hw9

Christopher Huong

2024-04-02

9E3. Which sort of parameters can Hamiltonian Monte Carlo not handle? Can you explain why?

Hamiltonian Monte Carlo cannot handle very flat, uninformative priors. This is because The algorithm will waste time visiting highly unprobable parameter spaces, and be unlikely to converge.
It also cannot handle discrete/categorical variables because it needs a continuous surface for the simulated frictionless particle to glide on.

9E4. Explain the difference between the effective number of samples, n_eff as calculated by Stan, and the actual number of samples.

The actual number of samples is the specified chain length (minus the warmup) which is the number of positions in the parameter space visited by the particle (when it stops). The effective number of samples is the chain length if there were no lag-1 autocorrelation between samples (parameters were samples indepedently of each other). Autocorrelated samples deceases the effective sample, because they are less informative than independent samples (since some information of one sample is already contained in the previous sample).

9E5. Which value should Rhat approach, when a chain is sampling the posterior distribution correctly?

Rhat is the ratio of total variance to the average within-chain variance. So when the total variance of all the chains equals the average within-chain variance, Rhat=1, and the chains are converging on the same range of parameter space, which is a good sign.

9M1. Re-estimate the terrain ruggedness model from the chapter, but now using a uniform prior for the standard deviation, sigma.

The uniform prior should be dunif(0,1). Use ulam to estimate the posterior. Does the different prior have any detectible influence on the posterior distribution of sigma? Why or why not?

```
library(rethinking)
```

```
data(rugged); d <- rugged; rm(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) #mean = 1
dd$rugged_std <- dd$rugged / max(dd$rugged) #max = 1
dd$cid <- ifelse(dd$cont_africa==1, 1, 2)

d1 <- list(
  log_gdp_std = dd$log_gdp_std,
  rugged_std = dd$rugged_std,
  cid = dd$cid
)
```

```
# sigma ~ dexp(1)
m1 <- 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=d1, chains=4,cores=4
)
```

```
## Running MCMC with 4 parallel chains, with 1 thread(s) per chain...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 2 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 2 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 3 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 3 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 3 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 4 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 4 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 finished in 0.1 seconds.
## Chain 2 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 finished in 0.1 seconds.
## Chain 3 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 finished in 0.1 seconds.
## Chain 4 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 4 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 4 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 4 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 4 finished in 0.2 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 0.1 seconds.
## Total execution time: 0.5 seconds.
```

```
# sigma ~ dunif(0,1)
m2 <- 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=d1, chains=4,cores=4
)
```

```
## Running MCMC with 4 parallel chains, with 1 thread(s) per chain...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 2 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 2 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 2 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 3 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 3 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 3 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 3 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 4 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 4 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 4 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 4 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 4 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 4 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 0.1 seconds.
## Chain 2 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 finished in 0.1 seconds.
## Chain 3 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 finished in 0.1 seconds.
## Chain 4 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 4 finished in 0.1 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 0.1 seconds.
## Total execution time: 0.3 seconds.
```

Compare parameter estimates

```
precis(m1, 2)
```

```
##              mean          sd         5.5%       94.5%      rhat ess_bulk
## a[1]   0.8869862 0.015927655  0.862480920  0.91281142 1.0042403 2261.229
## a[2]   1.0503117 0.010260373  1.033870000  1.06638110 1.0020114 3019.585
## b[1]   0.1301127 0.076145837  0.007414524  0.25092813 0.9990398 2382.606
## b[2]  -0.1418390 0.054651282 -0.227190980 -0.05583373 0.9996225 2274.754
## sigma  0.1114912 0.006147795  0.102055745  0.12158215 1.0047775 2666.566
```

```
precis(m2, 2)
```

```
##              mean          sd        5.5%       94.5%      rhat ess_bulk
## a[1]   0.8861493 0.016290310  0.86043880  0.91339271 1.0025688 3136.547
## a[2]   1.0506312 0.010305896  1.03367945  1.06737110 1.0000889 2927.390
## b[1]   0.1306889 0.072870677  0.01211215  0.24569822 0.9998233 2247.984
## b[2]  -0.1415880 0.058231856 -0.23531336 -0.04618315 1.0017019 2178.522
## sigma  0.1116766 0.006348706  0.10191745  0.12223813 1.0016996 2309.057
```

No difference, the likelihood overwhelms the priors. Also dunif(0,1) has all of its probability density between 0 and 1, and I think dexp(1) has most of its mass between 0 and 1

```
fx <- function(x){ exp(1)^-x}
integrate(fx, lower=0, upper=1)
```

```
## 0.6321206 with absolute error < 7e-15
```

Yep, most of the probability density for the exponential distribution with lambda=1 is between x=0 and x=1

9M2. Modify the terrain ruggedness model again. This time, change the prior for b[cid] to dexp(0.3). What does this do to the posterior distribution? Can you explain it?

```r
m3 <- 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 ~ dunif(0, 1)
  ),data=d1, chains=4,cores=4
)
```

```
## Running MCMC with 4 parallel chains, with 1 thread(s) per chain...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 4 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 4 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 4 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 4 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 4 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 4 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 4 finished in 0.2 seconds.
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 2 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 2 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 2 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 3 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 3 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 3 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 3 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 0.3 seconds.
## Chain 2 finished in 0.3 seconds.
## Chain 3 finished in 0.3 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 0.3 seconds.
## Total execution time: 0.5 seconds.
```

```
precis(m3, 2)
```

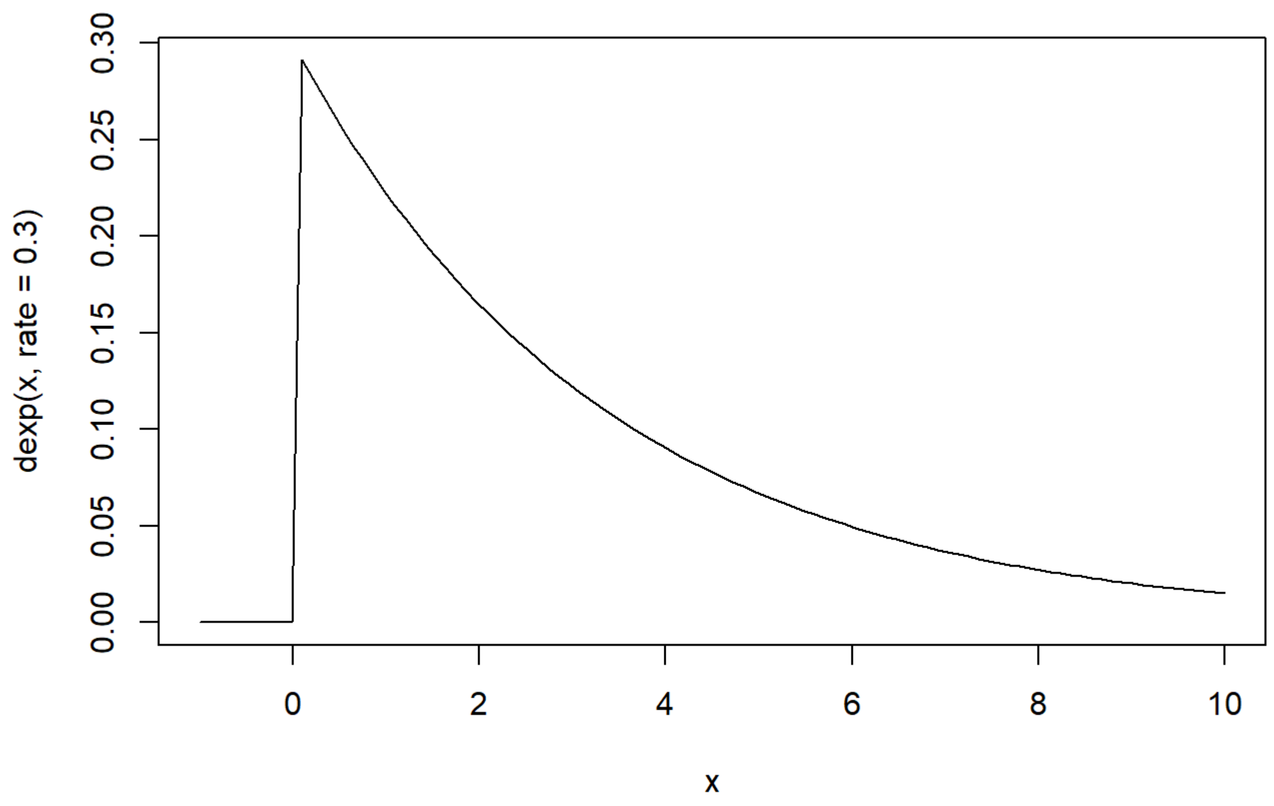
```
##             mean          sd        5.5%      94.5%      rhat ess_bulk
## a[1]  0.88708920 0.016833881 0.860826835 0.91508288 1.0041143 1717.288
## a[2]  1.04821740 0.010516688 1.031526150 1.06519275 0.9994354 2032.849
## b[1]  0.14938826 0.072924667 0.033531250 0.26669909 1.0017930 1138.217
## b[2]  0.01902554 0.017589925 0.001376953 0.05220501 1.0009499 1304.444
## sigma 0.11429173 0.006214132 0.104709945 0.12461595 1.0043364 1774.671
```

The estimates for a[cid] remained the same, which are the differences in average GDP for African vs non-African countries. The estimates for b[cid] differ between models, with the initial models indicating that cid[1] (African countries) have a positive relationship between ruggedness and GDP (the author hypothesized this was due to less access by colonialists), while non-African countries have a negative relationship between ruggedness and GDP (likely due to more ruggedness = more secluded = less trade).

The weakly informative Gaussian prior of dnorm(0, 0.3) allowed the GDP ~ ruggedness slope to have its direction (sign) be influenced by the data, and differ by level of [cid].

In contrast, using the exponential dexp(0.3) prior, which looks like this:

```
curve(dexp(x, rate = 0.3), from=-1, to=10)
```



Which gives values below 0 a prior plausibility of 0. Thus the model restricts the slope of ruggedness to positive values, and estimates a slope of (very near) 0.

9H3. Sometimes changing a prior for one parameter has unanticipated effects on other parameters. This is because when a parameter is highly correlated with another parameter in the posterior, the prior influences both parameters. Here's an example to work and think through. Go back to the leg length example in Chapter 6 and use the code there to simulate height and leg lengths for 100 imagined individuals. Below is the model you fit

before, resulting in a highly correlated posterior for the two beta parameters. This time, fit the model using ulam:

```
rm(list=ls())
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)
```

```
m5.8s <- 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 ~ dexp( 1 )
  ) , data=d, chains=4,
  start=list(a=10,bl=0,br=0.1,sigma=1) )
```

```
## Running MCMC with 4 sequential chains...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 6.2 seconds.
## Chain 2 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 2 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 2 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 2 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 finished in 5.0 seconds.
## Chain 3 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 3 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 3 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 3 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 3 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 finished in 4.2 seconds.
## Chain 4 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 4 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 4 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 4 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 4 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 4 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 4 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 4 finished in 6.0 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 5.4 seconds.
## Total execution time: 21.7 seconds.
```

```
m5.8s2 <- 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 ~ dexp( 1 )
  ) , data=d, chains=4,
  constraints=list(br="lower=0"),
  start=list(a=10,bl=0,br=0.1,sigma=1) )
```

```
## In file included from stan/lib/stan_math/stan/math/prim/prob/von_mises_lccdf.hpp:5,
##                  from stan/lib/stan_math/stan/math/prim/prob/von_mises_ccdf_log.hpp:4,
##                  from stan/lib/stan_math/stan/math/prim/prob.hpp:359,
##                  from stan/lib/stan_math/stan/math/prim.hpp:16,
##                  from stan/lib/stan_math/stan/math/rev.hpp:16,
##                  from stan/lib/stan_math/stan/math.hpp:19,
##                  from stan/src/stan/model/model_header.hpp:4,
##                  from C:/Users/chris/AppData/Local/Temp/RtmpgVu6TX/model-3ed86d003030.hpp:2:
## stan/lib/stan_math/stan/math/prim/prob/von_mises_cdf.hpp: In function 'stan::return_type_t<T_
x, T_sigma, T_l> stan::math::von_mises_cdf(const T_x&, const T_mu&, const T_k&)':
## stan/lib/stan_math/stan/math/prim/prob/von_mises_cdf.hpp:194: note: '-Wmisleading-indentatio
n' is disabled from this point onwards, since column-tracking was disabled due to the size of th
e code/headers
##   194 |       if (cdf_n < 0.0)
##       |
```

```
## stan/lib/stan_math/stan/math/prim/prob/von_mises_cdf.hpp:194: note: adding '-flarge-source-fi
les' will allow for more column-tracking support, at the expense of compilation time and memory
```

```
## Running MCMC with 4 sequential chains...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 4.9 seconds.
## Chain 2 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 2 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 2 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 2 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 finished in 3.8 seconds.
## Chain 3 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 3 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 3 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 3 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 3 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 finished in 5.0 seconds.
## Chain 4 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 4 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 4 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 4 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 4 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 4 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 4 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 4 finished in 5.4 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 4.8 seconds.
## Total execution time: 19.7 seconds.
```

```
## Warning: 24 of 2000 (1.0%) transitions ended with a divergence.
## See https://mc-stan.org/misc/warnings for details.
```

```
## Warning: 12 of 2000 (1.0%) transitions hit the maximum treedepth limit of 11.
## See https://mc-stan.org/misc/warnings for details.
```

These models predict height as a linear function of left and right leg lengths.

```
precis(m5.8s)
```

```
##            mean         sd       5.5%     94.5%     n_eff    Rhat4
## a     0.9940034 0.27743260  0.5552781 1.4545635  809.0290 1.000984
## bl    0.3206251 2.72276737 -3.9133610 4.8624456  611.2357 1.013634
## br    1.6723446 2.72558971 -2.8594454 5.9284403  612.8345 1.013542
## sigma 0.6338726 0.04590926  0.5637914 0.7109269 1003.5834 1.000980
```

```
precis(m5.8s2)
```

```
##             mean         sd       5.5%    94.5%     n_eff     Rhat4
## a      0.9757032 0.28906028  0.5224986 1.438248 1243.2885 0.9999253
## bl    -0.8463886 1.78887328 -3.8538860 1.588044  500.5347 1.0017724
## br     2.8431110 1.78940335  0.3938628 5.867722  499.3083 1.0017895
## sigma  0.6350030 0.04656494  0.5661548 0.714131 1034.0722 0.9997312
```

The first model summary is what we saw earlier in chapter 6; where the predictors (left and right legs) are highly correlated (thus redundant) and so many combinations of slopes bl and br can approximate the actual slope of height ~ leg length (which is around height = 1 + 2*leg_length), and thus produce the same predictions. This also explains the large sd's.

The second model pushes br to only positive values with the mean of the probability distribution at br=2.84, thus for the combinations of bl and br to approximate the actual slope for leg length (2), the mean slope of bl should be around -0.84.

9H4. For the two models fit in the previous problem, use WAIC or PSIS to compare the effective numbers of parameters for each

model. You will need to use log_lik=TRUE to instruct ulam to compute the terms that both WAIC and PSIS need. Which model has more effective parameters? Why?

```
# compare(m5.8s, m5.8s2, func=WAIC, log_lik="log_like")
```

This returns the error: “Error in attr(object,”cstanfit”)$draws : $ operator not defined for this S4 class” which I’ve attempted to troubleshoot for about an hour now. I’ll just assume that the WAIC / PSIS scores are the same since both models make the same predictions for the total influence of leg length on height.