9 Solutions

9E1. Only (3) is required.

9E2. Gibbs sampling requires that we use special priors that are *conjugate* with the likelihood. This means that holding all the other parameters constant, it is possible to derive analytical solutions for the posterior distribution of each parameter. These conditional distributions are used to make smart proposals for jumps in the Markov chain. Gibbs sampling is limited both by the necessity to use conjugate priors, as well as its tendency to get stuck in small regions of the posterior when the posterior distribution has either highly correlated parameters or high dimension.

9E3. Hamiltonian Monte Carlo cannot handle discrete parameters. This is because it requires a smooth surface to glide its imaginary particle over while sampling from the posterior distribution.

9E4. The effect number of samples `n_eff` is an estimate of the number of completely independent samples that would hold equivalent information about the posterior distribution. It is always smaller than the actual number of samples, because samples from a Markov chain tend to sequentially correlated or *autocorrelated*. As autocorrelation rises, `n_eff` gets smaller. At the limit of perfect autocorrelation, for example, all samples would have the same value and `n_eff` would be equal to 1, no matter the actual number of samples drawn.

9E5. `Rhat` should approach 1. How close should it get? People disagree, but it is common to judge that any value less than 1.1 indicates convergence. But like all heuristic indicators, `Rhat` can be fooled.

9E6. A healthy Markov chain should be both *stationary* and *well-mixing*. The first is necessary for inference. The second is desirable, because it means the chain is more efficient. A chain that is both of these things should resemble horizontal noise.

A chain that is malfunctioning, as the problem asks, would not be stationary. THis means it is not converging to the target distribution, the posterior distribution. Examples were provided in the chapter. A virtue of Hamiltonian Monte Carlo is that it makes such chains very obvious: they tend to be rather flat wandering trends. Sometimes they are perfectly flat.

The best test of convergence is always to compare multiple chains. So the best sketch of a malfunctioning trace plot would be one that shows multiple chains wandering into different regions of the parameter space. Figure 8.7 in the chapter, left side, provides an example.

9M1. Here is the model again, now using a uniform prior for `sigma`:

R code 9.1

```
# load and rep data
library(rethinking)
data(rugged)
d <- rugged
d$log_gdp <- log(d$rgdppc_2000)
dd <- d[ complete.cases(d$rgdppc_2000) , ]
```

```
dd$log_gdp_std <- dd$log_gdp / mean(dd$log_gdp)
dd$rugged_std <- dd$rugged / max(dd$rugged)
dd$cid <- ifelse( dd$cont_africa==1 , 1 , 2 )

dat_slim <- list(
    log_gdp_std = dd$log_gdp_std,
    rugged_std = dd$rugged_std,
    cid = as.integer( dd$cid ) )

# new model with uniform prior on sigma
m9.1_unif <- ulam(
    alist(
        log_gdp_std ~ dnorm( mu , sigma ) ,
        mu <- a[cid] + b[cid]*( rugged_std - 0.215 ) ,
        a[cid] ~ dnorm( 1 , 0.1 ) ,
        b[cid] ~ dnorm( 0 , 0.3 ) ,
        sigma ~ dunif( 0 , 1 )
    ) , data=dat_slim , chains=4 , cores=4 )
```

And here's the model with an exponential prior on `sigma`:

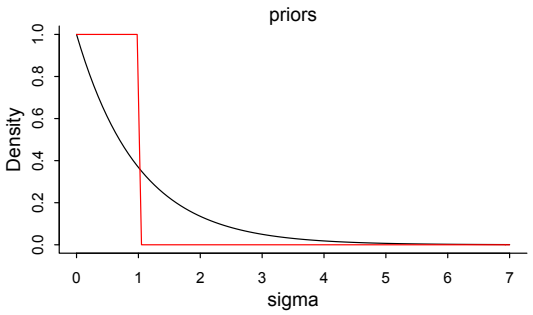
R code 9.2

```
m9.1_exp <- ulam(
    alist(
        log_gdp_std ~ dnorm( mu , sigma ) ,
        mu <- a[cid] + b[cid]*( rugged_std - 0.215 ) ,
        a[cid] ~ dnorm( 1 , 0.1 ) ,
        b[cid] ~ dnorm( 0 , 0.3 ) ,
        sigma ~ dexp( 1 )
    ) , data=dat_slim , chains=4 , cores=4 )
```

Before we look at the posterior distributions for each model, it may help to visualize each prior. So let's do that first. Maybe the easiest way is to draw random samples from each prior distribution and plot densities. But I'll use the `curve` function instead, just to provide an example of its use.

R code 9.3

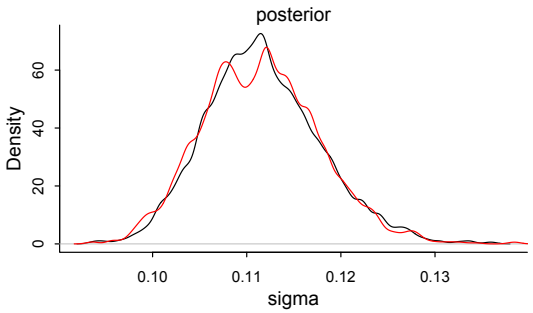
```
curve( dexp(x,1) , from=0 , to=7 ,
    xlab="sigma" , ylab="Density" , ylim=c(0,1) )
curve( dunif(x,0,1) , add=TRUE , col="red" )
mtext( "priors" )
```



Now let's compare the posterior distributions of `sigma` for both models.

R code 9.4

```
post <- extract.samples( m9.1_exp )
dens( post$sigma , xlab="sigma" )
post <- extract.samples( m9.1_unif )
dens( post$sigma , add=TRUE , col="red" )
mtext( "posterior" )
```



The posterior distributions are almost identical. Why? Because there is a lot of data to inform `sigma`. Can you find a prior that won't wash out?

9M2. The model code is easy enough (using the same data as in the previous problem):

R code 9.5

```
m9M2 <- ulam(
    alist(
        log_gdp_std ~ dnorm( mu , sigma ) ,
        mu <- a[cid] + b[cid]*( rugged_std - 0.215 ) ,
        a[cid] ~ dnorm( 1 , 0.1 ) ,
        b[cid] ~ dexp( 0.3 ) ,
        sigma ~ dexp( 1 )
    ) , data=dat_slim , chains=4 , cores=4 )
precis( m9M2 , 2 )
```

```
      mean   sd 5.5% 94.5% n_eff Rhat4
a[1]  0.89 0.02 0.86  0.91  1519     1
a[2]  1.05 0.01 1.03  1.07  1435     1
b[1]  0.15 0.07 0.03  0.27  1201     1
b[2]  0.02 0.02 0.00  0.05  1779     1
sigma 0.11 0.01 0.10  0.12  1344     1
```

The big difference here is `b[2]`. In the original model, the mass of this parameter is almost entirely below zero. Now it cannot go below zero, because the prior is not defined below zero. So instead the mass presses up against zero tightly.

9M3. You could use almost any model from the chapter. But I'll use the terrain ruggedness model again, since it was used in the other problems so far. I'll fix it at 1000 post-warmup samples for the sake of comparison. Then I'll compare a range of warmup values.

Here's some code to automate it all. You can do the same thing the hard way, just recoding the model each time. But it's a lot faster to compile the model once and reuse it.

R code 9.6

```
# compile model
# use fixed start values for comparability of runs
start <- list(a=c(1,1),b=c(0,0),sigma=1)
m9.1 <- ulam(
    alist(
        log_gdp_std ~ dnorm( mu , sigma ) ,
        mu <- a[cid] + b[cid]*( rugged_std - 0.215 ) ,
        a[cid] ~ dnorm( 1 , 0.1 ) ,
        b[cid] ~ dnorm( 0 , 0.3 ) ,
        sigma ~ dexp( 1 )
    ) , start=start , data=dat_slim , chains=1 , iter=1 )

# define warmup values to run through
warm_list <- c(5,10,100,500,1000)

# first make matrix to hold n_eff results
n_eff <- matrix( NA , nrow=length(warm_list) , ncol=5 )

# loop over warm_list and collect n_eff
for ( i in 1:length(warm_list) ) {
    w <- warm_list[i]
    m_temp <- ulam( m9.1 , chains=1 ,
        iter=1000+w , warmup=w , refresh=-1 , start=start )
    n_eff[i,] <- precis(m_temp,2)$n_eff
}

# add some names to rows and cols of result
# just to make it pretty
# there's always time to make data pretty
colnames(n_eff) <- rownames(precis(m_temp,2))
rownames(n_eff) <- warm_list
```

If you inspect the matrix `n_eff` now, you'll see parameters in columns and warmup values in rows:

```
       a[1]   a[2]   b[1]   b[2]  sigma
5     101.8   59.2    7.4   23.8   30.0
10   1247.2 1224.3  675.5  901.8  781.0
100  1404.6 1094.2  511.9  752.9  761.5
500  1940.5  962.6 1288.4  901.1 1199.9
1000 1072.3 1689.1 1905.9 1340.3 1295.6
```

So you can see that for this model and this data, very little warmup is needed. Basically anything more than 10 is the same. This isn't always going to be true, however. For the more complicated non-linear models that your'll meet in later chapters, and especially multilevel models, more warmup is often helpful. How much? It depends, unfortunately. Models and data are just too diverse to give useful general advice here.

But I can show you a common situation that benefits from more warmup. Let's go back to the leg data example from Chapter 6:

R code 9.7

```
N <- 100                          # number of individuals
height <- rnorm(N,10,2)           # sim total height of each
leg_prop <- runif(N,0.4,0.5)      # leg as proportion of height
leg_left <- leg_prop*height +     # sim left leg as proportion + error
    rnorm( N , 0 , 0.02 )
```

```
leg_right <- leg_prop*height +    # sim right leg as proportion + error
    rnorm( N , 0 , 0.02 )
                                  # combine into data frame
d <- data.frame(height,leg_left,leg_right)
```

And now to run these data through the same process, using the model from Chapter 5 as well:

R code 9.8

```
m <- ulam(
    alist(
        height ~ dnorm( mu , sigma ) ,
        mu <- a + bl*leg_left + br*leg_right ,
        a ~ dnorm( 10 , 100 ) ,
        bl ~ dnorm( 2 , 10 ) ,
        br ~ dnorm( 2 , 10 ) ,
        sigma ~ dunif( 0 , 10 )
    ) , data=d , iter=1 )

warm_list <- c(5,10,100,500,1000)
n_eff <- matrix( NA , nrow=length(warm_list) , ncol=4 )
for ( i in 1:length(warm_list) ) {
    w <- warm_list[i]
    m_temp <- ulam( m , chains=1 ,
        iter=1000+w , warmup=w , refresh=-1 )
    n_eff[i,] <- precis(m_temp,2)$n_eff
}
colnames(n_eff) <- rownames(precis(m_temp,2))
rownames(n_eff) <- warm_list
round(n_eff,1)
```

```
         a    bl    br sigma
5      8.3   3.2   3.0  54.9
10    65.7   6.7   6.7  70.0
100  515.9  92.2  92.5 676.3
500  538.6 292.8 295.0 552.4
1000 488.4 306.5 307.4 567.7
```

This time, even with a simpler model and fewer observations, it took longer for the benefits of more warmup to taper off. Efficiency improved all the way up to 1000 warmup steps, at least for the parameters `bl` and `br`. Why? Because in this posterior, the parameters `bl` and `br` are highly correlated, recall. Take a look at `pairs(m_temp)` for example. This makes it harder to sample efficiently from the posterior.

In complex multilevel models, there are nearly always batches of highly correlated parameters. So being conservative about warmup often pays off.

9H1. What this code does is sample from the priors. There is no likelihood, and that's okay. The posterior distribution is then just a merger of the priors. What is tricky about this problem though is that the Cauchy prior for the parameter `b` will not produce the kind of trace plot you might expect from a good Markov chain. This is because Cauchy is a very long tailed distribution, so it'll occasionally make distance leaps out into the tail. We'll look at the `precis` output, then the trace plot, so you can see what I mean.

Run the provided code, then inspect the marginal posterior:

R code 9.9

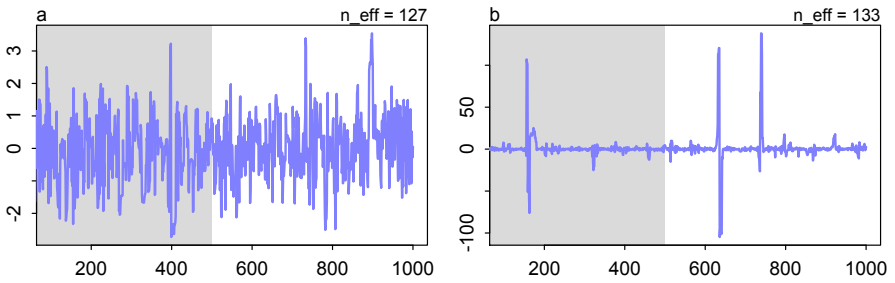
```
precis(mp)
```

```
  mean    sd  5.5% 94.5% n_eff Rhat4
a 0.05  0.97 -1.38  1.50   127  1.01
b 1.01 19.09 -5.83  8.03   133  1.00
```

Now let's look at the trace plot, which is what has alarmed students in the past, when I have assigned this problem as homework:

R code 9.10

```
traceplot( mp , n_col=2 , lwd=2 )
```



The trace plot might look a little weird to you, because the trace for `b` has some big spikes in it. That's how a Cauchy behaves, though. It has thick tails, so needs to occasionally sample way out. The trace plot for `a` is typical Gaussian in shape. Since the posterior distribution does often tend towards Gaussian for many parameters, it's possible to get too used to expecting every trace to look like the one on the left. But you have to think about the influence of priors in this case. The trace on the right is just fine.

9H2. First, load and prepare the data.

R code 9.11

```
library(rethinking)
data(WaffleDivorce)
d <- WaffleDivorce
d$D <- standardize( d$Divorce )
d$M <- standardize( d$Marriage )
d$A <- standardize( d$MedianAgeMarriage )
d_trim <- list(D=d$D,M=d$M,A=d$A)
```

Now to fit the models over again, this time using `ulam`. Note that we need to add `log_lik=TRUE` to get the terms needed to compute PSIS or WAIC.

R code 9.12

```
m5.1_stan <- ulam(
    alist(
        D ~ dnorm( mu , sigma ) ,
        mu <- a + bA * A ,
        a ~ dnorm( 0 , 0.2 ) ,
        bA ~ dnorm( 0 , 0.5 ) ,
        sigma ~ dexp( 1 )
    ) , data=d_trim , chains=4 , cores=4 , log_lik=TRUE )
m5.2_stan <- ulam(
    alist(
```

```
        D ~ dnorm( mu , sigma ) ,
        mu <- a + bM * M ,
        a ~ dnorm( 0 , 0.2 ) ,
        bM ~ dnorm( 0 , 0.5 ) ,
        sigma ~ dexp( 1 )
    ) , data=d_trim , chains=4 , cores=4 , log_lik=TRUE )
m5.3_stan <- ulam(
    alist(
        D ~ dnorm( mu , sigma ) ,
        mu <- a + bM*M + bA*A ,
        a ~ dnorm( 0 , 0.2 ) ,
        bM ~ dnorm( 0 , 0.5 ) ,
        bA ~ dnorm( 0 , 0.5 ) ,
        sigma ~ dexp( 1 )
    ) , data=d_trim , chains=4 , cores=4 , log_lik=TRUE )
```

Since you are possibly still getting used to Markov chains, I recommend checking the trace and trank plots for each model, as well as inspecting the `n_eff` and `Rhat` values for each parameter in each model.

Now to compare the models:

R code 9.13

```
compare( m5.1_stan , m5.2_stan , m5.3_stan , func=PSIS )
```

```
Some Pareto k values are high (>0.5)
           PSIS    SE dPSIS  dSE pPSIS weight
m5.1_stan 125.9 12.89   0.0   NA   3.7   0.72
m5.3_stan 127.8 13.00   1.8 0.70   4.8   0.28
m5.2_stan 139.3  9.94  13.4 9.33   3.0   0.00
```

And WAIC is quite similar:

R code 9.14

```
compare( m5.1_stan , m5.2_stan , m5.3_stan , func=WAIC )
```

```
           WAIC    SE dWAIC  dSE pWAIC weight
m5.1_stan 125.7 12.61   0.0   NA   3.6   0.71
m5.3_stan 127.5 12.73   1.8 0.70   4.7   0.29
m5.2_stan 139.2  9.81  13.5 9.17   2.9   0.00
```

The model with only age-at-marriage comes out on top, although the model with both predictors does nearly as well. In fact, the PSIS/WAIC of both models is nearly identical. I'd call this is a tie, because even though one model does a bit better than the other, the difference between them is of no consequence. How can we explain this? Well, look at the marginal posterior for `m5.3_stan`:

R code 9.15

```
precis(m5.3_stan)
```

```
       mean   sd  5.5% 94.5% n_eff Rhat4
a      0.00 0.10 -0.16  0.16  1561     1
bM    -0.06 0.15 -0.30  0.18  1012     1
bA    -0.61 0.15 -0.87 -0.36  1072     1
sigma  0.83 0.09  0.70  0.98  1563     1
```

While this model includes marriage rate as a predictor, it estimates very little expected influence for it, as well as substantial uncertainty about the direction of any influence it might have. So models `m5.3_stan` and `m5.1_stan` make practically the same predictions. After accounting for the larger penalty for `m5.3_stan`—4.7 instead of 3.7—the two models rank almost the same. This makes sense,

because you already learned back in Chapter 5 that marriage rate probably gets its correlation with divorce rate through a correlation with age at marriage. So even though including marriage rate in a model doesn't really aid in prediction, there is enough evidence here that the parameter `bR` can be estimated well enough, and including marriage rate doesn't hurt prediction either.

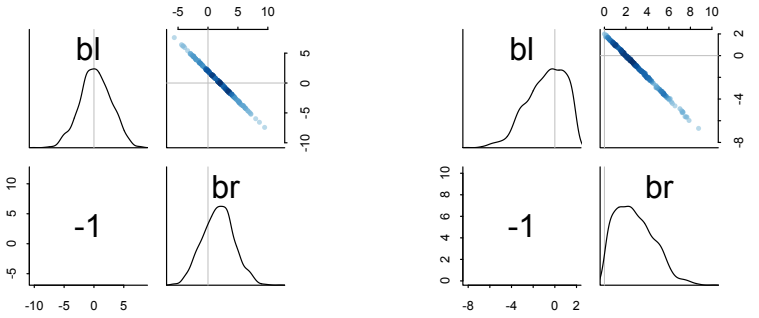
Or at least that's what PSIS/WAIC expects. Only the future will tell which model is actually better for forecasting. PSIS/WAIC is not an oracle. It's a golem.

9H3. Here is the simulation code again, just copied from the chapter:

R code 9.16

```
N <- 100                          # number of individuals
set.seed(909)
height <- rnorm(N,10,2)           # sim total height of each
leg_prop <- runif(N,0.4,0.5)      # leg as proportion of height
leg_left <- leg_prop*height +     # sim left leg as proportion + error
    rnorm( N , 0 , 0.02 )
leg_right <- leg_prop*height +    # sim right leg as proportion + error
    rnorm( N , 0 , 0.02 )
                                  # combine into data frame
d <- data.frame(height,leg_left,leg_right)
```

Run the `ulam` models in the prompt, and then we can look at the `pairs` plot for each model, which is probably the quickest way to see the impact of the change in prior. The posterior distributions for bl and br are symmetric, and perfectly negatively correlated, in `m5.8s`. In the other model, with the truncated prior, they are still perfectly negatively correlated, but now they look like mirror images, one being skewed left and the other right. See below (left: `m5.8s`; right: `m5.8s2`).



What has happened is that the posterior distributions are necessarily negatively correlated with one another. That arises because the left and right legs contain the same information. So this is the lack of identifiability thing returning. So when we change the prior on one of the parameters, that information has to cascade into the posterior distribution of the other parameter as well. Otherwise the negative posterior correlation wouldn't be maintained.

9H4. Inspecting WAIC for the two models, you get:

R code 9.17

```
compare( m5.8s , m5.8s2 )
```

```
        WAIC    SE dWAIC  dSE pWAIC weight
m5.8s2 194.2 11.27     0   NA   2.7   0.62
m5.8s  195.2 11.16     1 0.72   3.2   0.38
```

This is a good tie. You'll get slightly different numbers, on account of simulation variance. Note that the model with the truncated prior is less flexible, as indicated by `pWAIC`. Why? Because the prior is more informative, the variance in the posterior distribution is smaller. But the same model, `m5.8s2`, also fits the sample worse, as a consequence of the more informative prior.

9H5. To do as the problem asks, we'll need a new vector for the population sizes of the islands. Let's call it `pop_size` and give it randomly shuffled values from 1 to 10:

R code 9.18

```
pop_size <- sample( 1:10 )
```

Now most of the same code will work, but we'll need to extract two elements of `pop_size` when we construct the probability of moving. Why? Because its population size that matters when deciding to move, not each island's position. Now that size and position are not the same numbers, we just have use indexing to translate from position to population size. Here's how it works. The new code is at the end, where `prob_move` is calculated. And I've added the line above to the start, as well.

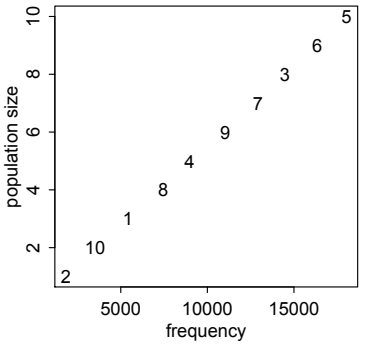
R code 9.19

```
num_weeks <- 1e5
positions <- rep(0,num_weeks)
pop_size <- sample( 1:10 )
current <- 10
for ( i in 1:num_weeks ) {
    # 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 <- pop_size[proposal]/pop_size[current]
    current <- ifelse( runif(1) < prob_move , proposal , current )
}
```

To verify this is working as intended, compare the frequency in the samples of each island to each island's population size:

R code 9.20

```
# compute frequencies
f <- table( positions )

# plot frequencies against relative population sizes
# label each point with island index
plot( as.vector(f) , pop_size , type="n" ,
    xlab="frequency" , ylab="population size" ) # empty plot
text(  , x=f , y=pop_size )
```



Each island appears in the samples in direct proportion to its population size. Run the code a few times and watch the index values shuffle around.

In a typical Metropolis context, remember, the island index values are parameter values and the population sizes are posterior probabilities.

9H6. Okay this is one of the hardest problems in the book. Why? Because it is asking a lot. There wasn't an example in the chapter of actually using the Metropolis algorithm with data and a model. So converting the silly island hopping example to a practical model fitting example is a huge challenge.

Really this problem is a way to sneak in this extra content, as actually writing Markov chains is beyond the scope of the book. But if you are curious, here's one solution.

Here's the raw data for the globe tossing example:

$$\text{W L W W W L W L W}$$

That's 6 waters in 9 tosses. The likelihood is binomial, and we'll use a uniform prior, as in Chapter 2. So this is the model:

$$w \sim \text{Binomial}(n, p)$$
$$p \sim \text{Uniform}(0, 1)$$

where w is the observed number of water and n is the number of globe tosses. The parameter p is the target of inference. We need a posterior distribution for it. The prior distribution is the given uniform density form zero to one.

To get a working Metropolis algorithm for this model, think of the different values of p as the island indexes and the product of the likelihood and prior as the population sizes. What is tricky here is that there an infinite number of islands: every continuous value that p can take from zero to one. That's hardly a problem, though. We just need a different way of proposing moves, so that we can land on a range of islands. It'll make more sense, once you see the code.

R code 9.21

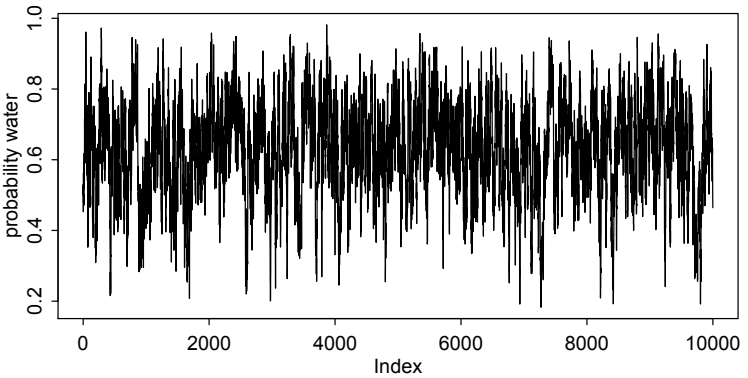
```
num_samples <- 1e4
p_samples <- rep(NA,num_samples)
p <- 0.5 # initialize chain with p=0.5
for ( i in 1:num_samples ) {
    # record current parameter value
    p_samples[i] <- p

    # generate a uniform proposal from -0.1 to +0.1
    proposal <- p + runif(1,-0.1,0.1)
    # now reflect off boundaries at 0 and 1
    # this is needed so proposals are symmetric
```

```
    if ( proposal < 0 ) proposal <- abs(proposal)
    if ( proposal > 1 ) proposal <- 1-(proposal-1)

    # compute posterior prob of current and proposal
    prob_current <- dbinom(6,size=9,prob=p) * dunif(p,0,1)
    prob_proposal <- dbinom(6,size=9,prob=proposal) * dunif(proposal,0,1)

    # move?
    prob_move <- prob_proposal/prob_current
    p <- ifelse( runif(1) < prob_move , proposal , p )
}
```

That's really all there is to it. Once the loop finishes—it'll be very very fast—take a look at the trace plot:

R code 9.22

```
plot( p_samples , type="l" , ylab="probability water" )
```

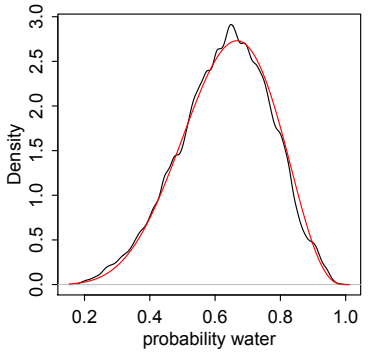


This chain is fine, but it is more autocorrelated than a typical Stan chain. That's why it tends to wander around a bit. It is stationary, but it isn't mixing very well. This is common of such a simple Metropolis chain. But it is working.

Now look at the posterior distribution implied by these samples. I'll also compare it to the analytically derived posterior for this model:

R code 9.23

```
dens( p_samples , xlab="probability water" )
curve( dbeta(x,7,4) , add=TRUE , col="red" )
```



Not bad.

So what would you do if the model had more than one parameter? You just add another proposal for each parameter, accepting or rejecting each proposal independent of the others. Suppose for example we want to do a simple linear regression with a Metropolis chain. Let's simulate some simple Gaussian data and then compute the posterior distribution of the mean and standard deviation using a custom Metropolis algorithm. The model is:

$$y_i \sim \text{Normal}(\mu, \sigma)$$
$$\mu \sim \text{Normal}(0, 10)$$
$$\sigma \sim \text{Uniform}(0, 10)$$

This is the code:

R code 9.24

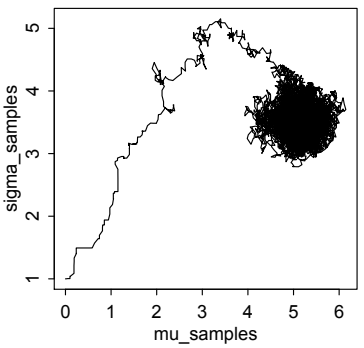
```
# simulate some data
# 100 observations with mean 5 and sd 3
y <- rnorm( 100 , 5 , 3 )

# now chain to sample from posterior
num_samples <- 1e4
mu_samples <- rep(NA,num_samples)
sigma_samples <- rep(NA,num_samples)
mu <- 0
sigma <- 1
for ( i in 1:num_samples ) {
    # record current parameter values
    mu_samples[i] <- mu
    sigma_samples[i] <- sigma

    # proposal for mu
    mu_prop <- mu + runif(1,-0.1,0.1)

    # compute posterior prob of mu and mu_prop
    # this is done treating sigma like a constant
    # will do calculations on log scale, as we should
    # so log priors get added to log likelihood
    log_prob_current <- sum(dnorm(y,mu,sigma,TRUE)) +
                        dnorm(mu,0,10,TRUE) + dunif(sigma,0,10,TRUE)
    log_prob_proposal <- sum(dnorm(y,mu_prop,sigma,TRUE)) +
                        dnorm(mu_prop,0,10,TRUE) + dunif(sigma,0,10,TRUE)
    # move?
    prob_move <- exp( log_prob_proposal - log_prob_current )
```

```
    mu <- ifelse( runif(1) < prob_move , mu_prop , mu )

    # proposal for sigma
    sigma_prop <- sigma + runif(1,-0.1,0.1)
    # reflect off boundary at zero
    if ( sigma_prop < 0 ) sigma_prop <- abs(sigma_prop)

    # compute posterior probabilities
    log_prob_current <- sum(dnorm(y,mu,sigma,TRUE)) +
                        dnorm(mu,0,10,TRUE) + dunif(sigma,0,10,TRUE)
    log_prob_proposal <- sum(dnorm(y,mu,sigma_prop,TRUE)) +
                        dnorm(mu,0,10,TRUE) + dunif(sigma_prop,0,10,TRUE)
    # move?
    prob_move <- exp( log_prob_proposal - log_prob_current )
    sigma <- ifelse( runif(1) < prob_move , sigma_prop , sigma )
}
```

This code is more complex, but it is really the same strategy, just with two internal steps, one for each parameter. You can process `mu_samples` and `sigma_samples` as usual. I'll show the sequential samples plotted together now, so you can see how the chain wanders into the high probability region of the posterior as the chain evolves:

R code 9.25

```
plot( mu_samples , sigma_samples , type="l" )
```



We initialized the chain at $(0, 1)$. It then quickly wandered up and to the right, eventually falling into the orbit of the high density region of the posterior. That long trail into the high density region is often called "burn in" and usually trimmed off before analysis. You don't have to do such trimming with Stan chains, because Stan's warm-up phase takes care of a similar task, and Stan only returns post-warmup samples to you.

9H7. Following the example in the box starting on page 276, we need to write functions for both the log-probability of the data, the `U` function, and its gradient, the `U_gradient` function.

The log-probability is nothing new. We did it in the previous problem. Now we'll build it into its own function:

R code 9.26

```
# y is successes, n is trials
U_globe <- function( q ) {
```

```
    U <- dbinom(y,size=n,prob=q,log=TRUE) + dunif(q,0,1,log=TRUE)
    return( -U )
}
```

The gradient requires some thought. We need to compute:

$$\frac{\partial U(y|n,p)}{\partial p}$$

where y is the observed count 6 and $n = 9$. This is made a bit easier by the fact that we used a uniform prior, which has a constant gradient. So we can omit the prior in this case. To get the derivative above, you can either aggressively apply the chain rule, or plug the binomial log-probability expression into a symbolic system like Mathematica and get the answer:

$$\frac{\partial \log U(y|n,p)}{\partial p} = \frac{y - np}{p(1-p)}$$

And this gives us the gradient function:

R code 9.27

```
U_globe_gradient <- function( q ) {
    G <- ( y - n*q )/( q*(1-q) )
    return( -G )
}
```

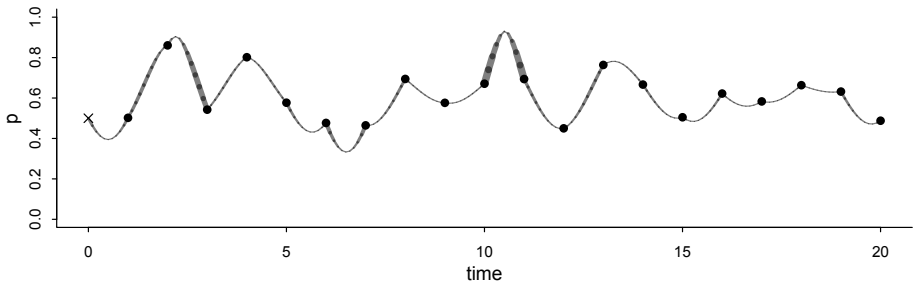
Now we can modify the code in Rcode 9.7 on page 277 to run the HMC simulation. This example has only one dimension, p, so instead of the fancy 2D plot of trajectories, I'll plot time on the horizontal axis and the position on the vertical. All this code does really is loop and call `HMC2`, feeding the result back in each time to get a new trajectory. The samples are stored in `samples`.

R code 9.28

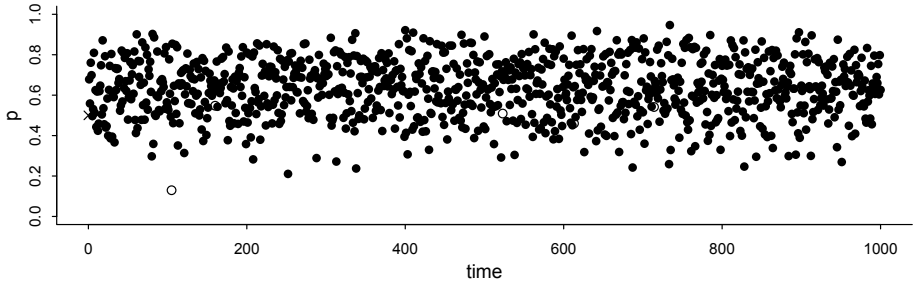
```
# data
y <- 6
n <- 9

# initialize everything
Q <- list()
Q$q <- 0.5
n_samples <- 20
plot( NULL , xlab="time" , ylab="p" , ylim=c(0,1) , xlim=c(0,n_samples) )
path_col <- col.alpha("black",0.5)
points( 0 , Q$q , pch=4 , col="black" )

step <- 0.03
L <- 10

samples <- rep(NA,n_samples)

show_trajectory <- TRUE
for ( i in 1:n_samples ) {
    Q <- HMC2( U_globe , U_globe_gradient , step , L , Q$q )
    # plot trajectory
    if ( show_trajectory==TRUE )
      for ( j in 1:L ) {
        K0 <- sum(Q$ptraj[j,]^2)/2 # kinetic energy
        tx <- (i-1) + c( 1/L*(j-1) , 1/L*j )
        lines( tx , Q$traj[j:(j+1),1] , col=path_col , lwd=1+2*K0 )
```

```
      }
    points( i , Q$traj[L+1,1] , pch=ifelse( Q$accept==1 , 16 , 1 ) )
    if ( Q$accept==1 ) samples[i] <- Q$q
}
```

Running this, you should get something like:



Increase the number of samples to, say, 1000 (you might want to disable plotting the trajectories):



The open points are divergent trajectories. They aren't stored in `samples`, but they do imply that we could tune the step size a bit to do better. See if you can tune the divergent trajectories out.