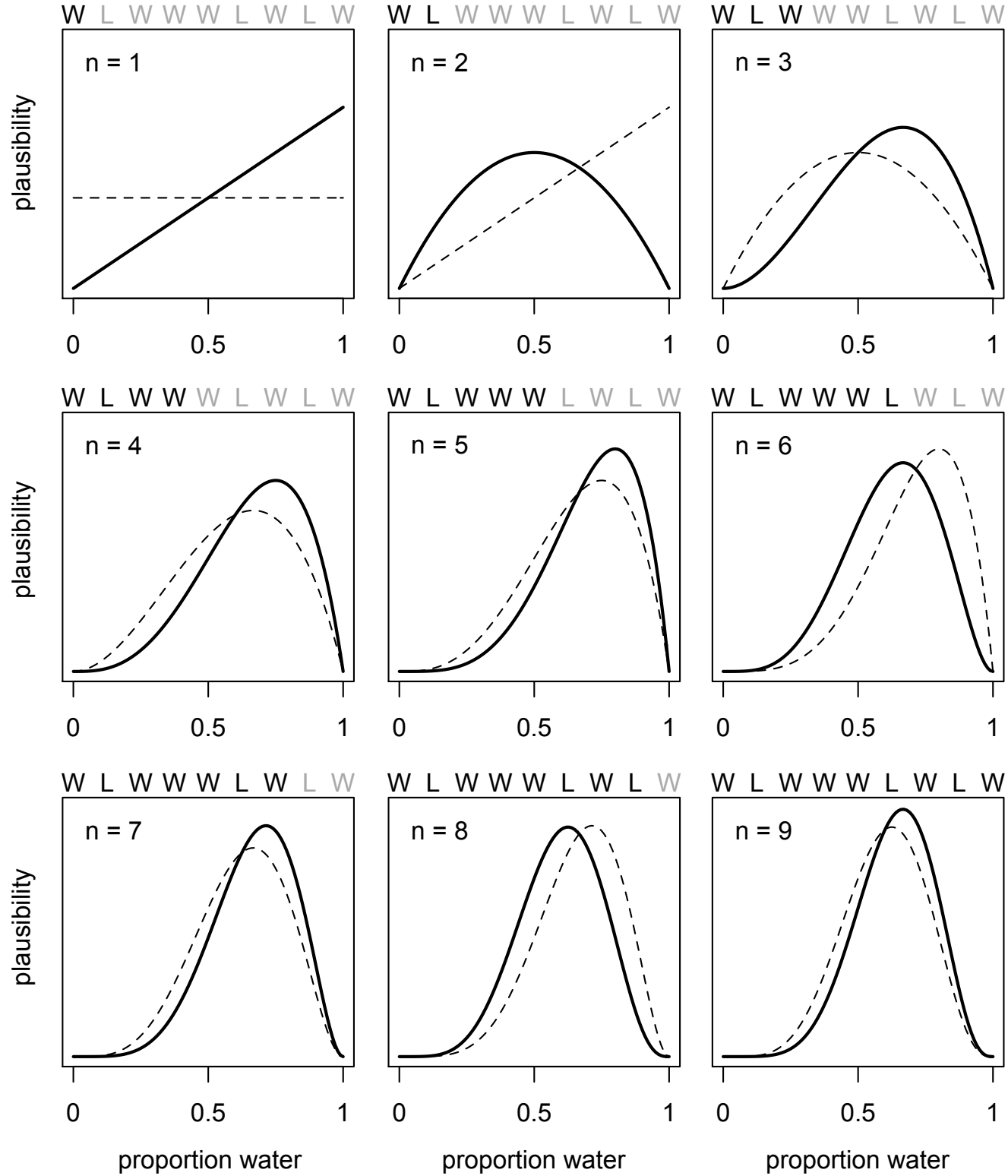
Statistical Rethinking
Winter 2019

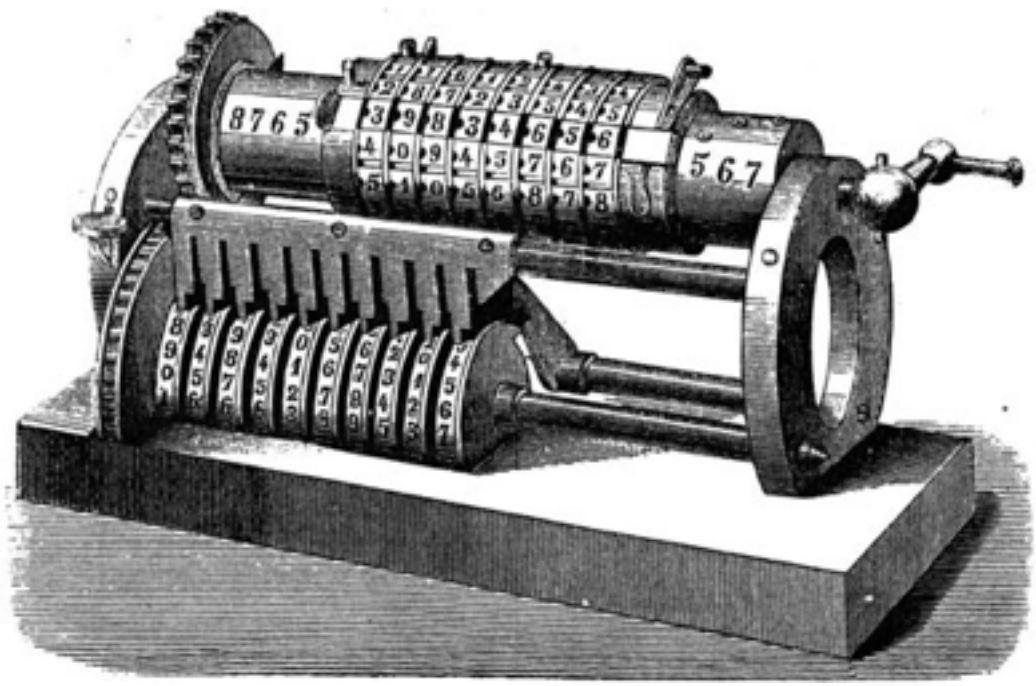
Lecture 10 / Week 5

Markov Chain
Monte Carlo


W L W W W L W L W
n = 1
n = 2
n = 3
n = 4
n = 5
n = 6
n = 7
n = 8
n = 9
plausibility
0
0.5
1
proportion water

Computing the posterior

1. Analytical approach (often impossible)
2. Grid approximation (very intensive)
3. Quadratic approximation (limited)
4. Markov chain Monte Carlo (intensive)

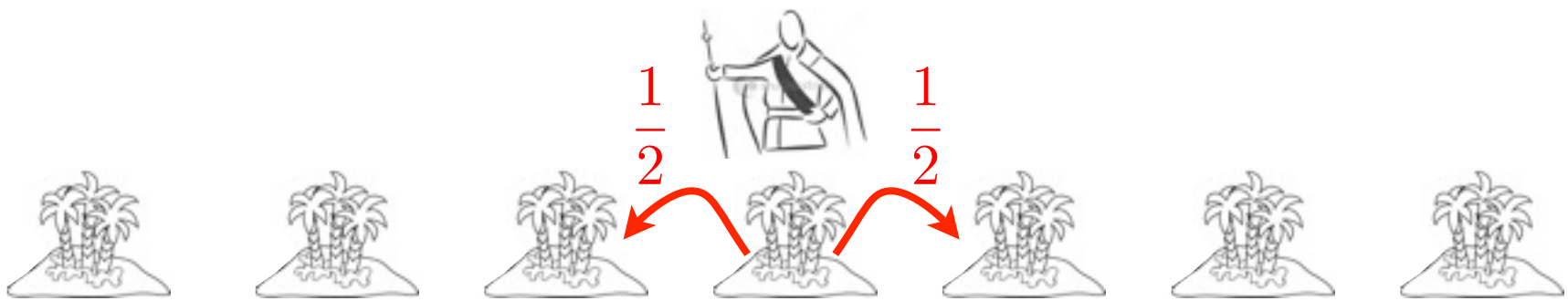


King Markov

The Metropolis Archipelago

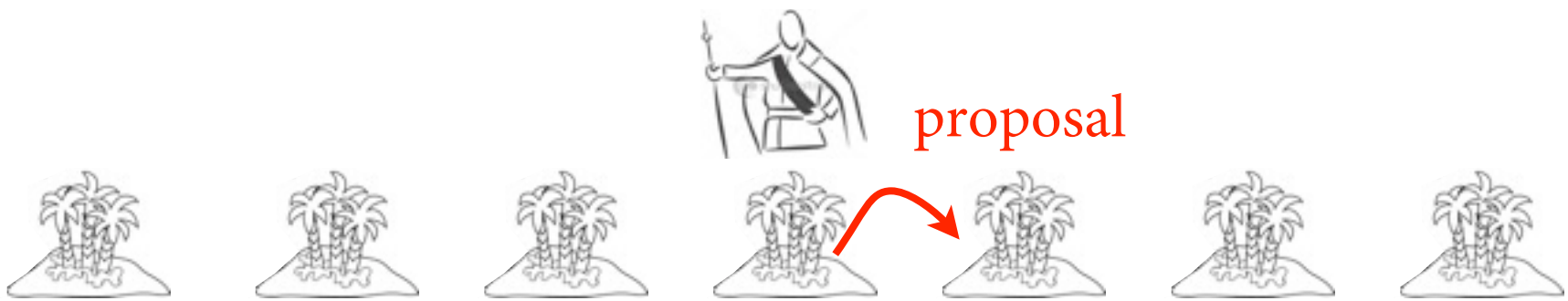
Contract: King Markov must visit each island in proportion to its population size.

Here's how he does it...



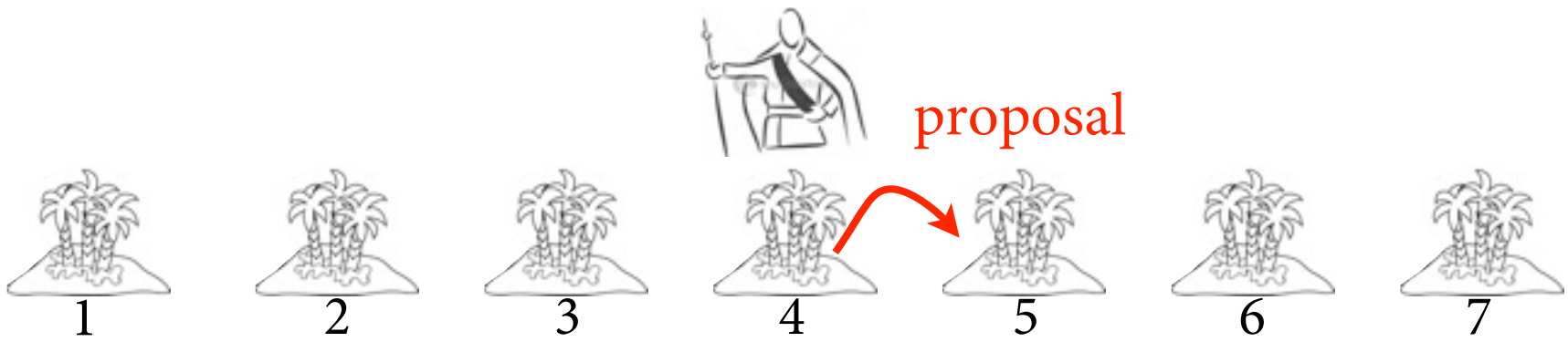
(1) Flip a coin to choose island on left or right.
Call it the “proposal” island.

(1) Flip a coin to choose island on left or right.
Call it the "proposal" island.



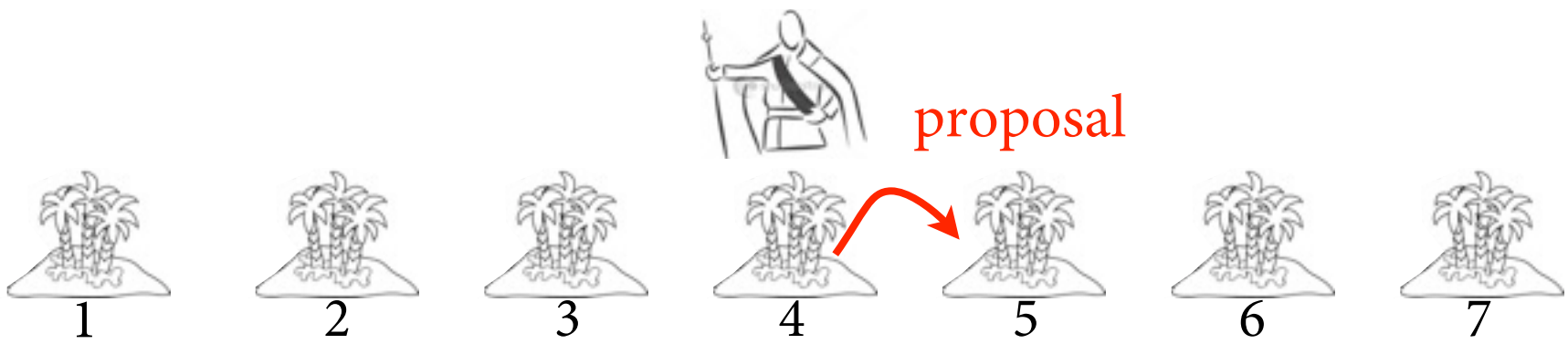
(2) Find population of proposal island.

(1) Flip a coin to choose island on left or right.
Call it the "proposal" island.



(2) Find population of proposal island.

(1) Flip a coin to choose island on left or right.
Call it the "proposal" island.

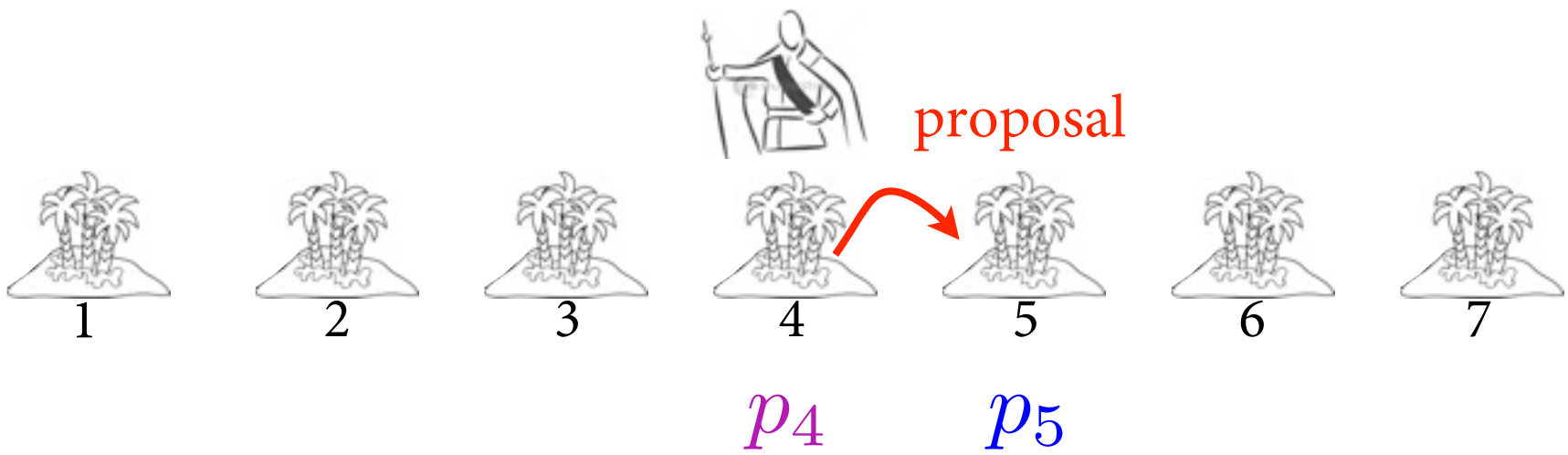


p_5

(2) Find population of proposal island.

(1) Flip a coin to choose island on left or right.
Call it the "proposal" island.

(2) Find population of proposal island.

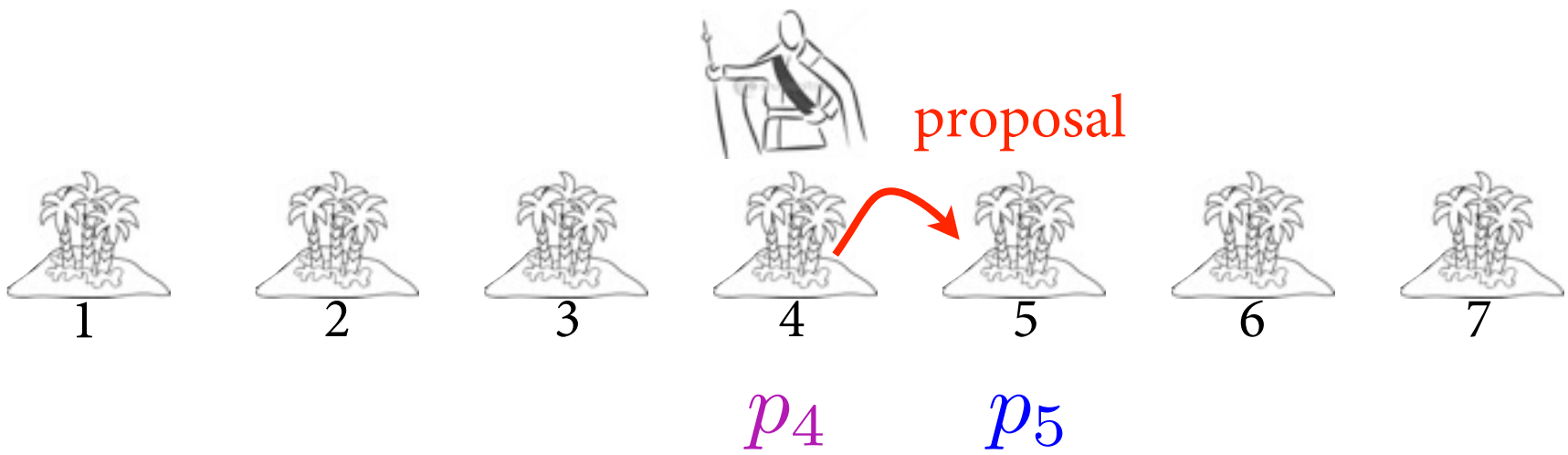


(3) Find population of current island.

(1) Flip a coin to choose island on left or right.
Call it the "proposal" island.

(2) Find population of proposal island.

(3) Find population of current island.



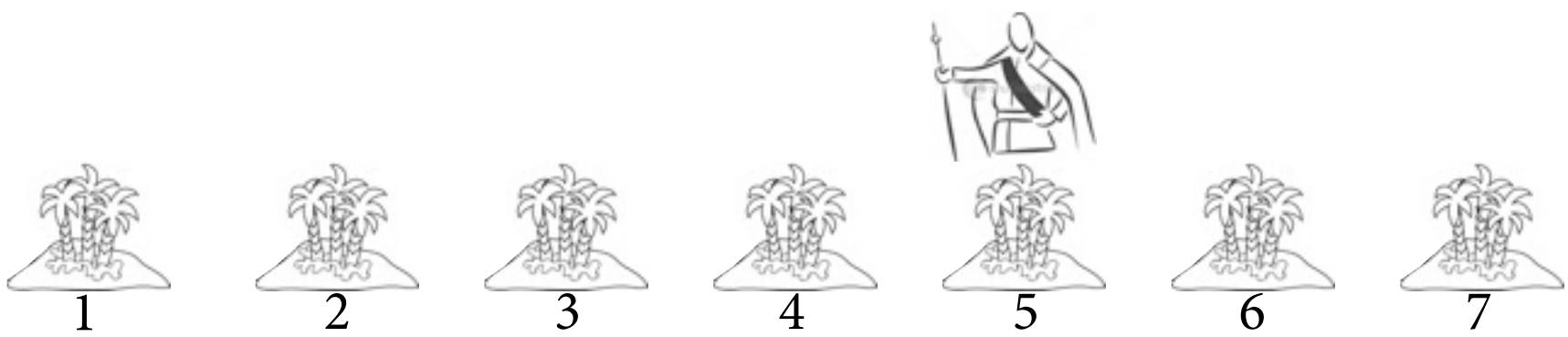
(4) Move to proposal, with probability = $\frac{p_5}{p_4}$

(1) Flip a coin to choose island on left or right.
Call it the "proposal" island.

(2) Find population of proposal island.

(3) Find population of current island.

(4) Move to proposal, with probability = $\frac{p_5}{p_4}$



(5) Repeat from (1)

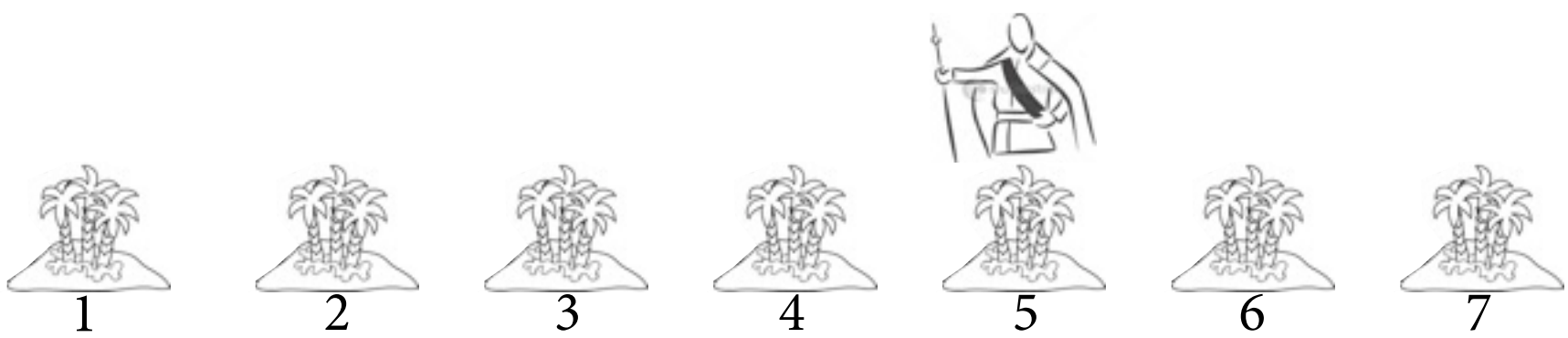
(1) Flip a coin to choose island on left or right.
Call it the "proposal" island.

(2) Find population of proposal island.

(3) Find population of current island.

(4) Move to proposal, with probability = $\frac{p_5}{p_4}$

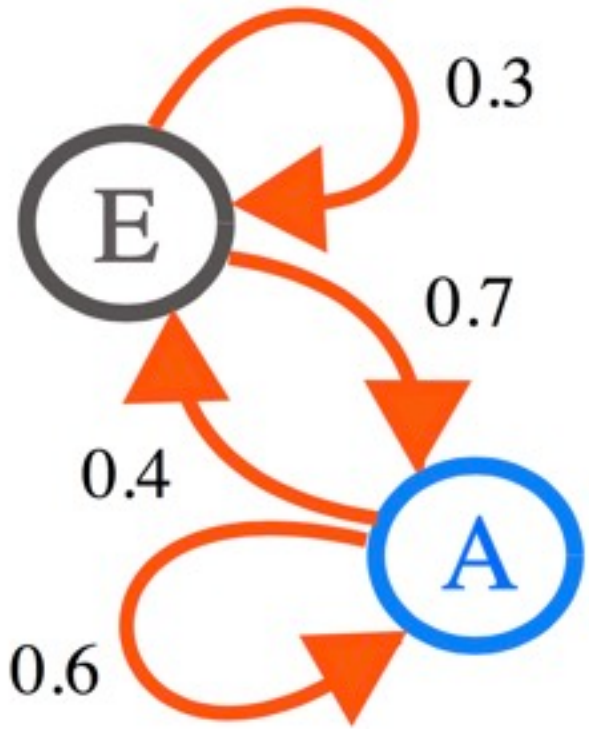
(5) Repeat from (1)



This procedure ensures visiting each island in proportion to its population, *in the long run.*

Markov chain Monte Carlo

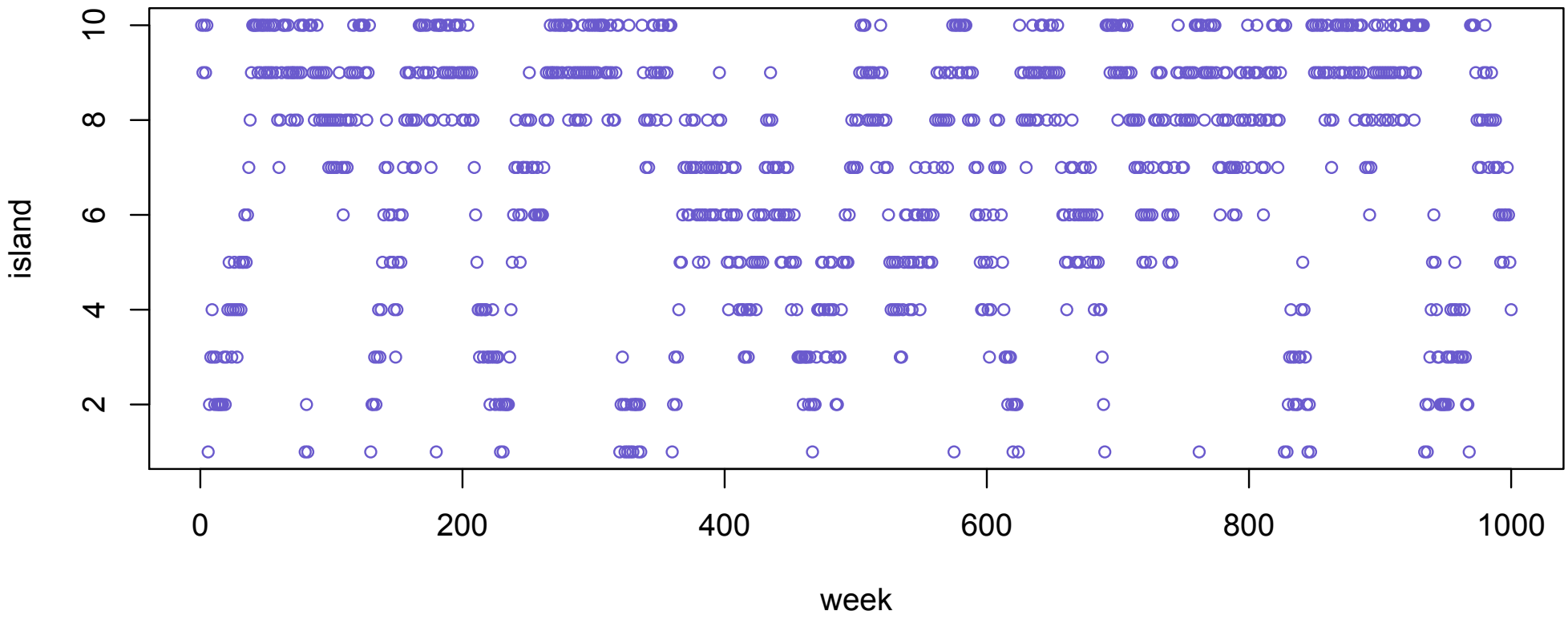
- Markov chain Monte Carlo (MCMC)
  - Understand the approach
  - Meet different algorithms
- Interfaces to MCMC: Stan & `ulam`
- How to sample responsibly
- How to recognize and fix problems



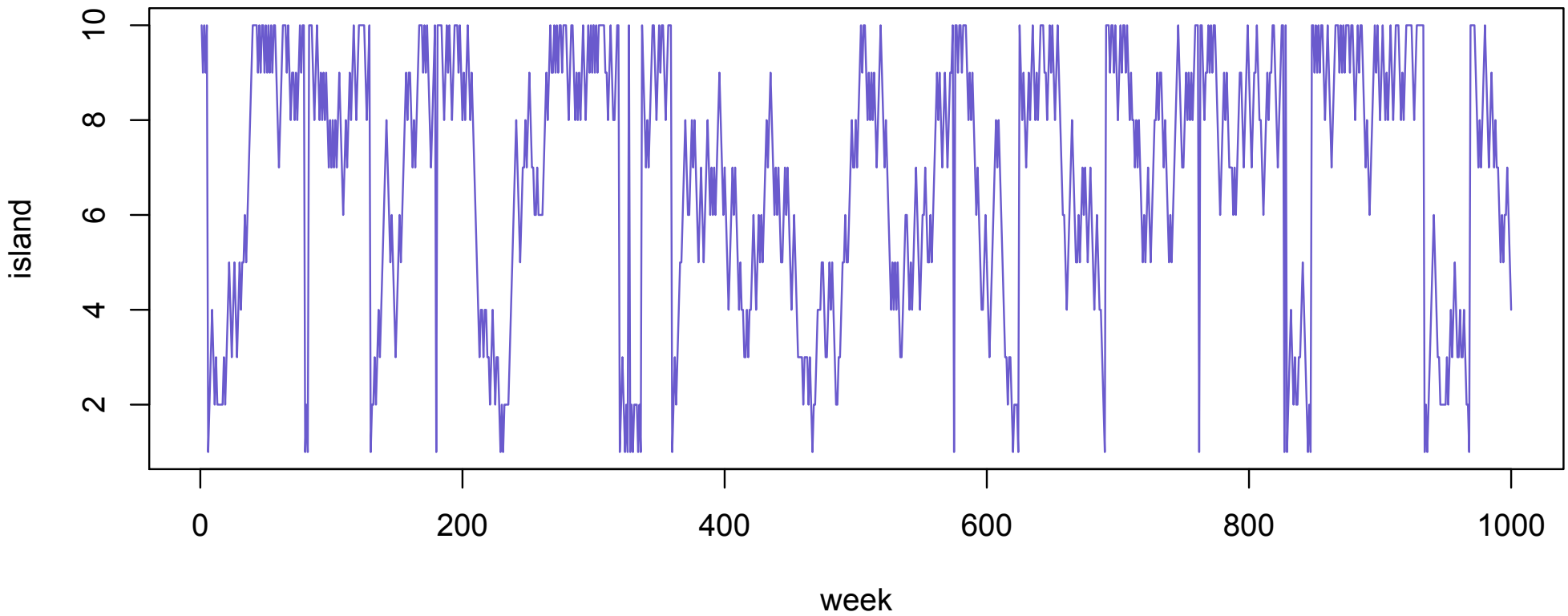
Metropolis algorithm

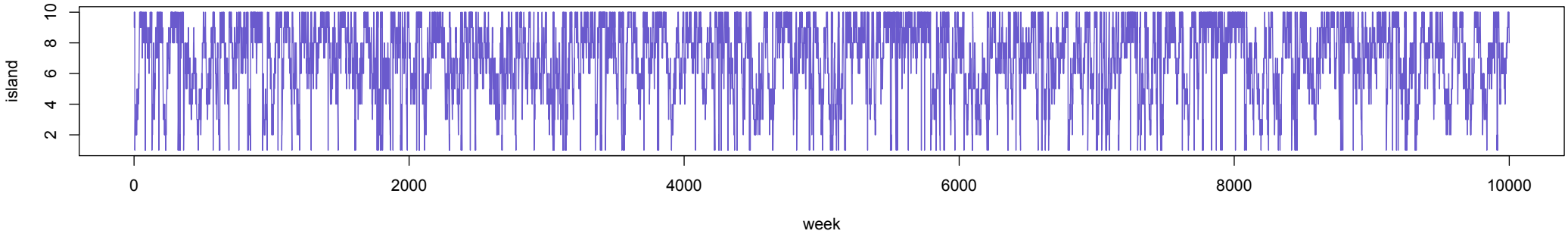
```
num_weeks <- 1e5
positions <- rep(0,num_weeks)
current <- 10
for ( i in 1:num_weeks ) {
    # record current position
    positions[i] <- current

    # 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 <- proposal/current
    current <- ifelse( runif(1) < prob_move , proposal , current )
}
```

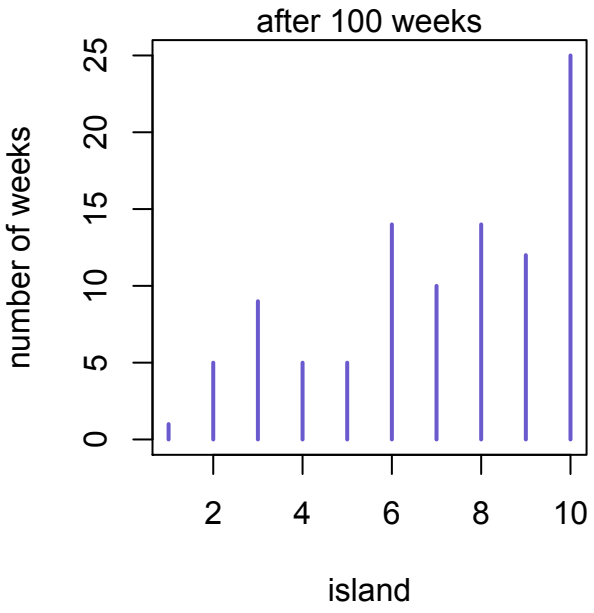
Markov's chain of visits

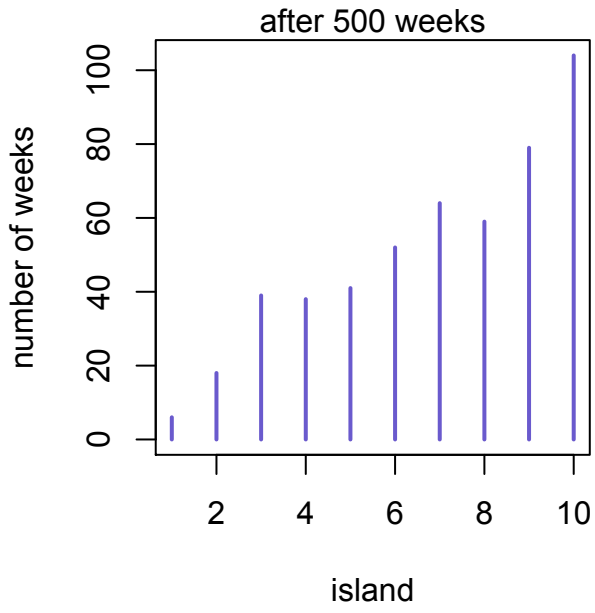


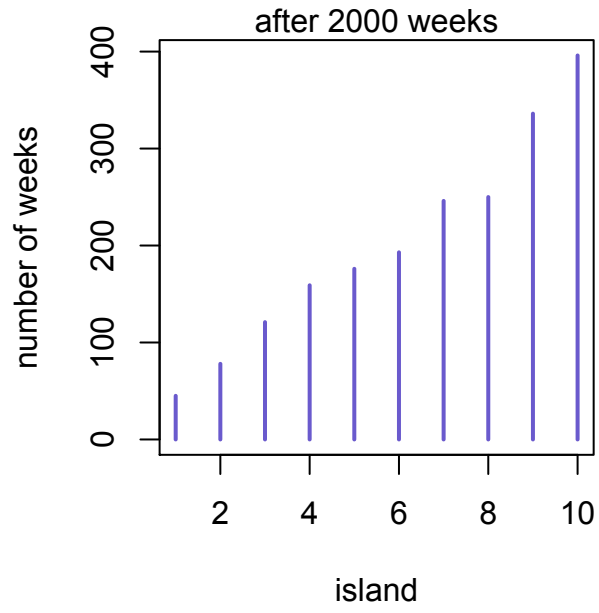
Markov's chain of visits




island
2
4
6
8
10
0
2000
4000
6000
8000
10000
week

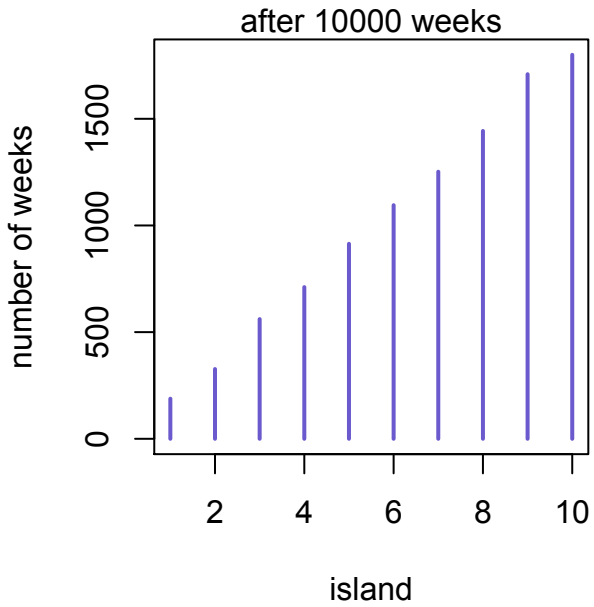

after 100 weeks
number of weeks
0
5
10
15
20
25
2
4
6
8
10
island


after 500 weeks
number of weeks
0
20
40
60
80
100
2
4
6
8
10
island


after 2000 weeks
number of weeks
0
100
200
300
400
2
4
6
8
10
island

Markov's chain of visits

- Converges to correct proportions, in the long run
- No matter which island starts
- As long as proposals are *symmetric*
- Example of *Metropolis algorithm*



Metropolis and MCMC

- Usual use is to draw samples from a posterior distribution
  - "Islands": parameter values
  - "Population size": proportional to posterior probability
- Works for any number of dimensions (parameters)
- Works for continuous as well as discrete parameters

Metropolis and MCMC

- *Metropolis*: Simple version of *Markov chain Monte Carlo* (MCMC)
- Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953)

THE JOURNAL OF CHEMICAL PHYSICS VOLUME 21, NUMBER 6 JUNE, 1953

Equation of State Calculations by Fast Computing Machines

NICHOLAS METROPOLIS, ARIANNA W. ROSENBLUTH, MARSHALL N. ROSENBLUTH, AND AUGUSTA H. TELLER,
Los Alamos Scientific Laboratory, Los Alamos, New Mexico

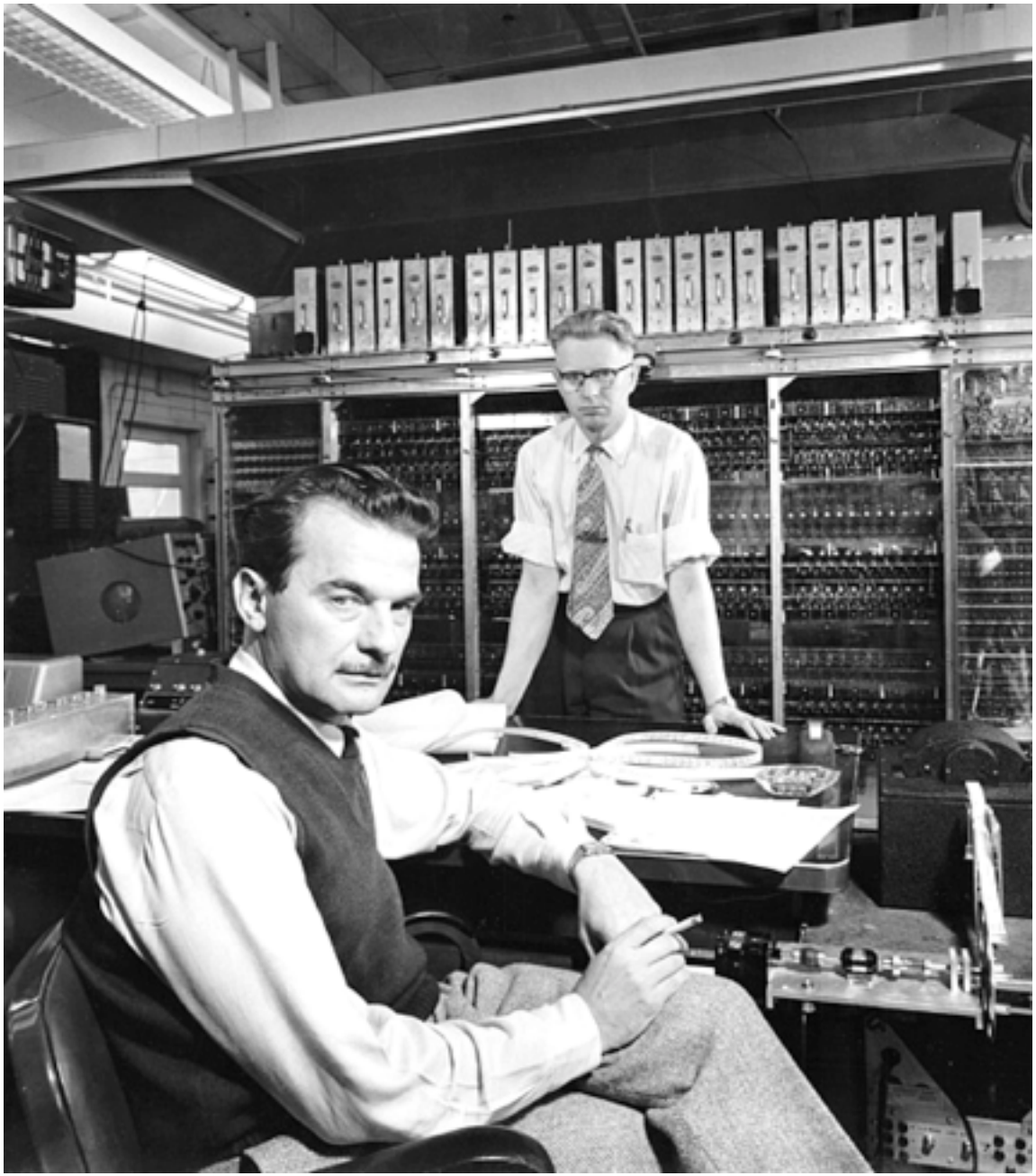
AND

EDWARD TELLER,* *Department of Physics, University of Chicago, Chicago, Illinois*

(Received March 6, 1953)

A general method, suitable for fast computing machines, for investigating such properties as equations of state for substances consisting of interacting individual molecules is described. The method consists of a modified Monte Carlo integration over configuration space. Results for the two-dimensional rigid-sphere system have been obtained on the Los Alamos MANIAC and are presented here. These results are compared to the free volume equation of state and to a four-term virial coefficient expansion.

MANIAC:
Mathematical Analyzer, Numerical Integrator, and Computer



MANIAC:
1000 pounds
5 kilobytes of memory
70k multiplications/sec

Your laptop:
4–7 pounds
2–8 million kilobytes
Billions of multiplications/sec

Metropolis and MCMC

- *Metropolis*: Simple version of *Markov chain Monte Carlo* (MCMC)
- *Chain*: Sequence of draws from distribution
- *Markov chain*: History doesn't matter, just where you are now
- *Monte Carlo*: Random simulation

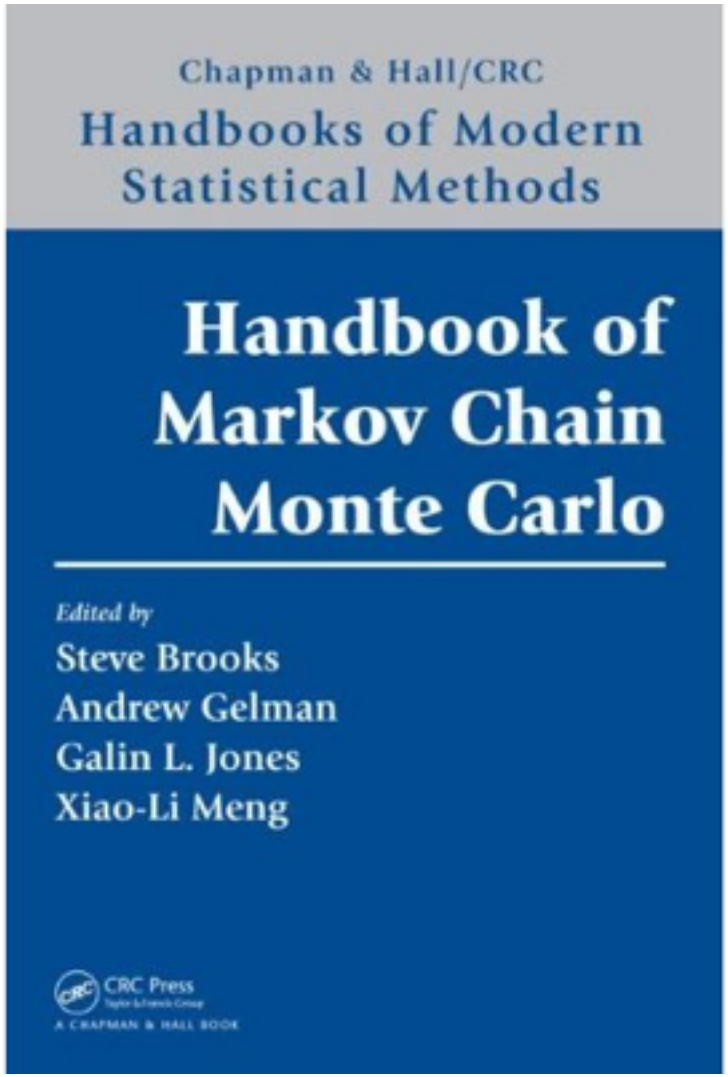
Andrei Andreyevich Markov
(Ма́рков)
(1856–1922)

Why MCMC?

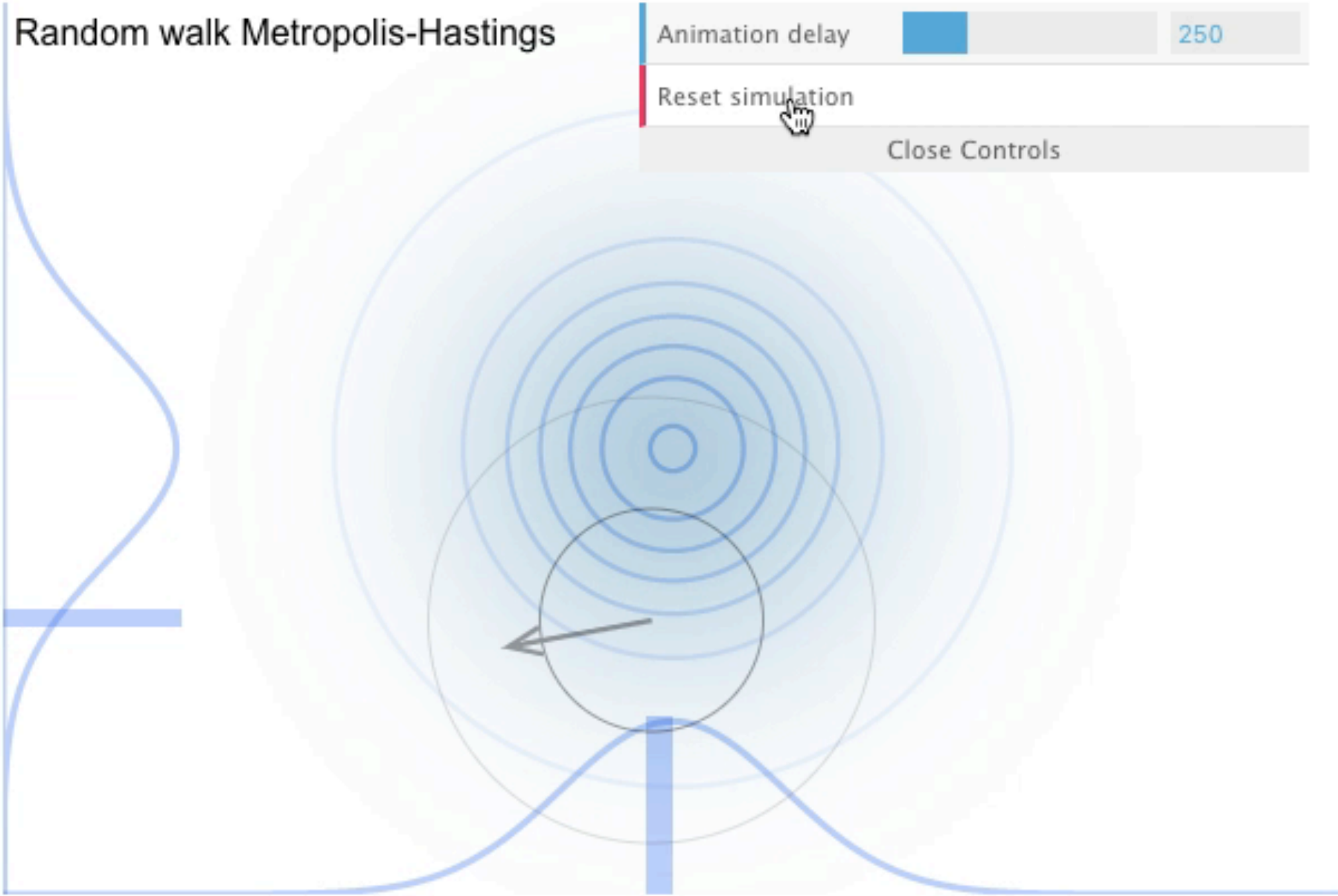
- Sometimes can't write an integrated posterior
- Even when can, often cannot use it
- Many problems are like this: Multilevel models, networks, phylogenies, spatial models
- Optimization not a good strategy in high dimensions — must have full distribution
- **MCMC is not fancy. It is old and essential.**

MCMC strategies

- Metropolis: Granddaddy of them all
- Metropolis-Hastings (MH): More general
- Gibbs sampling (GS): Efficient version of MH
- Metropolis and Gibbs are "guess and check" strategies
- Hamiltonian Monte Carlo (HMC) fundamentally different
- New methods being developed, but future belongs to the gradient

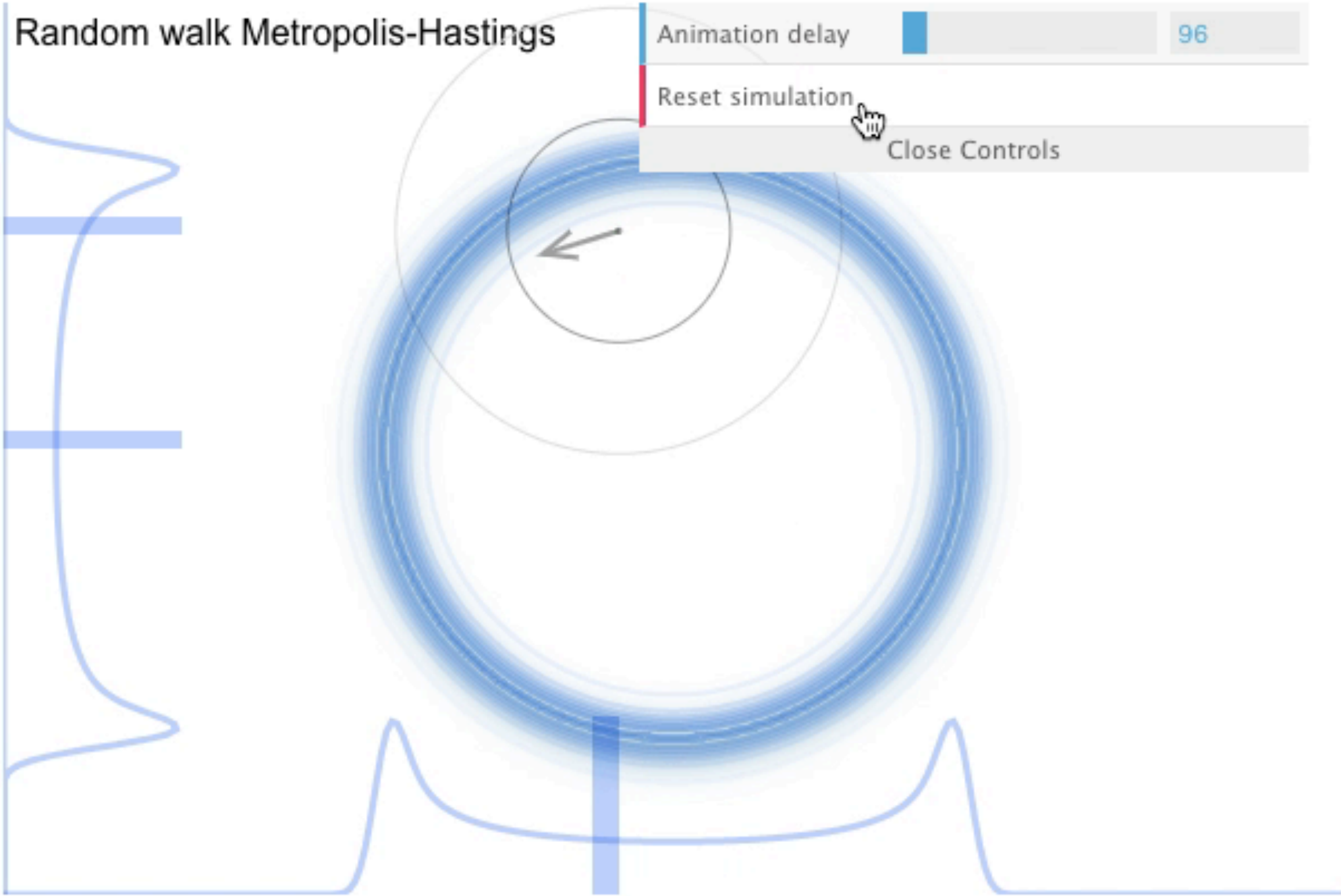


https://chi-feng.github.io/mcmc-demo/



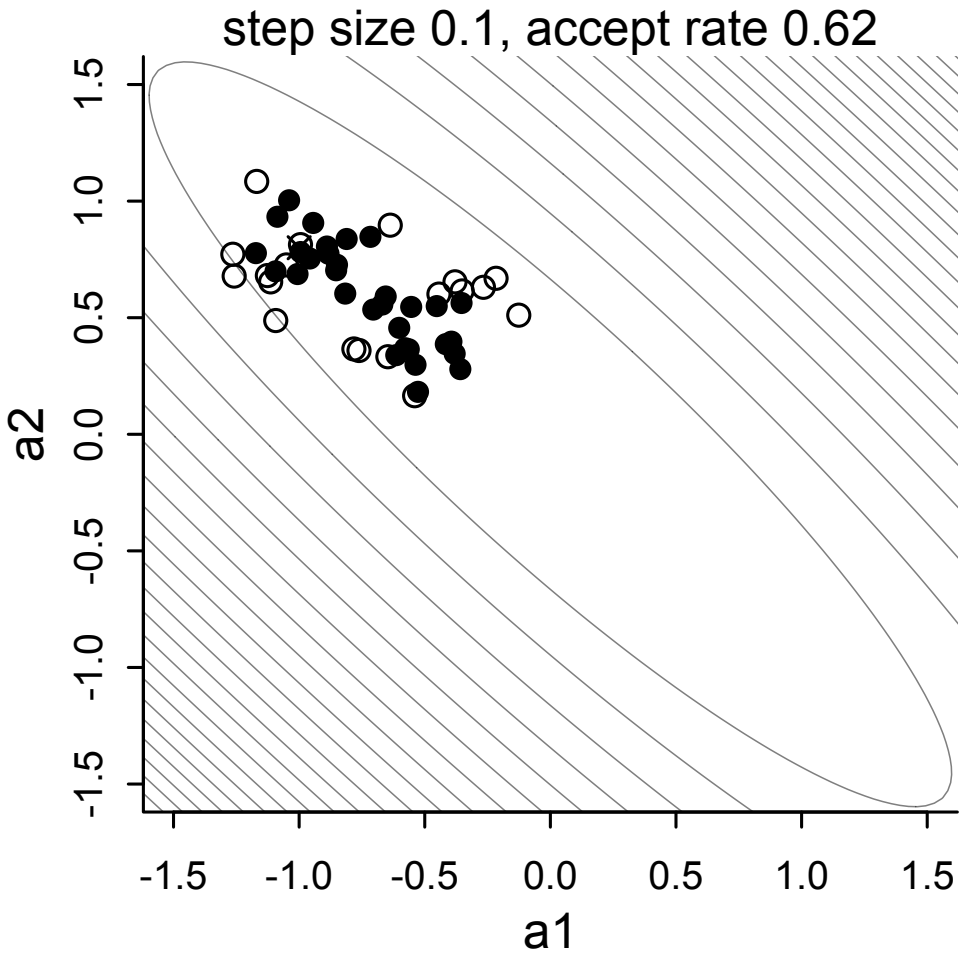
https://chi-feng.github.io/mcmc-demo/

https://chi-feng.github.io/mcmc-demo/

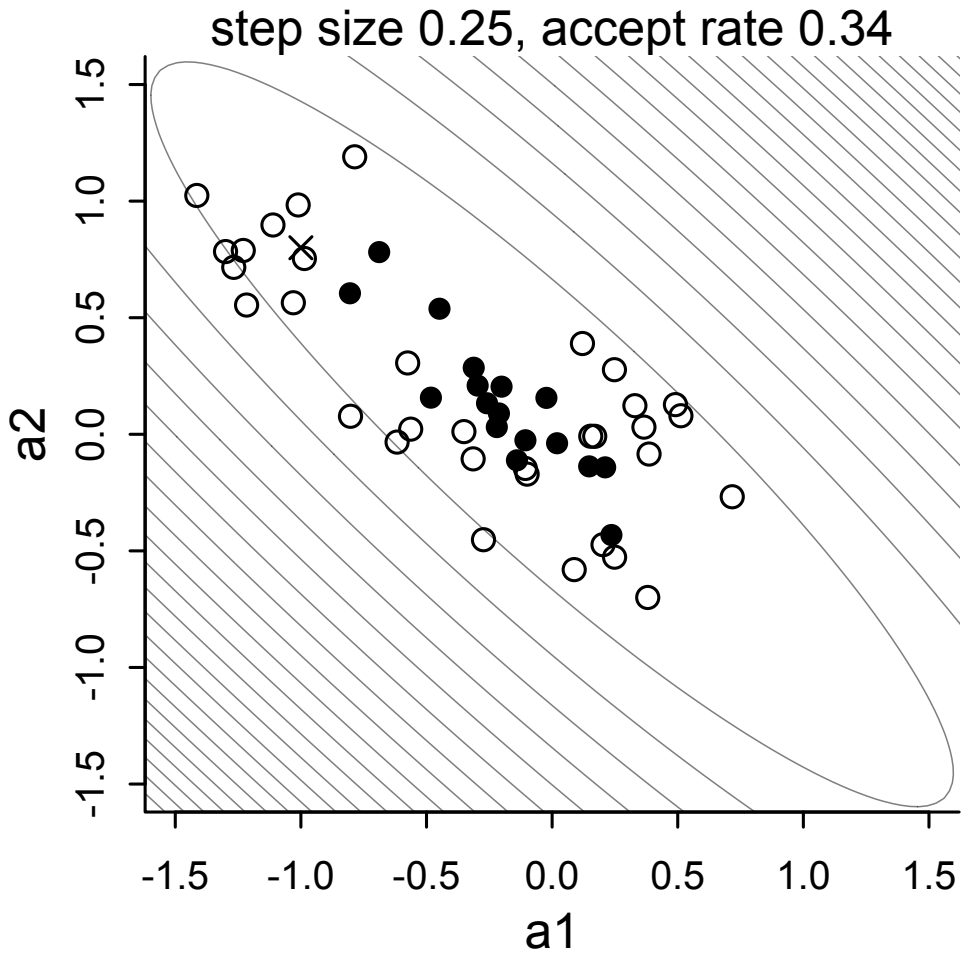


https://chi-feng.github.io/mcmc-demo/

Metropolis gets stuck



small steps, slow walk



bigs steps, low accept rate

Figure 9.3

Hamiltonian Monte Carlo

- Problem with Gibbs sampling (GS)
  - High dimension spaces are *concentrated*
  - GS gets stuck, degenerates towards random walk
  - Inefficient because re-explores
- Hamiltonian dynamics to the rescue
  - represent parameter state as particle
  - flick it around frictionless log-posterior
  - record positions
  - no more "guess and check"
  - all proposals are good proposals

William Rowan Hamilton (1805–1865) Commemorated on Irish Euro coin

Hamiltonian parable

- King Monty's kingdom is a narrow valley N–S
- Population distribution inversely proportional to altitude

Hamiltonian parable

- King Monty's kingdom is a narrow valley N–S
- Population distribution inversely proportional to altitude
- Algorithm:
  - Start driving randomly N or S at random speed
  - Car speeds up as it goes downhill
  - Car slows as it goes uphill, might turn around
  - Drive for pre-specified duration, then stop
  - Repeat
- Stopping positions will be proportional to population

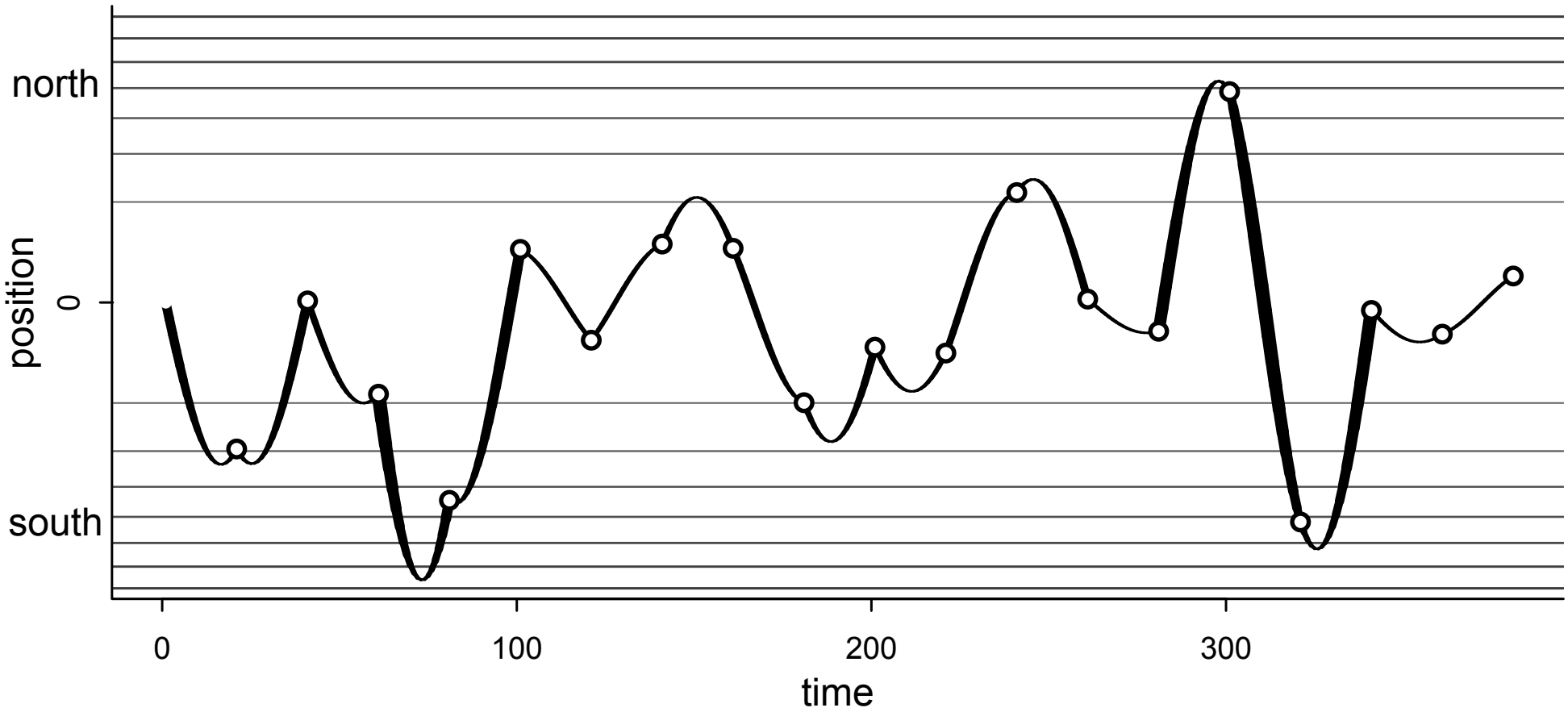
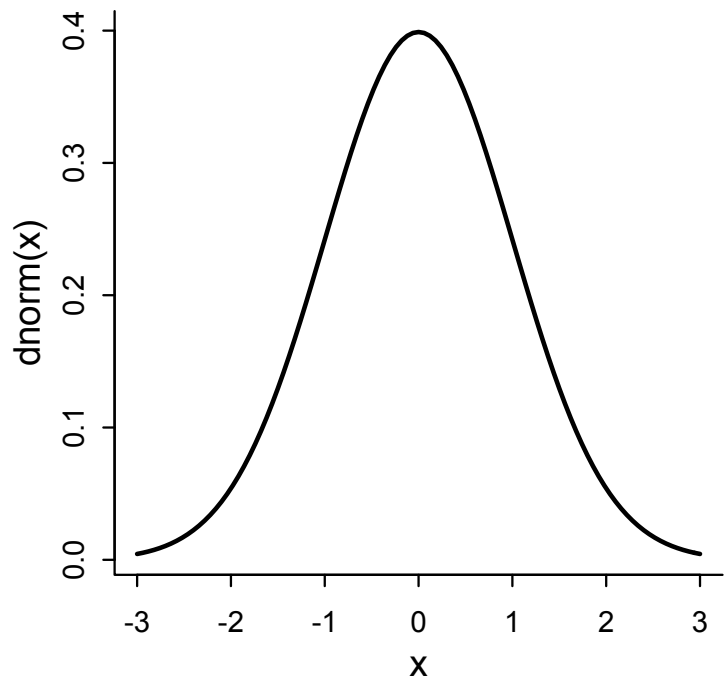
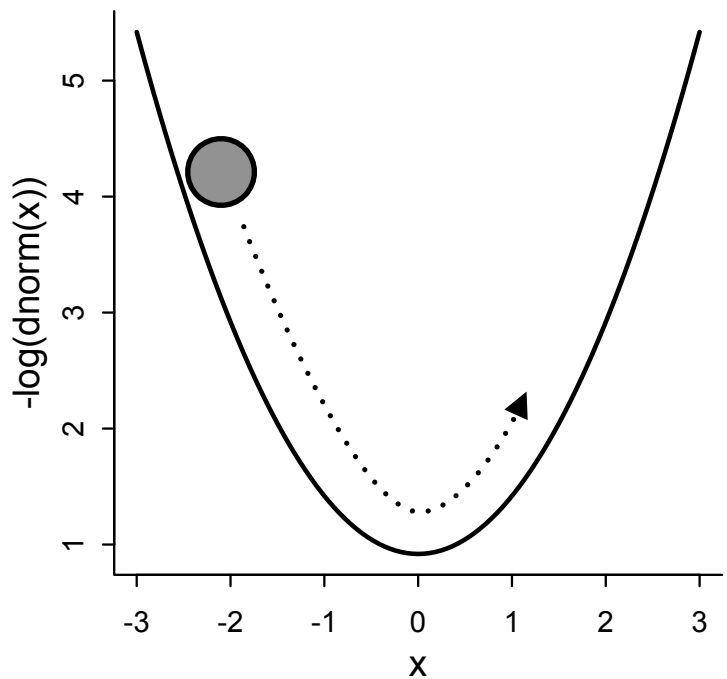


Figure 9.5

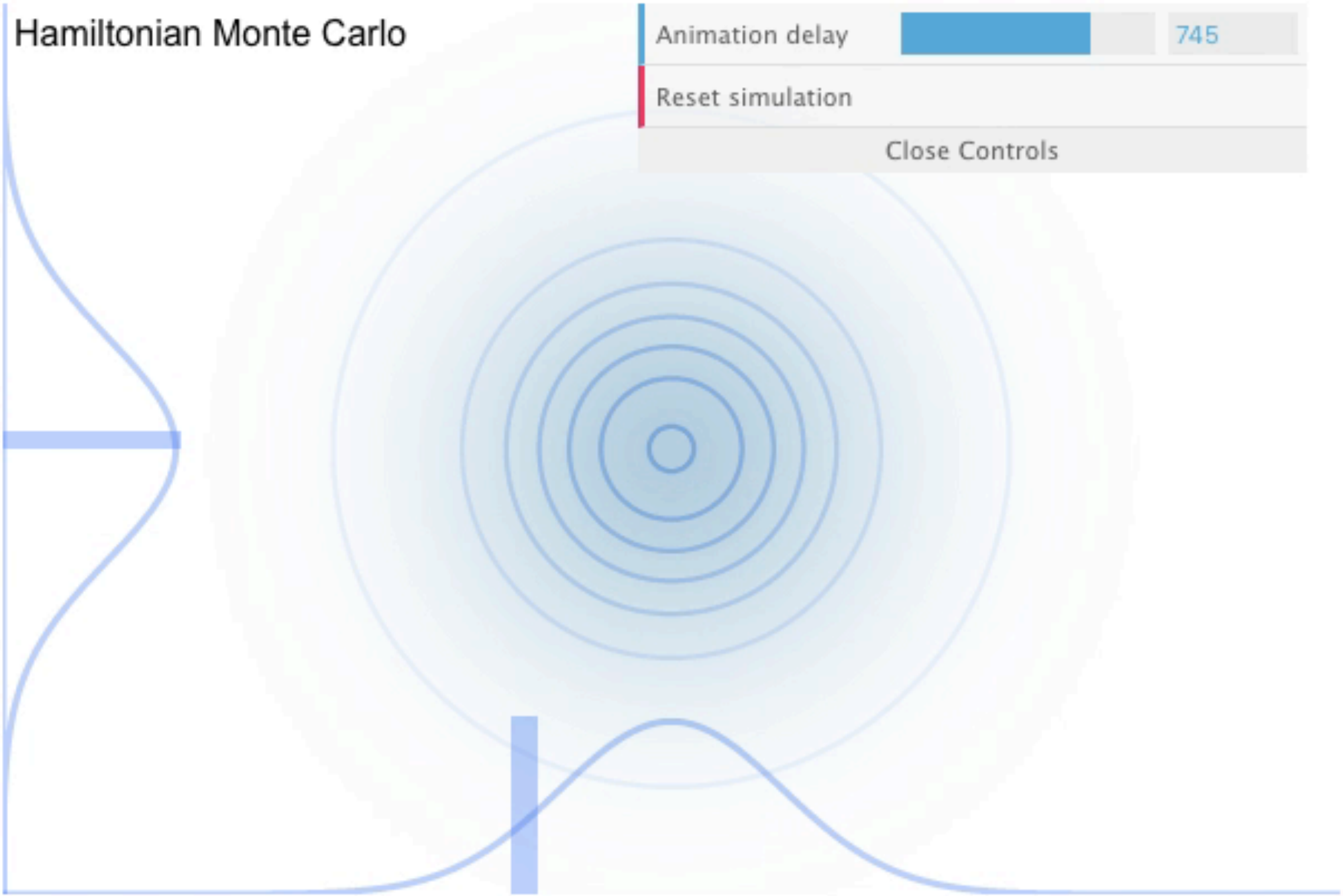
Hamiltonian Monte Carlo

- Location is parameter values
- Really simulate motion on frictionless surface
- Surface is minus-log-posterior
- Series of simulations, each starting from previous
- Stopping points comprise valid MCMC samples

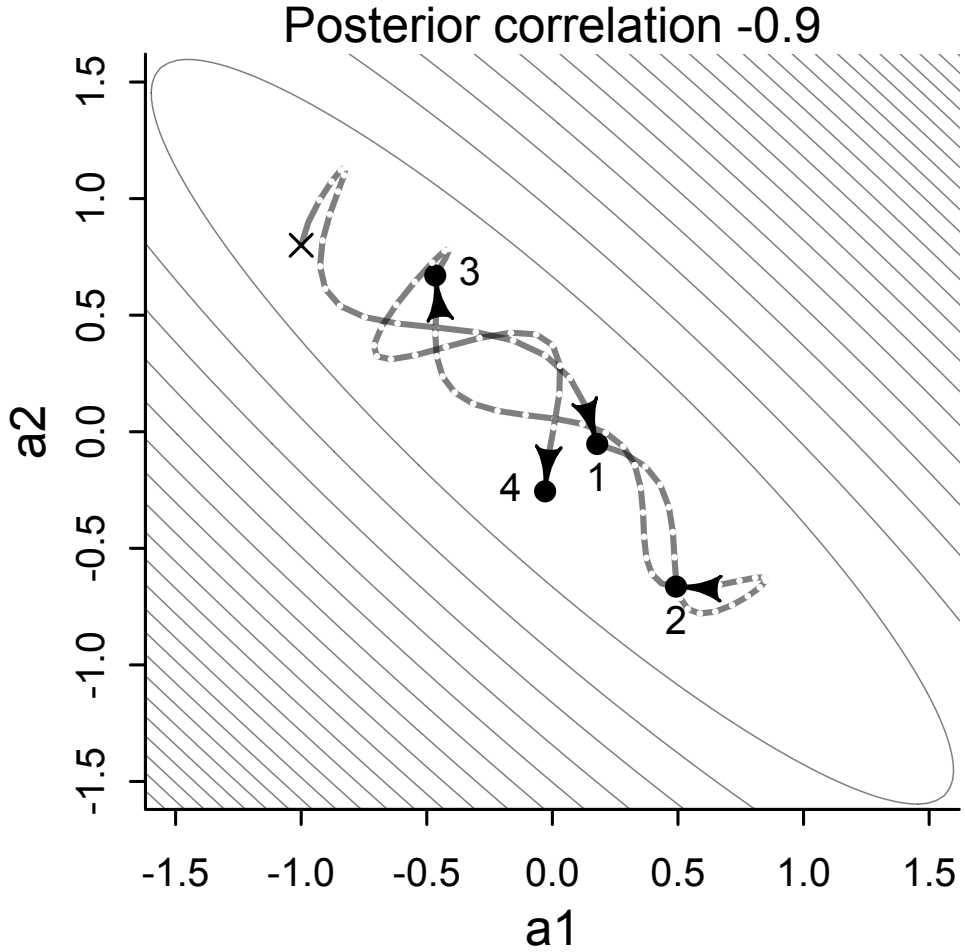




https://chi-feng.github.io/mcmc-demo/



https://chi-feng.github.io/mcmc-demo/


Posterior correlation -0.9
a2
a1
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
1
2
3
4

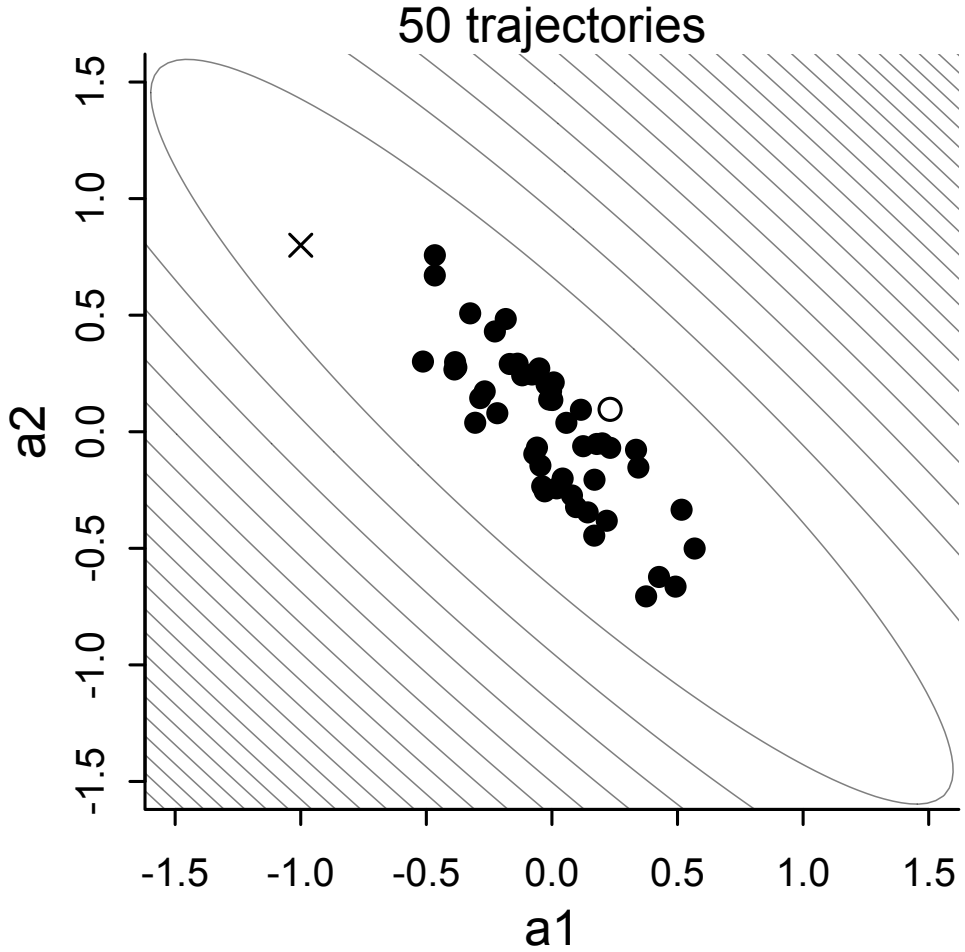
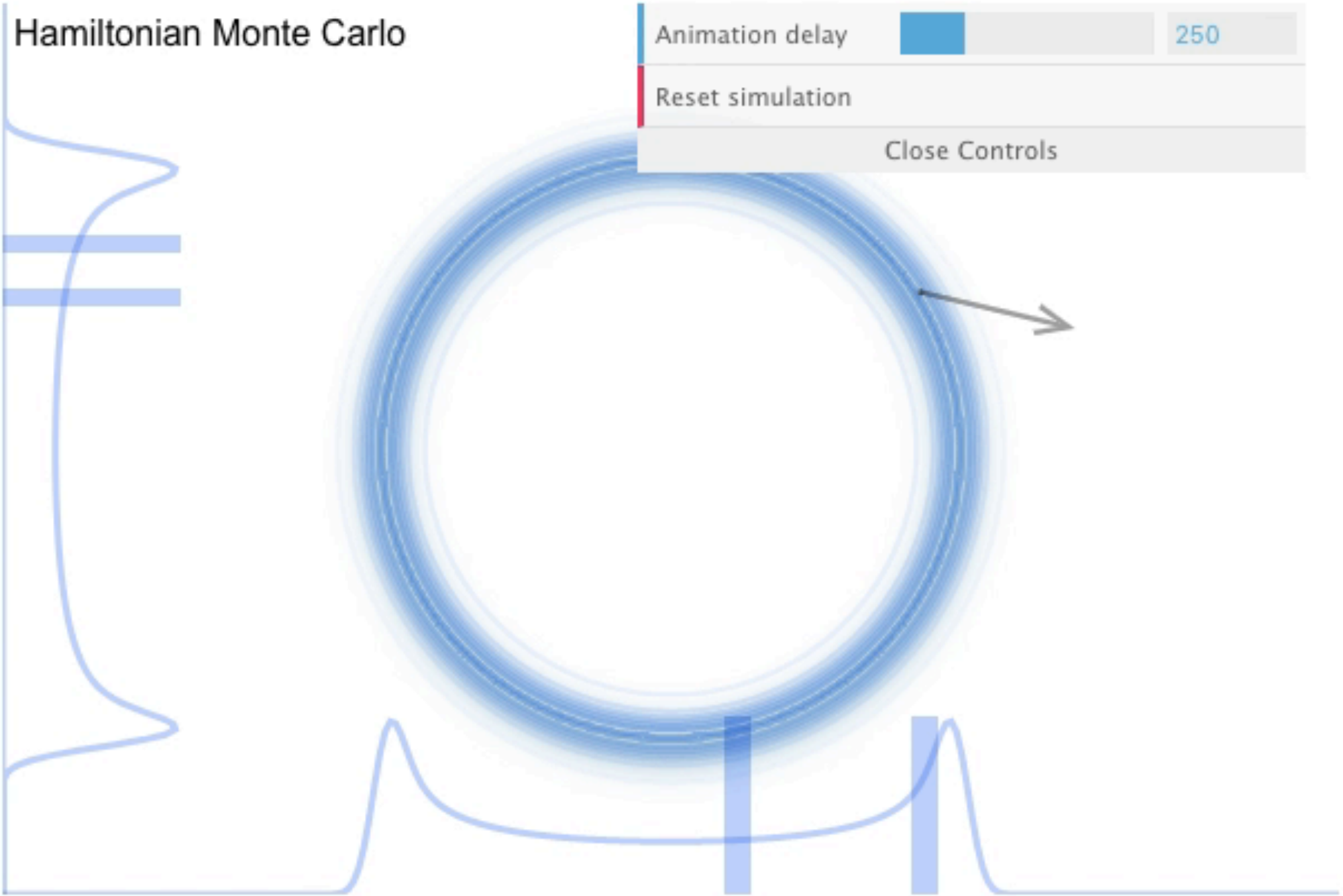

50 trajectories
a2
a1
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5

Figure 9.6

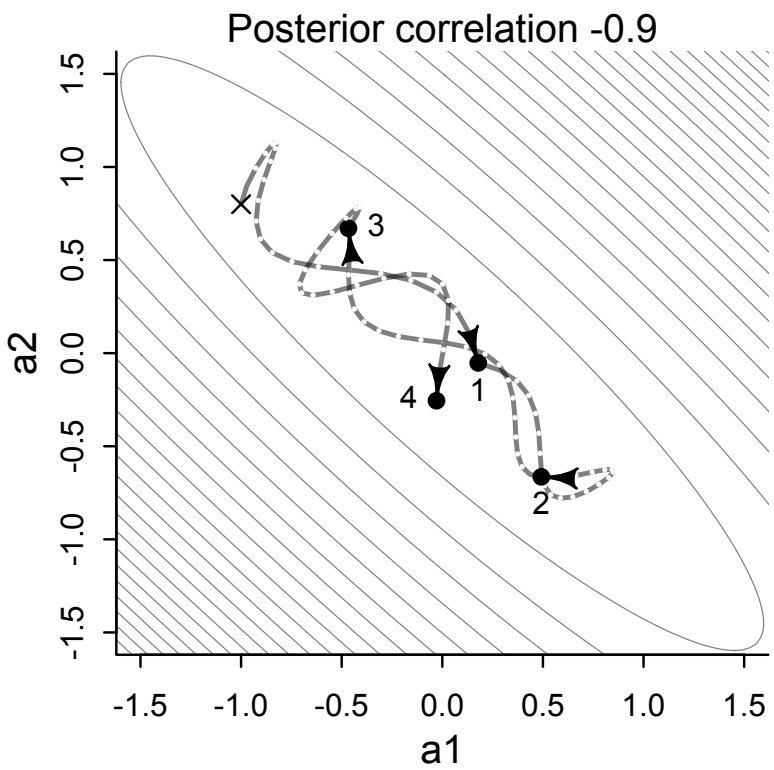
https://chi-feng.github.io/mcmc-demo/



https://chi-feng.github.io/mcmc-demo/

Hamiltonian Monte Carlo

- Why does HMC work much better?
- Doesn't get stuck — follows gradient
- Extra variables (momentum, energy) provide diagnostics
- But also requires more
  - Gradients — curvature of log-posterior
  - "Mass" of particle
  - Number of leaps in a single trajectory
  - Size of individual leaps
- These need to be tuned right
- Gradients are unique to each model



R code 9.3

```
# U needs to return neg-log-probability
myU4 <- function( q , a=0 , b=1 , k=0 , d=1 ) {
    muy <- q[1]
    mux <- q[2]
    U <- sum( dnorm(y,muy,1,log=TRUE) ) + sum( dnorm(x,mux,1,log=TRUE) ) +
        dnorm(muy,a,b,log=TRUE) + dnorm(mux,k,d,log=TRUE)
    return( -U )
}
```

Now the gradient function requires two partial derivatives. Luckily, Gaussian derivatives are very clean. The derivative of the logarithm of any univariate Gaussian with mean a and standard deviation b with respect to a is:

$$\frac{\partial \log \mathrm{N}(y|a,b)}{\partial a} = \frac{y-a}{b^2}$$

And since the derivative of a sum is a sum of derivatives, this is all we need to write the gradients:

$$\frac{\partial U}{\partial \mu_x} = \frac{\partial \log \mathrm{N}(x|\mu_x,1)}{\partial \mu_x} + \frac{\partial \log \mathrm{N}(\mu_x|0,0.5)}{\partial \mu_x} = \sum_i \frac{x_i - \mu_x}{1^2} + \frac{0-\mu_x}{0.5^2}$$

And the gradient for μ_y has the same form. Now in code form:

R code 9.4

```
# gradient function
# need vector of partial derivatives of U with respect to vector q
myU_grad4 <- function( q , a=0 , b=1 , k=0 , d=1 ) {
    muy <- q[1]
    mux <- q[2]
    G1 <- sum( y - muy ) + (a - muy)/b^2 #dU/dmuy
    G2 <- sum( x - mux ) + (k - mux)/d^2 #dU/dmuy
    return( c( -G1 , -G2 ) ) # negative bc energy is neg-log-prob
}
# test data
set.seed(7)
y <- rnorm(50)
x <- rnorm(50)
x <- as.numeric(scale(x))
y <- as.numeric(scale(y))
```

The gradient function above isn't too bad for this model. But it can be terrifying for a reasonably complex model. That is why tools like Stan build the gradients dynamically, using the model definition. Now we are ready to visit the heart. To understand some of the details here, you should read Radford Neal's chapter in the *Handbook of Markov Chain Monte Carlo*. Armed with the log-posterior and gradient functions, here's the code to produce Figure 9.6:

R code 9.5

```
library(shape) # for fancy arrows
Q <- list()
Q$q <- c(-0.1,0.2)
pr <- 0.3
plot( NULL , ylab="muy" , xlab="mux" , xlim=c(-pr,pr) , ylim=c(-pr,pr) )
step <- 0.03
L <- 11 # 0.03/28 for U-turns --- 11 for working example
n_samples <- 4
path_col <- col.alpha("black",0.5)
points( Q$q[1] , Q$q[2] , pch=4 , col="black" )
for ( i in 1:n_samples ) {
    Q <- HMC2( myU4 , myU_grad4 , step , L , Q$q )
    if ( n_samples < 10 ) {
      for ( j in 1:L ) {
        K0 <- sum(Q$ptraj[j,]^2)/2 # kinetic energy
        lines( Q$traj[j:(j+1),1] , Q$traj[j:(j+1),2] , col=path_col , lwd=1+2*K0 )
      }
      points( Q$traj[1:L+1,] , pch=16 , col="white" , cex=0.35 )
      Arrows( Q$traj[L,1] , Q$traj[L,2] , Q$traj[L+1,1] , Q$traj[L+1,2] ,
          arr.length=0.35 , arr.adj = 0.7 )
      text( Q$traj[L+1,1] , Q$traj[L+1,2] , i , cex=0.8 , pos=4 , offset=0.4 )
    }
    points( Q$traj[L+1,1] , Q$traj[L+1,2] , pch=ifelse( Q$accept==1 , 16 , 1 ) ,
        col=ifelse( abs(Q$dH)>0.1 , "red" , "black" ) )
}
```


The function `HMC2` is built into `rethinking`. It is based upon one of Radford Neal's example scripts.[140] It isn't actually too complicated. Let's tour through it, one step at a time, to take the magic away. This function runs a single trajectory, and so produces a single sample. You need to use it repeatedly to build a chain. That's what the loop above does. The first chunk of the function chooses random momentum—the flick of the particle—and initializes the trajectory.

R code 9.6

```
HMC2 <- function (U, grad_U, epsilon, L, current_q) {
  q = current_q
  p = rnorm(length(q),0,1) # random flick - p is momentum.
  current_p = p
  # Make a half step for momentum at the beginning
  p = p - epsilon * grad_U(q) / 2
  # initialize bookkeeping - saves trajectory
  qtraj <- matrix(NA,nrow=L+1,ncol=length(q))
  ptraj <- qtraj
  qtraj[1,] <- current_q
  ptraj[1,] <- p
```

Then the action comes in a loop over leapfrog steps. `L` steps are taken, using the gradient to compute a linear approximation of the log-posterior surface at each point.

R code 9.7

```
  # Alternate full steps for position and momentum
  for ( i in 1:L ) {
    q = q + epsilon * p # Full step for the position
    # Make a full step for the momentum, except at end of trajectory
    if ( i!=L ) {
        p = p - epsilon * grad_U(q)
        ptraj[i+1,] <- p
    }
    qtraj[i+1,] <- q
  }
```

Notice how the step size `epsilon` is added to the position and momentum vectors. It is in this way that the path is only an approximation, because it is a series of linear jumps, not an actual smooth curve. This can have important consequences, if the log-posterior bends sharply and the simulation jumps over a bend. All that remains is clean up: ensure the proposal is symmetric so the Markov chain is valid and decide whether to accept or reject the proposal.

R code 9.8

```
  # Make a half step for momentum at the end
  p = p - epsilon * grad_U(q) / 2
  ptraj[L+1,] <- p
```

The U-Turn Problem

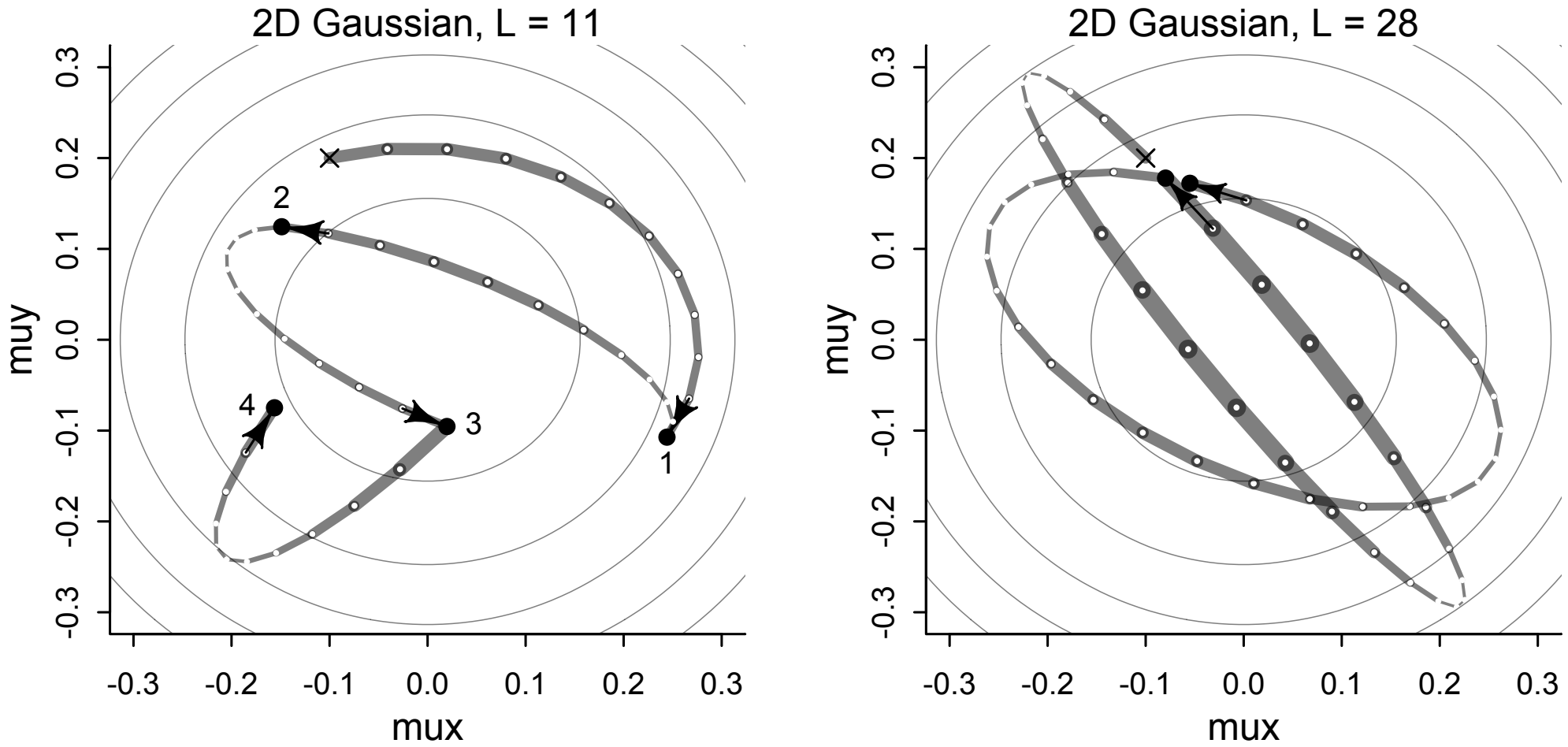
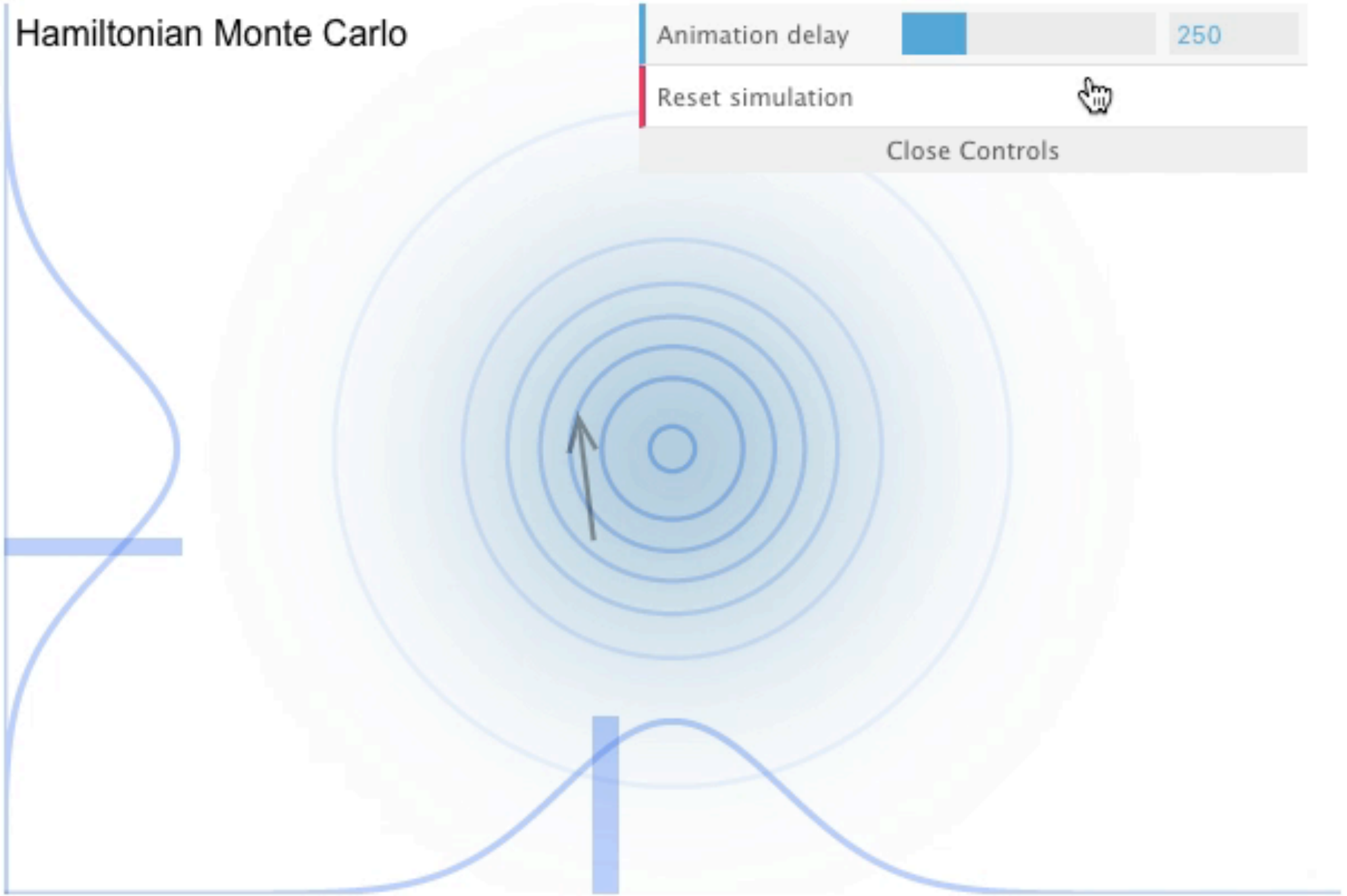


Figure 9.6

https://chi-feng.github.io/mcmc-demo/



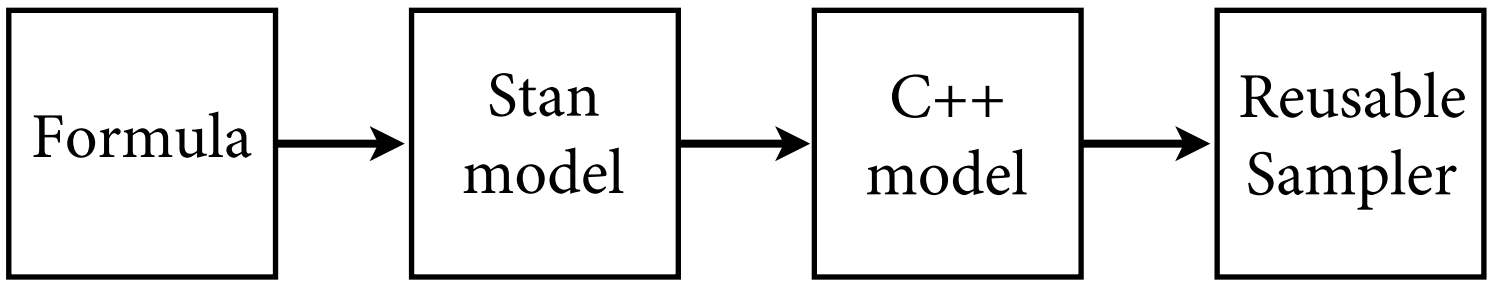
https://chi-feng.github.io/mcmc-demo/

Stan is NUTS

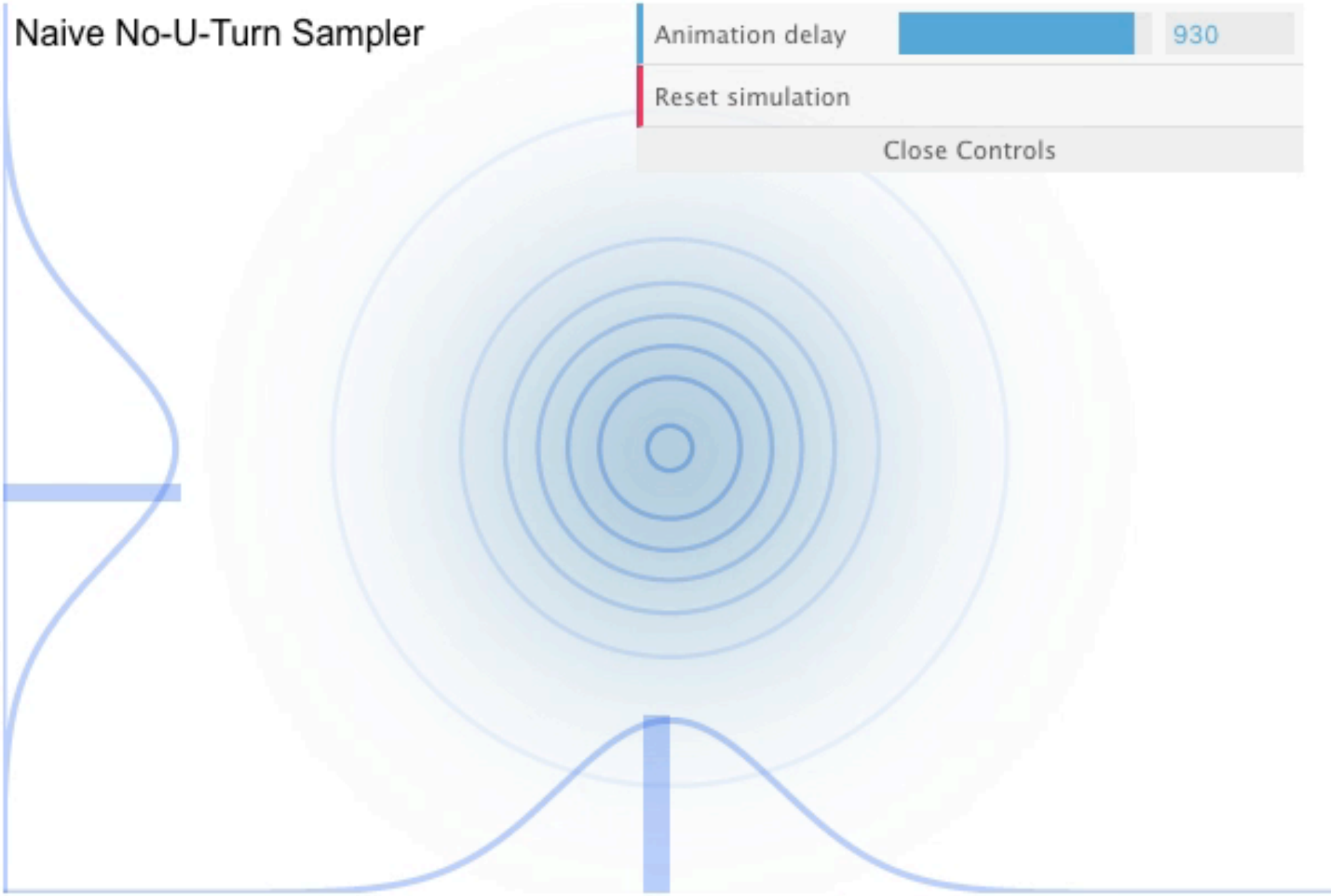
No U-Turn Sampler

Automatic Step Size and Number Adaptation

- No U-Turn Sampler (NUTS2): Adaptive Hamiltonian Monte Carlo
- Implemented in Stan (rstan: mc-stan.org)
- Stan figures out gradient for you
  - autodiff, back-propagation

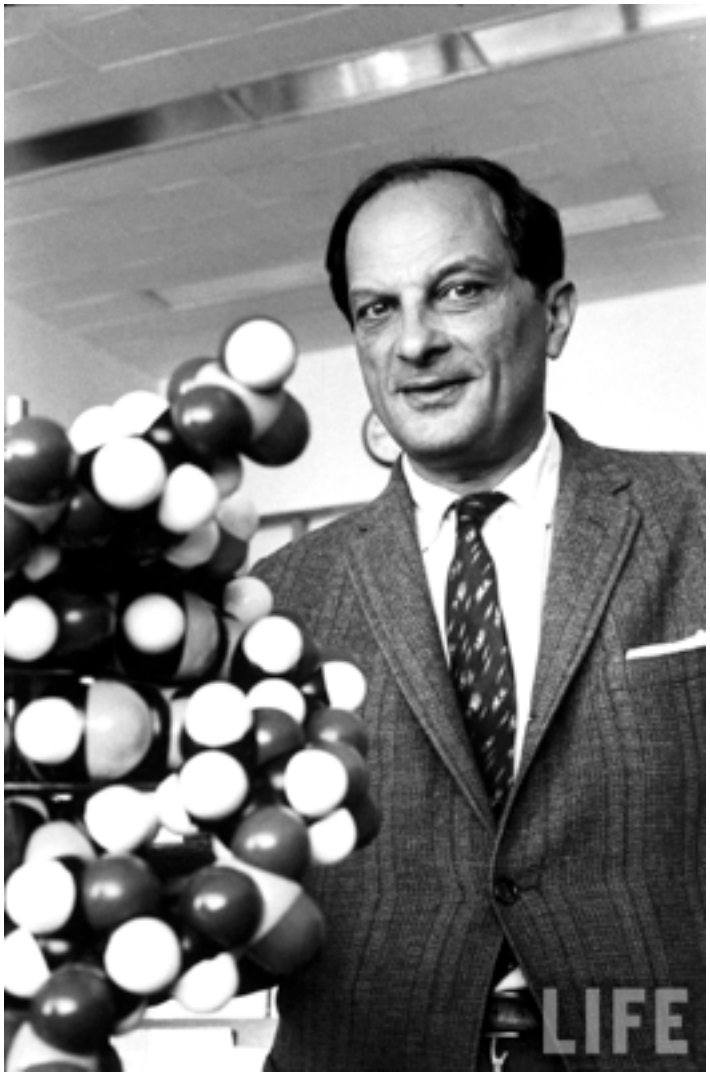


https://chi-feng.github.io/mcmc-demo/



https://chi-feng.github.io/mcmc-demo/

mc-stan.org



Stanislaw Ulam (1909–1984)

Interfaces

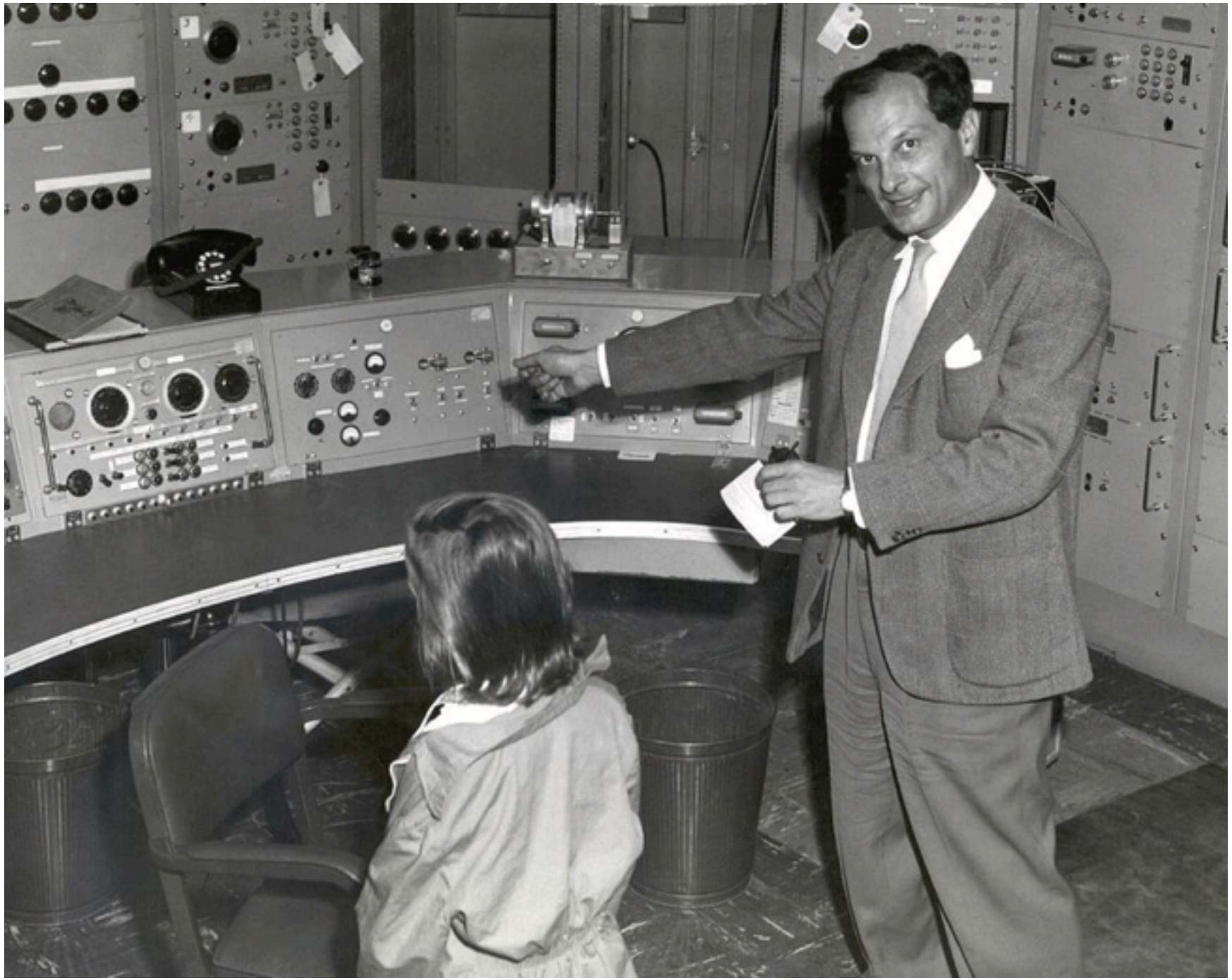
ways to run Stan

Stan Interfaces

The Stan modeling language and statistical algorithms are exposed through interfaces into many popular computing environments.

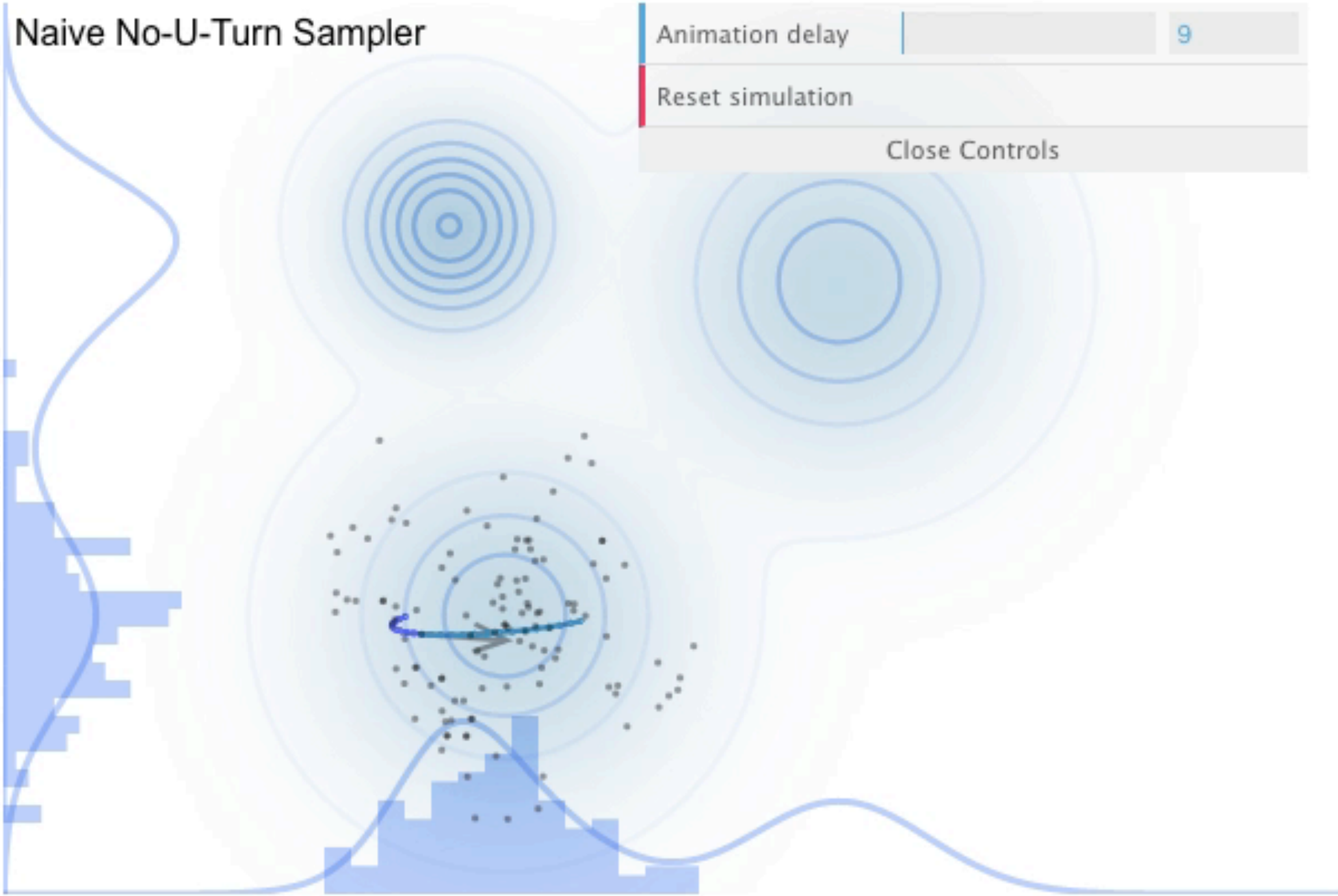
- RStan (R)
- PyStan (Python)
- CmdStan (shell, command-line terminal)
- MatlabStan (MATLAB)
- Stan.jl (Julia)
- StataStan (Stata)
- MathematicaStan (Mathematica)

Programs written in the Stan modeling language are portable across interfaces.



Stanislaw Ulam and his daughter Claire with MANIAC

https://chi-feng.github.io/mcmc-demo/



https://chi-feng.github.io/mcmc-demo/

HMC Praxis

- Back to terrain ruggedness
- Re-approximate posterior
- Practical details
- What to expect from healthy Markov chains

One hand QUAP'ing

R code 9.10

```
m8.5 <- quap(
    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=dd )
precis( m8.5 , depth=2 )
```

```
       mean   sd  5.5% 94.5%
a[1]   0.89 0.02  0.86  0.91
a[2]   1.05 0.01  1.03  1.07
b[1]   0.13 0.07  0.01  0.25
b[2]  -0.14 0.05 -0.23 -0.06
sigma  0.11 0.01  0.10  0.12
```

Hamiltonian Flows

- Interface to Stan: ulam

code 9.11

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

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 )
    ) ,
    data=dat_slim , chains=4 , cores=4 , iter=1000 )
```

R code 9.14

Hamiltonian Flows

- What happens when you use ulam?
  - Translates formula into raw Stan model code
  - Stan then builds a custom NUTS sampler
  - Sampler runs
  - Samples fed back to R

```
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 )
    ) ,
    data=dat_slim , chains=4 , cores=4 , iter=1000 )
```

R code 9.14

Hamiltonian Flows

```
    data=dat_slim , chains=4 , cores=4 , iter=1000 )
```

```
show( m9.1 )
```

```
Hamiltonian Monte Carlo approximation
2000 samples from 4 chains

Sampling durations (seconds):
         warmup sample total
chain:1   0.12   0.08  0.20
chain:2   0.12   0.08  0.19
chain:3   0.12   0.08  0.20
chain:4   0.12   0.08  0.20
```

- Num samples = total minus warmup
- Default warmup is half

Hamiltonian Flows

R code 9.16

```
precis( m9.1 , 2 )
```

```
       mean   sd  5.5% 94.5% n_eff Rhat
sigma  0.11 0.01  0.10  0.12  2641    1
b[1]   0.13 0.07  0.02  0.26  2484    1
b[2]  -0.14 0.05 -0.23 -0.06  2546    1
a[1]   0.89 0.02  0.86  0.91  3331    1
a[2]   1.05 0.01  1.03  1.07  3243    1
```

- n_eff: number of effective samples
  - can be larger than actual samples!
- Rhat: Convergence diagnostic — “1” is good

```
pairs( m9.1 )
```

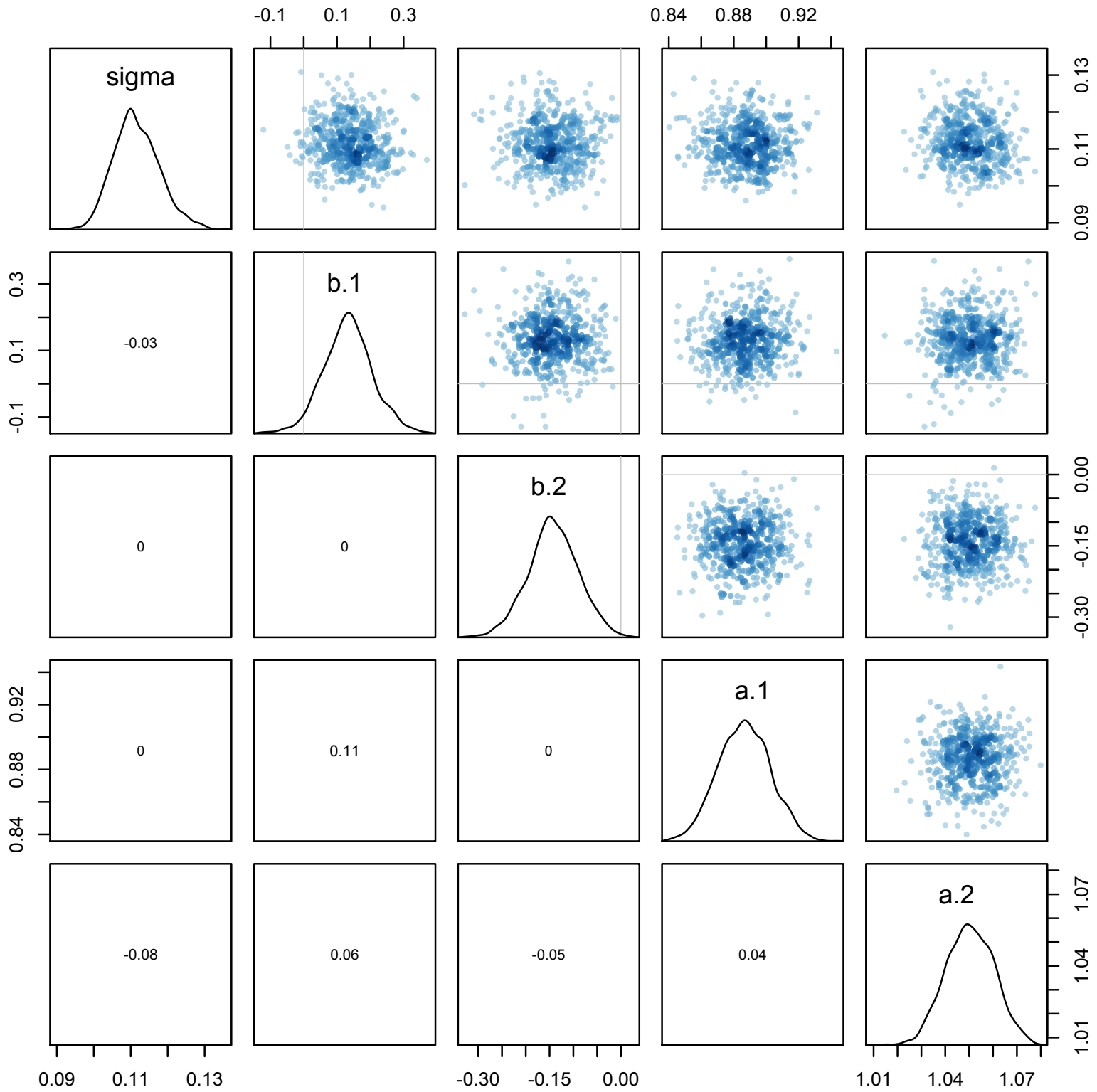


Figure 9.7

Check the chains

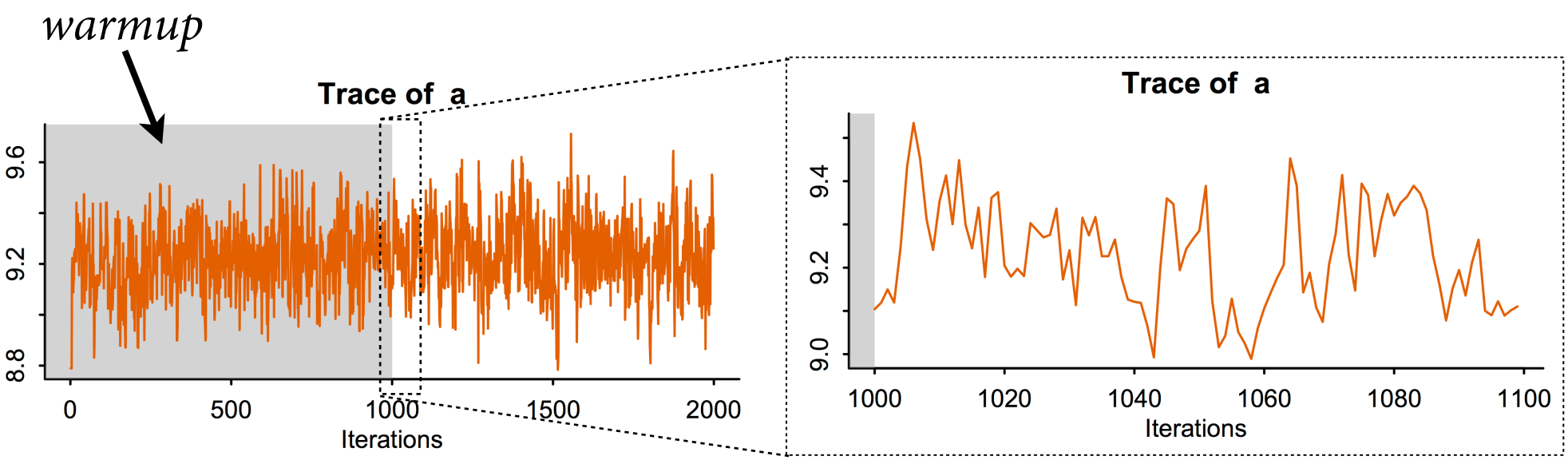
- Sometimes it doesn't work
- Good chains:
  - Converge to same target distribution
  - Once there, explore efficiently
- Different ways to check
  - Trace plots
  - Convergence diagnostics (n_eff, Rhat)
  - Special warnings (divergent transitions)

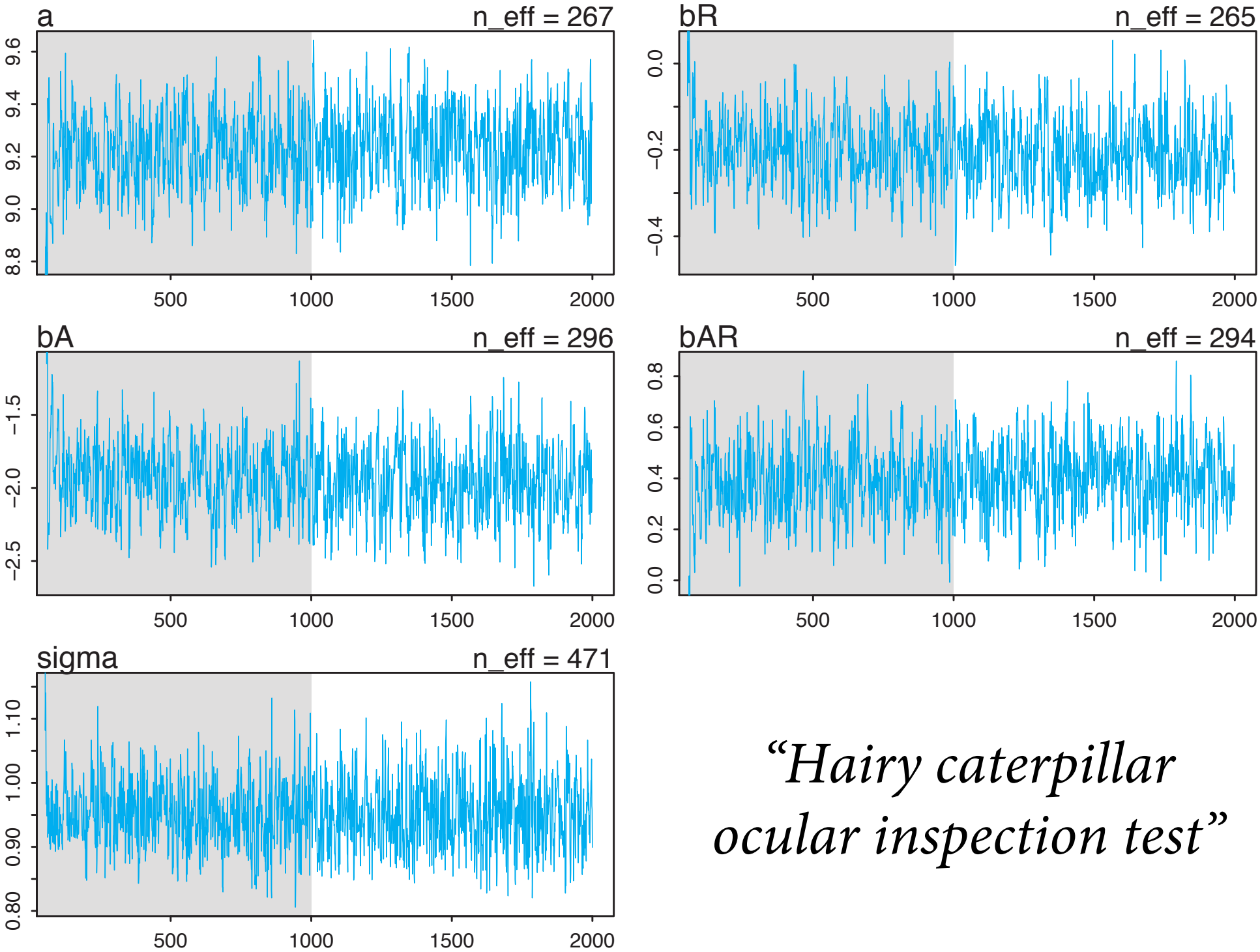
Check the chains

- Trace plot: Plot of sequential samples in chain
- Shows some problems, but not all
- Want to see "hairy caterpillar"

R code 9.18

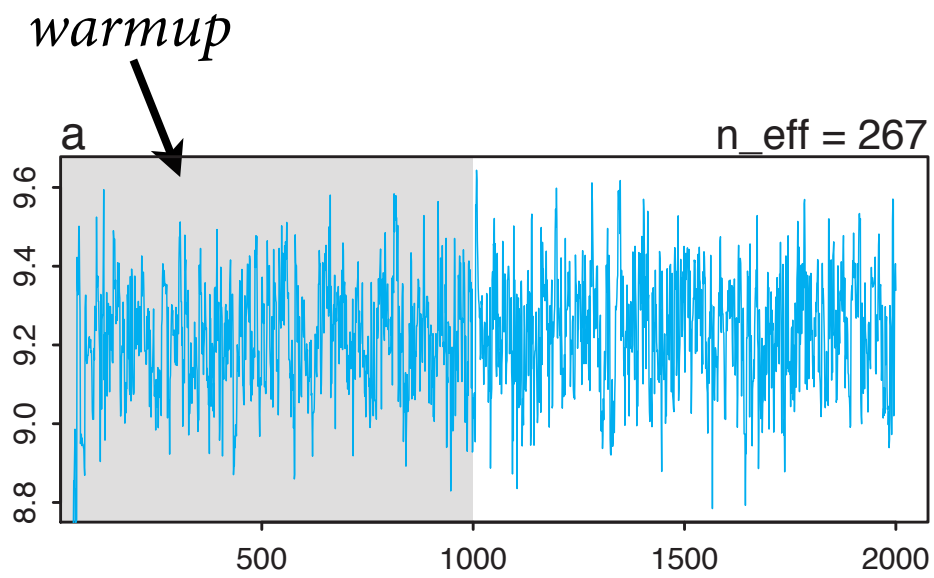
```
traceplot( m9.1 )
```





"Hairy caterpillar ocular inspection test"

Warmup



- What is “warmup”?
- Adaptation to posterior for efficient sampling
  - Figures out good step size
- Samples during warmup NOT from posterior
- Automatically discarded by precis/summary and other functions
- Warmup is NOT “burn in”

Convergence diagnostics

- n_eff: "effective" number of samples
  - n_eff/n < 0.1, be alarmed
- R-hat
  - R-hat: crudely, ratio of variance between chains to variance within chains
  - Should approach 1
- Both may mislead

R code 9.16

```
precis( m9.1 , 2 )
```

```
       mean   sd  5.5% 94.5% n_eff Rhat
sigma  0.11 0.01  0.10  0.12  2641    1
b[1]   0.13 0.07  0.02  0.26  2484    1
b[2]  -0.14 0.05 -0.23 -0.06  2546    1
a[1]   0.89 0.02  0.86  0.91  3331    1
a[2]   1.05 0.01  1.03  1.07  3243    1
```

A wild chain

- Two observations: {–1,1}
- Estimate mean and standard deviation

R code 9.19

```
y <- c(-1,1)
set.seed(11)
m9.2 <- ulam(
    alist(
        y ~ dnorm( mu , sigma ) ,
        mu <- alpha ,
        alpha ~ dnorm( 0 , 1000 ) ,
        sigma ~ dexp( 0.0001 )
    ) ,
    data=list(y=y) , chains=2 )
```

A wild chain

R code 9.19

```
y <- c(-1,1)
set.seed(11)
m9.2 <- ulam(
    alist(
        y ~ dnorm( mu , sigma ) ,
        mu <- alpha ,
        alpha ~ dnorm( 0 , 1000 ) ,
        sigma ~ dexp( 0.0001 )
    ) ,
    data=list(y=y) , chains=2 )
```

R code 9.20

```
precis( m9.2 )
```

```
        mean      sd    5.5%   94.5% n_eff Rhat
sigma 596.99 1542.85    2.17 2461.44    57 1.06
alpha -59.45  355.53 -811.45  359.49    32 1.05
```

A wild chain

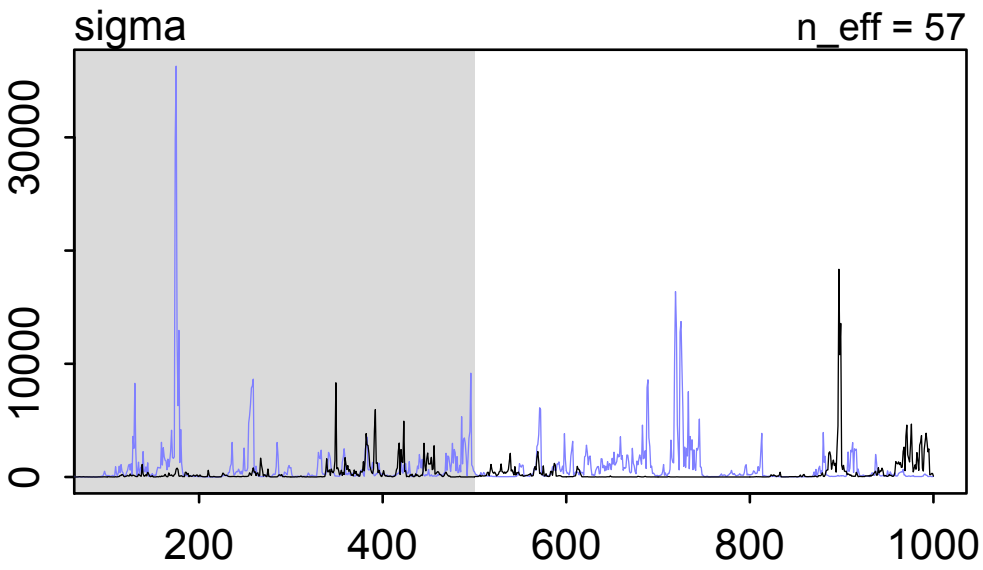
```
precis( m9.2 )
```

R code 9.20

```
        mean      sd    5.5%   94.5% n_eff Rhat
sigma 596.99 1542.85    2.17 2461.44    57 1.06
alpha -59.45  355.53 -811.45  359.49    32 1.05
```

```
Warning messages:
1: There were 70 divergent transitions after warmup. Increasing adapt_delta
above 0.95 may help. See
http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
```

- What the what is a **divergent transition**?
- Hamiltonian approximation "broke"
- Chain has trouble exploring some part of posterior


sigma
n_eff = 57
30000
10000
0
200
400
600
800
1000

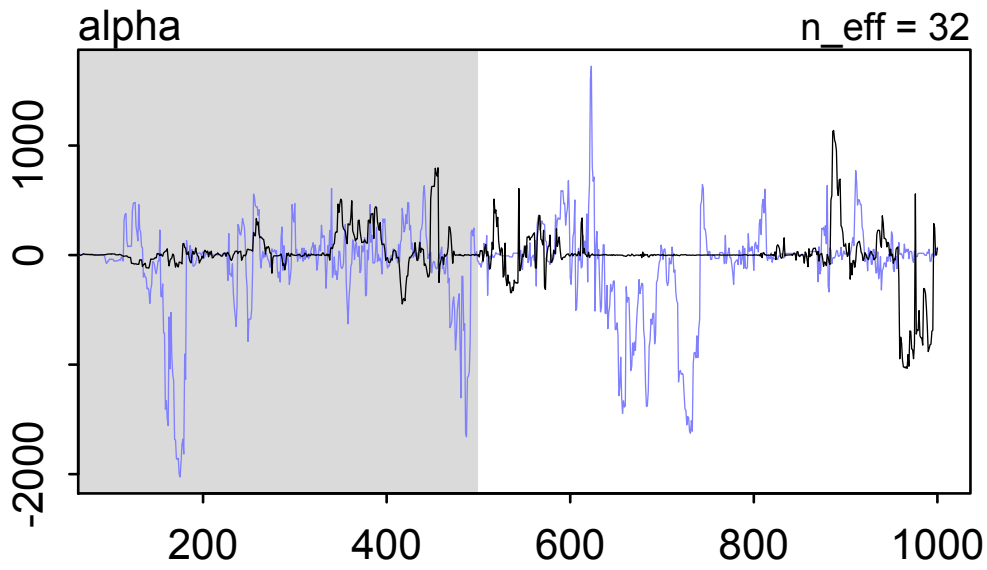
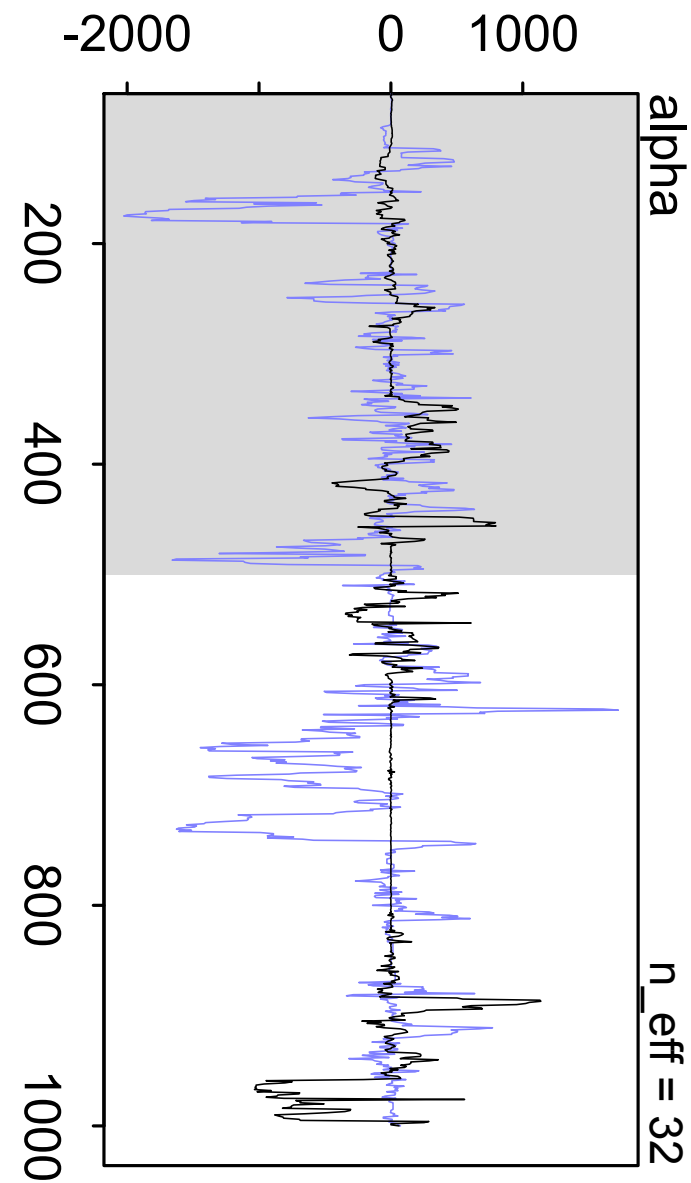

alpha
n_eff = 32
1000
0
-2000
200
400
600
800
1000

Figure 9.9

A wild chain

- Problem is flat priors
  - Flat means flat forever
  - Much probability out to thousands
  - Also a problem in BUGS/JAGS
- Fix with weakly informative priors

$$y_i \sim \text{Normal}(\mu, \sigma)$$
$$\mu = \alpha$$
$$\alpha \sim \text{Normal}(1, 10)$$
$$\sigma \sim \text{Exponential}(1)$$



A tame chain

R code 9.21

```
set.seed(11)
m9.3 <- ulam(
    alist(
        y ~ dnorm( mu , sigma ) ,
        mu <- alpha ,
        alpha ~ dnorm( 1 , 10 ) ,
        sigma ~ dexp( 1 )
    ) ,
    data=list(y=y) , chains=2 )
precis( m9.3 )
```

```
      mean   sd  5.5% 94.5% n_eff Rhat
sigma 1.53 0.75  0.66  2.86   317    1
alpha 0.04 1.14 -1.71  1.80   284    1
```

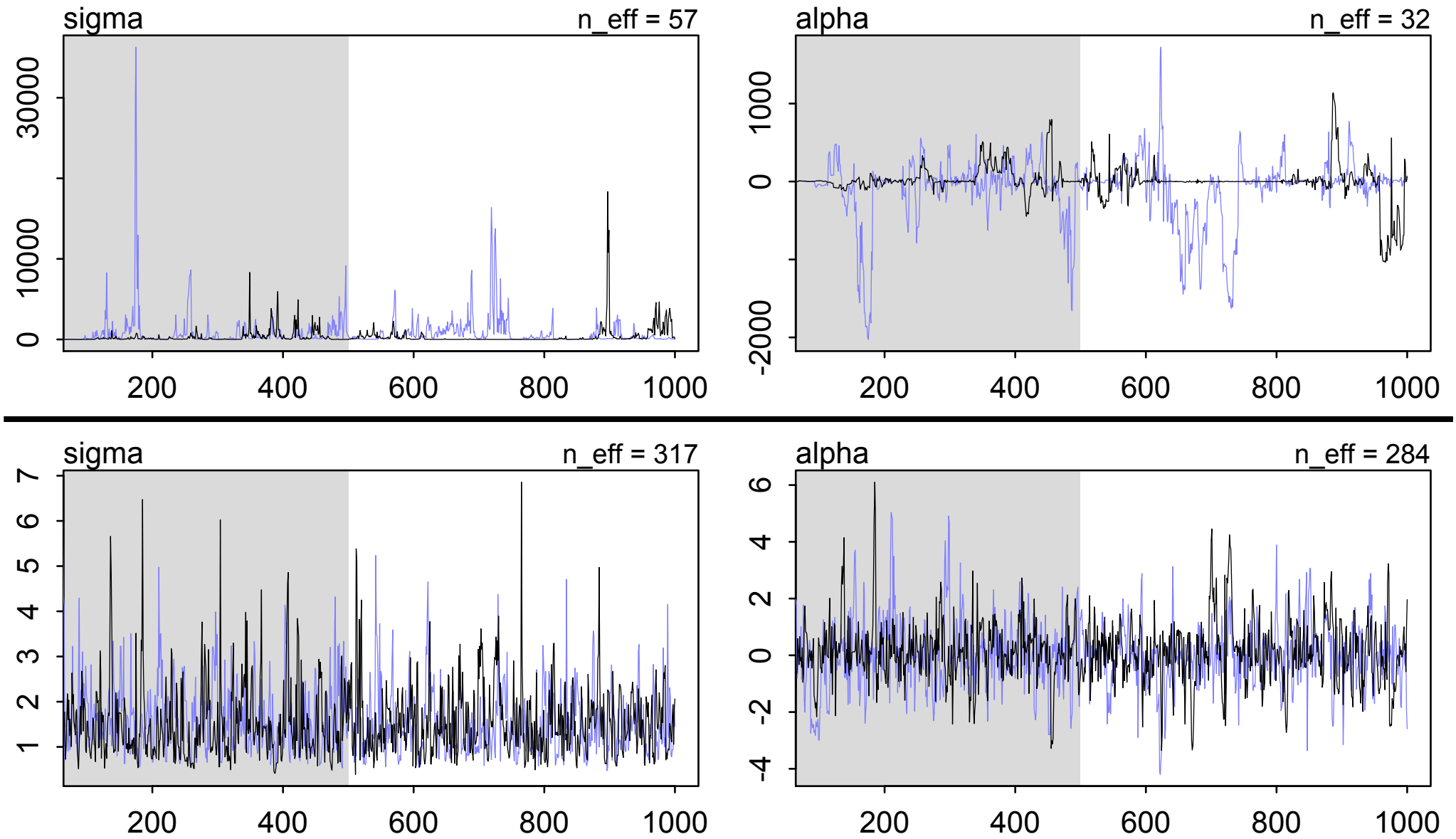
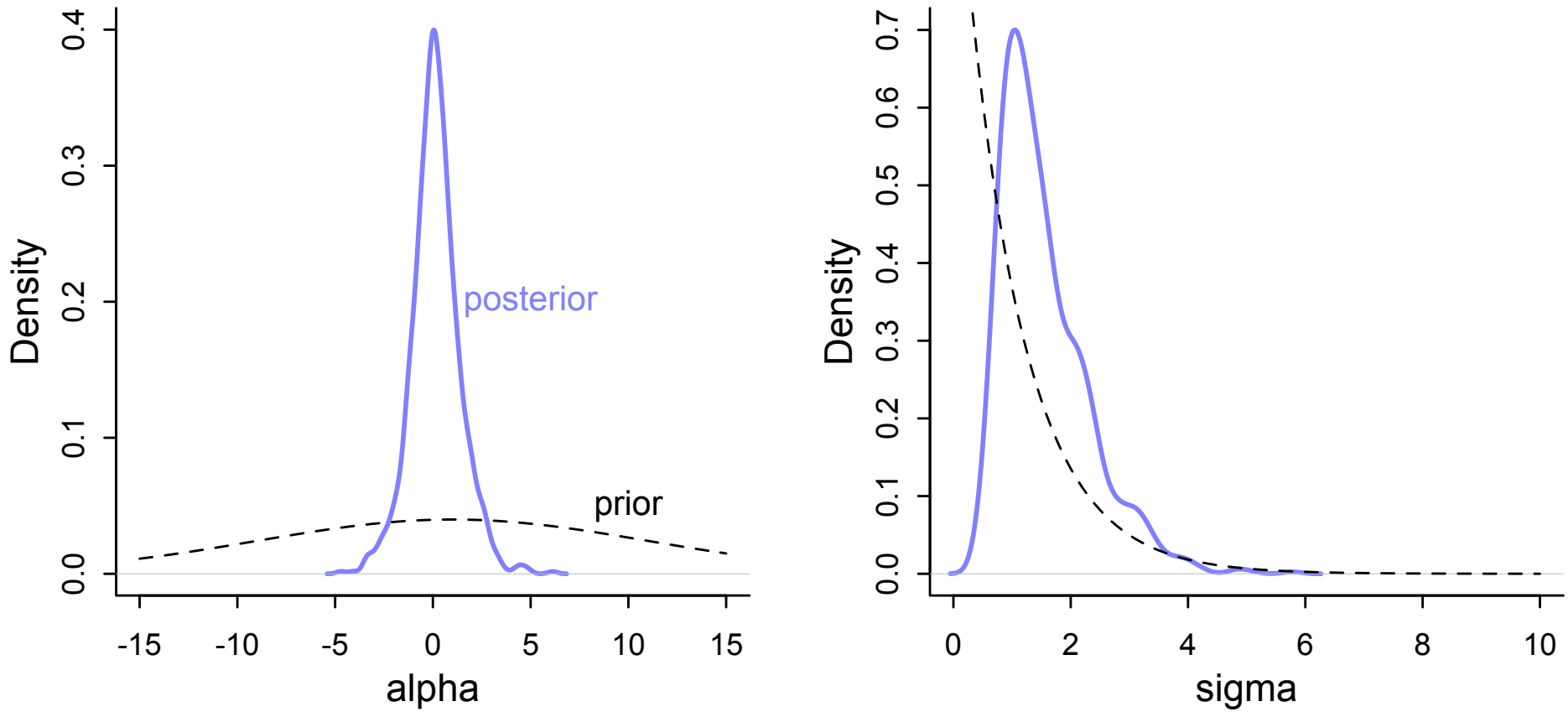

sigma
n_eff = 57
alpha
n_eff = 32
sigma
n_eff = 317
alpha
n_eff = 284
200
400
600
800
1000

Figure 9.9

A tame chain



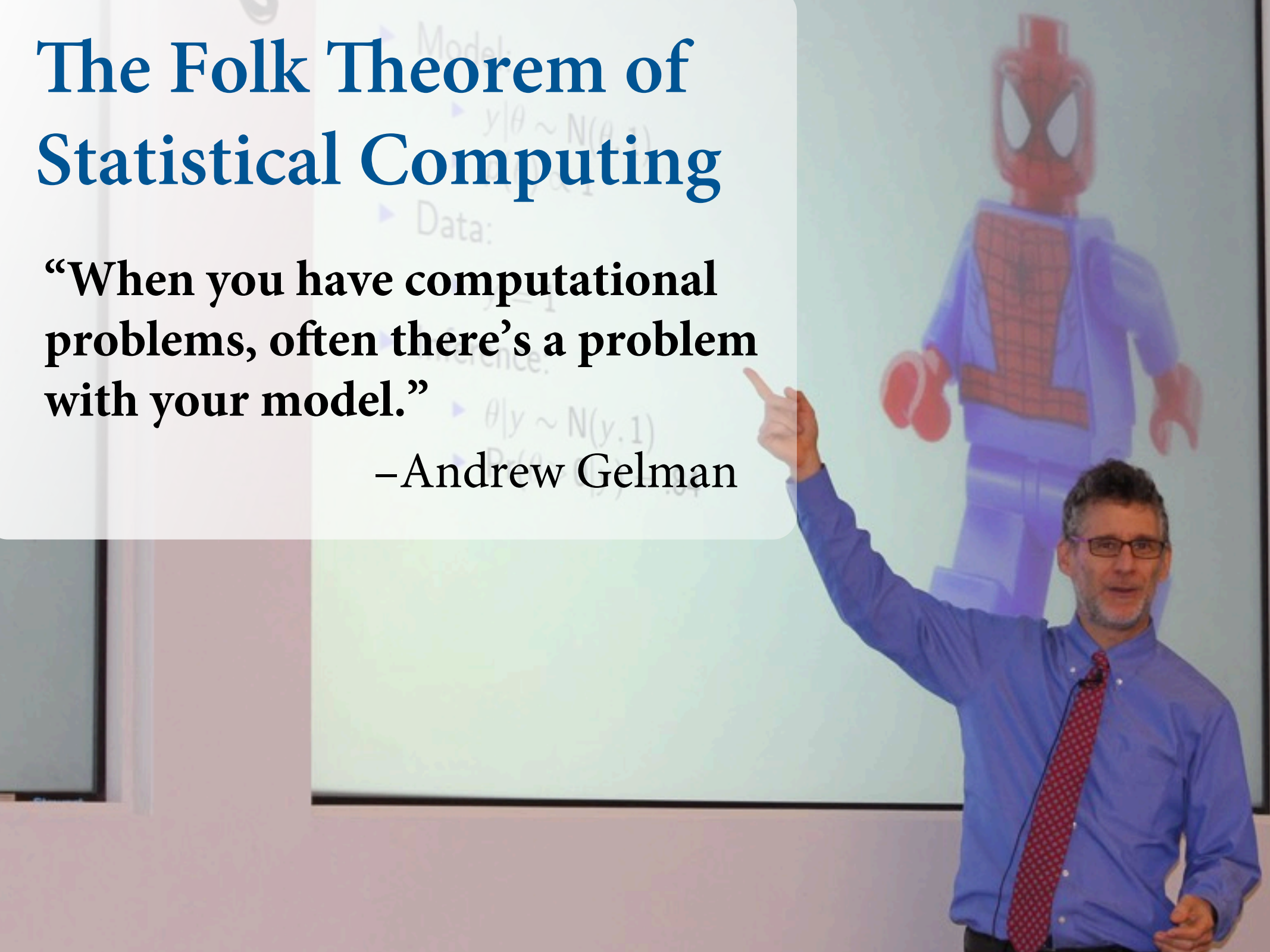
Even with only 2 observations, these priors have no effect on inference! Except to allow you to make inferences...

Figure 9.10

The Folk Theorem of Statistical Computing

"When you have computational problems, often there's a problem with your model."

–Andrew Gelman



Unidentified chains

$$y_i \sim \text{Normal}(\mu, \sigma)$$
$$\mu = \alpha_1 + \alpha_2$$
$$\sigma \sim \text{Exponential}(1)$$

R code 9.22

```
set.seed(41)
y <- rnorm( 100 , mean=0 , sd=1 )
```

R code 9.23

```
m9.4 <- ulam(
    alist(
        y ~ dnorm( mu , sigma ) ,
        mu <- a1 + a2 ,
        a1 ~ dnorm( 0 , 1000 ),
        a2 ~ dnorm( 0 , 1000 ),
        sigma ~ dexp( 1 )
    ) ,
    data=list(y=y) , chains=2 )
precis( m9.4 )
```

```
        mean      sd    5.5%  94.5% n_eff Rhat
sigma   1.05    0.08    0.91   1.17     5 1.32
a2     21.46 291.97 -473.77 475.59     2 2.01
a1    -21.27 291.96 -475.54 473.96     2 2.01
```

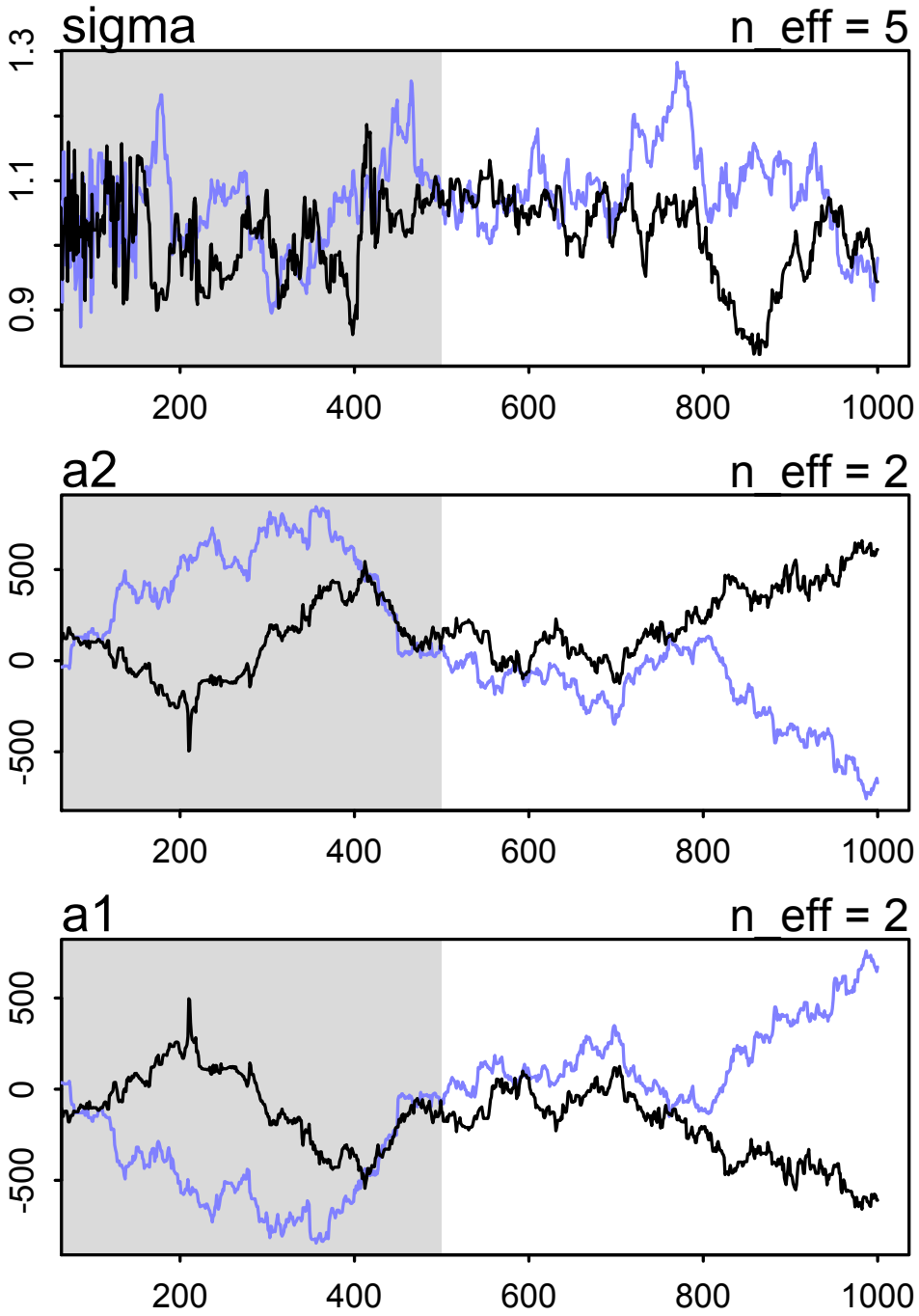

sigma
n_eff = 5
1.3
1.1
0.9
200
400
600
800
1000
a2
n_eff = 2
500
0
-500
200
400
600
800
1000
a1
n_eff = 2
500
0
-500
200
400
600
800
1000

Figure 9.11

Unidentified chains

R code 9.24

```
m9.5 <- ulam(
    alist(
        y ~ dnorm( mu , sigma ) ,
        mu <- a1 + a2 ,
        a1 ~ dnorm( 0 , 10 ),
        a2 ~ dnorm( 0 , 10 ),
        sigma ~ dexp( 1 )
    ) ,
    data=list(y=y) , chains=2 )
precis( m9.5 )
```

```
       mean   sd   5.5% 94.5% n_eff Rhat
sigma  1.04 0.08   0.93  1.18   287    1
a2     0.22 7.23 -10.67 12.26   244    1
a1    -0.03 7.22 -12.09 10.87   245    1
```

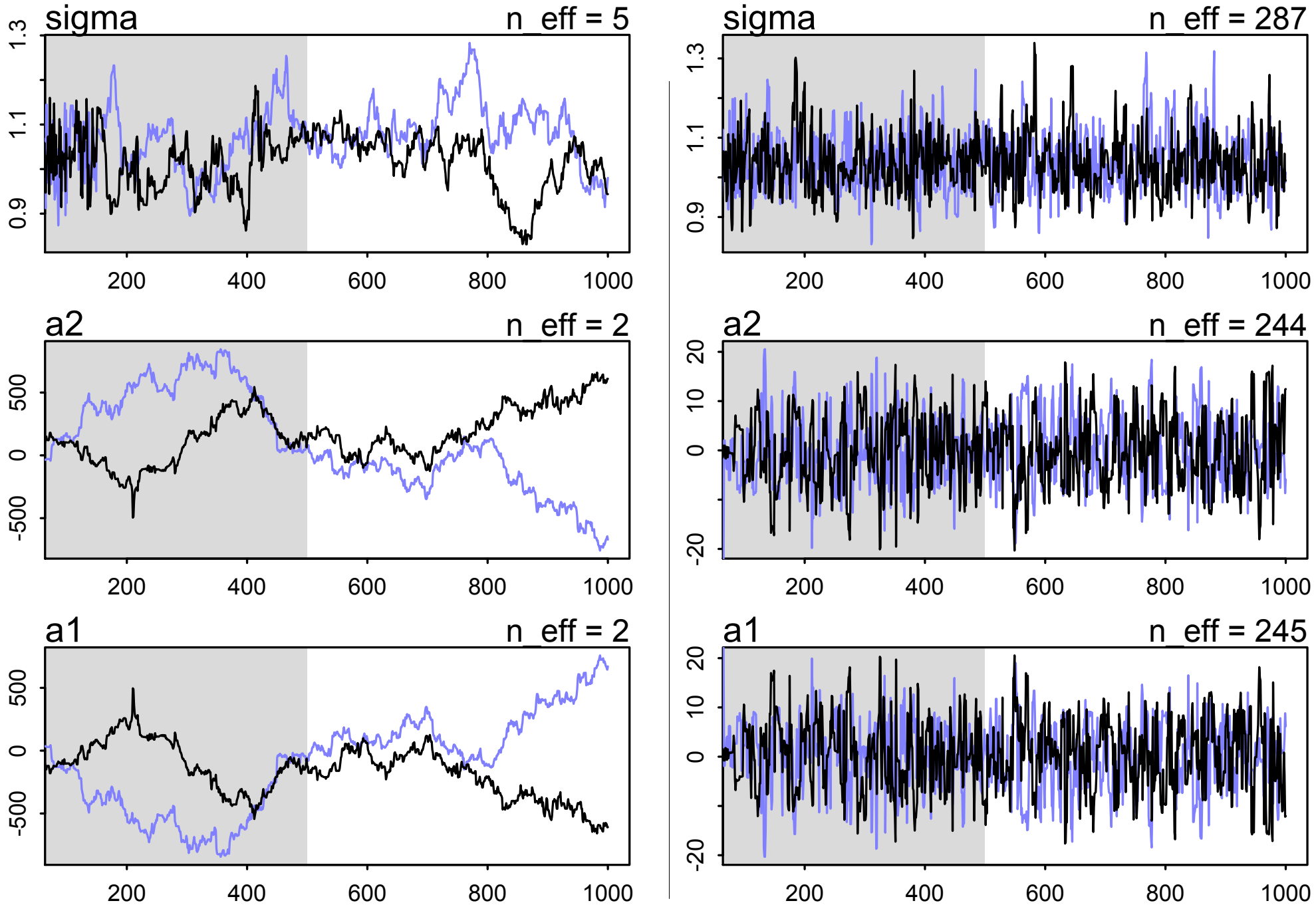

sigma
n_eff = 5
sigma
n_eff = 287
a2
n_eff = 2
a2
n_eff = 244
a1
n_eff = 2
a1
n_eff = 245
200
400
600
800
1000

Figure 9.11

Homeward

- Updated book PDF (20 Jan) & rethinking 1.82
- data(Wines2012): Interactions and Markov chains
- Next week:
  Maximizing Entropy for Inferential Justice
  Generalized Linear Models (GLMs)