BAYESIAN INFERENCE IN STAN

DANIEL LEE

bearlee@alum.mit.edu

http://mc-stan.org

OBLIGATORY DISCLOSURE

- I am a researcher at Columbia University.
- I am a cofounder of Stan Group Inc. and have equity.
- I am required to disclose this information.

WHAT WE'LL COVER

- What is Stan?
- What is Stan good for?
- What is Stan not good?

- Usage from R using RStan
- How to get help after the course is over.

- Goal: turn everyone into a Stan user!
  (or at least eliminate enough hurdles to becoming one)

WHAT WE WON'T COVER

- Full blown stats course
- All details of numeric computation
- Every reason Stan is different from _____
- Deep details about implementation of Stan

WHO AM I

- Stan developer
  - Since 2011
- I work on
  - the math library (C++, reverse-mode autodiff)
  - the Stan library (language and algorithms)
  - CmdStan, RStan, and PyStan
- Collaborate with others

HOW TO GET THE MOST OUT OF THE COURSE

- Ask questions
- Try the exercises
- Talk to each other

STAN EXAMPLE

RSTAN INSTALLATION CHECK

RSTAN INSTALLATION: V2.11.1

Who's having trouble?

- Check that version is 2.11.1.

```
> sessionInfo()
```

- Windows:
  - Updated path when installing RTools?
  - RTools must match R version
- Mac
  - Accept Xcode license?
- Linux
  - You're on your own! j/k
  - What compiler are you using?

SETUP: LOAD RSTAN PACKAGE

1. Pick a working folder.

   For me, it's ~/dev/alaska

2. Open R / RStudio.

3. Set your working directory in R.

```
setwd("~/dev/alaska")
```

4. Load RStan

```
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

FIRST EXAMPLE: 8 SCHOOLS

- Data from 8 schools.
  Study from Educational Testing Services (ETS) to analyze effect of SAT coaching
- SAT-V in eight high schools
- No prior reason to believe any program was
  - more effective than others
  - more similar than others

Rubin, 1981. “Estimation in parallel randomized experiments”

DATA

School	Estimated treatment effect	Standard error (of effect)
A	28	15
B	8	10
C	-3	16
D	7	11
E	-1	9
F	1	11
G	18	10
H	12	18

WHAT'S THE MODEL?

- Data

$$y_j, \sigma_j$$

- Parameters

$$\theta_j, \mu, \tau > 0$$

- Statistical model

$$y_j \sim \text{Normal}(\theta_j, \sigma_j)$$

$$\theta_j \sim \text{Normal}(\mu, \tau)$$

TYPE THE DATA

- In R

```
J <- 8
y <- c(28, 8, -3, 7, -1, 1, 18, 12)
sigma <- c(15, 10, 16, 11, 9, 11, 10, 18)
```

- Why am I typing the data?

WRITE THE STAN PROGRAM

- Type this into a file, eight_schools.stan

```
data {
  int<lower=0> J;
  real y[J];
  real<lower=0> sigma[J];
}
parameters {
  real mu;
  real<lower=0> tau;
  real theta[J];
}
model {
  for (j in 1:J)
    y[j] ~ normal(theta[j], sigma[j]);
  for (j in 1:J)
    theta[j] ~ normal(mu, tau);
}
```

RUN THE PROGRAM

- From R:

```
library(rstan)
rstan_options(auto_writer = TRUE)
options(mc.cores = parallel::detectCores())



fit <- stan("eight_schools.stan")
```

SUMMARY OF THE FIT

- fit
- print(fit)

```
> fit
Inference for Stan model: eight_schools.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

           mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
mu         8.64    0.85 5.53  -1.93   4.92   8.48  12.78  18.30    42 1.08
tau        7.57    0.59 5.80   1.62   3.16   6.19  10.26  22.68    95 1.05
theta[1]  12.98    0.37 8.83  -2.38   6.82  12.37  17.35  33.19   568 1.02
theta[2]   8.87    0.95 6.83  -4.73   4.23   8.84  14.01  21.13    51 1.07
theta[3]   6.30    0.79 8.21 -12.77   1.62   7.00  12.65  20.73   107 1.04
theta[4]   8.38    0.71 6.92  -5.86   3.79   8.66  14.03  21.74    94 1.04
theta[5]   5.27    1.04 6.89  -9.31   0.78   5.72  10.44  15.77    44 1.10
theta[6]   6.32    0.89 7.32 -10.34   1.79   6.61  11.80  19.06    68 1.06
theta[7]  11.71    0.24 6.79  -0.50   6.94  11.94  15.15  26.24   778 1.01
theta[8]   9.36    1.12 9.08  -9.46   3.98   8.78  15.71  27.39    66 1.06
lp__     -18.66    0.59 4.96 -28.32 -22.26 -18.77 -14.47 -10.51    71 1.07

Samples were drawn using NUTS(diag_e) at Mon Aug 22 16:17:10 2016.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

SUMMARY

- RStan installed!
- Wrote data in R
- Passed data to RStan directly from R
- Fit a hierarchical model
  (This one has problems. We'll get to that later.)

Coming up next

- What problem are we solving?

BAYESIAN INFERENCE IN STAN

BAYESIAN INFERENCE

WHAT ARE WE TRYING TO DO?

WHAT ARE WE TRYING TO DO?

- Assume we have data.
  Let's call the observations y
  and the predictors x

- We can imagine some description of the data.
  This is our statistical model.

- The statistical model has parameters θ
  The joint probability distribution function is $p(\theta, y, x)$

- We often write this as
  $$p(\theta, y, x) = p(y, x \mid \theta)\, p(\theta)$$

POINTS OF CONFUSION; OTHER NOTATION

- I'm a statistician. Statisticians overload $p(\cdot)$ all the time. They're usually all different.
- The vertical bar is read as "given"
- All probability distributions integrates to 1. Finding normalizing constants is hard. $\int p(x)dx = 1$
- Likelihood functions $\mathcal{L}(\theta) = p(y, x \mid \theta)$ aren't probability distributions over the parameters. (What is $p(y, x \mid \theta)$ a distribution over?)
- Expectation is $\mathbb{E}[f(\theta)] = \int f(\theta)\, p(\theta)\, d\theta$

NORMALIZING CONSTANTS ARE HARD

- It's easy enough specifying conditional distributions to build up a joint distribution
- The normalizing constants are often intractable

$$\int p(y, x, \theta) d\theta = \int p(y, x, \mid \theta) p(\theta) d\theta$$

- Conjugate models are a class of likelihood + prior pairs that result in an analytic posterior

INFERENCE

- observations
  predictors
  parameters
  joint model

$$y$$
$$x$$
$$\theta$$
$$p(y, x, \theta) = p(y, x \mid \theta)\, p(\theta)$$

- Frequentists want
  (single value)

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, p(x, y \mid \theta)$$

- Bayesians want
  (probability distribution)
  (use in expectations)

$$p(\theta \mid y, x) = \frac{p(y, x, \theta)}{\int p(y, x, \theta) d\theta}$$

EXAMPLE:
MALE BIRTH RATIO

BIRTH RATE BY SEX

- Laplace's data on live births in Paris from 1745 – 1770:

sex	live births
male	241,945
female	251,527

- Question 1 (estimation)
  What is the birth rate of boys vs girls?
- Question 2 (event probability)
  Is a boy more likely to be born than a girl?
- Bayes (1763) set up the "Bayesian" model
- Laplace (1781, 1786) solved for the posterior

BAYES' BINOMIAL MODEL

- Data
  - total number of male births $y = 241,945$
  - total births $N = 493,472$
- Parameter
  - proportion of male births $\theta \in \{0, 1\}$
- Likelihood

$$p(y \mid N, \theta) = \text{Binomial}(y \mid N, \theta) = \binom{N}{y} \theta^y (1 - \theta)^{N-y}$$

- Prior

$$p(\theta) = \text{Uniform}(\theta \mid 0, 1) = 1$$

POSTERIOR IS ANALYTIC

- Laplace calculated the posterior distribution analytically

$$p(\theta \mid y, N) = \text{Beta}(\theta \mid y + 1, N - y + 1)$$

$$y = 241,945$$
$$N = 493,472$$

- Posterior mean

$$\mathbb{E}[\theta] = \frac{1 + 241,945}{1 + 241,945 + 1 + 251,527} \approx 0.4902913$$

- MLE

$$\hat{\theta} = \frac{241,945}{241,945 + 251,527} \approx 0.4902912$$

- MLE approaches posterior mean as observations increase

ANSWERING QUESTIONS

- Question 1 (estimation)
  What is the birth rate of boys vs girls?

- We can look at 99% interquartile range:

$$Q_{0.005} = \mathrm{CDF}^{-1}(0.005)$$
$$Q_{0.995} = \mathrm{CDF}^{-1}(0.995)$$

```
> qbeta(c(0.005, 0.995), 241945 + 1, 251527 + 1)
[1] 0.4884583 0.4921244
```

ANSWERING QUESTIONS

- Question 2 (event probability)
  Is a boy more likely to be born than a girl?

$$
\begin{aligned}
\Pr[\theta > 0.5] &= \int_{\Theta} \mathbb{1}[\theta > 0.5]\, p(\theta \mid y, N) d\theta \\
&= \int_{0.5}^{1} p(\theta \mid y, N) d\theta \\
&= 1 - \int_{0}^{0.5} p(\theta \mid y, N) d\theta \\
&\approx 10^{-42}
\end{aligned}
$$

MONTE CARLO INTEGRATION

MONTE CARLO INTEGRATION: SUMMARY

- Instead of an analytic distribution,
  suppose we have **i.i.d. draws** from this distribution.

- Monte Carlo is a way to use these draws to calculate any expectation with respect to the distribution.

- Why? Almost everything we're concerned with is an expectation.

Expectations of Function of R.V.

- Suppose $f(\theta)$ is a function of random variable vector θ
- Suppose the density of θ is $p(\theta)$
  - *Warning*: θ overloaded as random and bound variable
- Then $f(\theta)$ is also random variable, with expectation

$$\mathbb{E}[f(\theta)] = \int_{\Theta} f(\theta)\ p(\theta)\ d\theta.$$

  - where Θ is support of $p(\theta)$ (i.e., $\Theta = \{\theta \mid p(\theta) > 0\}$

QoI as Expectations

- Most Bayesian quantities of interest (QoI) are expectations over the posterior $p(\theta \mid y)$ of functions $f(\theta)$
- **Bayesian parameter estimation**: $\hat{\theta}$
  - $f(\theta) = \theta$
  - $\hat{\theta} = \mathbb{E}[\theta|y]$ minimizes expected square error
- **Bayesian parameter (co)variance estimation**: $\text{var}[\theta \mid y]$
  - $f(\theta) = (\theta - \hat{\theta})^2$
- **Bayesian event probability**: $\Pr[A \mid y]$
  - $f(\theta) = \text{I}(\theta \in A)$

Expectations via Monte Carlo

- Generate draws $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(M)}$ drawn from $p(\theta)$
- Monte Carlo Estimator **plugs in average** for expectation:

$$\mathbb{E}[f(\theta)|y] \approx \frac{1}{M} \sum_{m=1}^{M} f(\theta^{(m)})$$

- Can be made **as accurate as desired**, because

$$\mathbb{E}[f(\theta)] = \lim_{M \to \infty} \frac{1}{M} \sum_{m=1}^{M} f(\theta^{(m)})$$

- *Reminder*: By CLT, error goes down as $1 / \sqrt{M}$

Accuracy of Monte Carlo

- Monte Carlo is *not* an approximation!
- It can be made exact to within any ϵ
- Monte Carlo draws are i.i.d. by definition
- Central limit theorem: expected error decreases at rate of

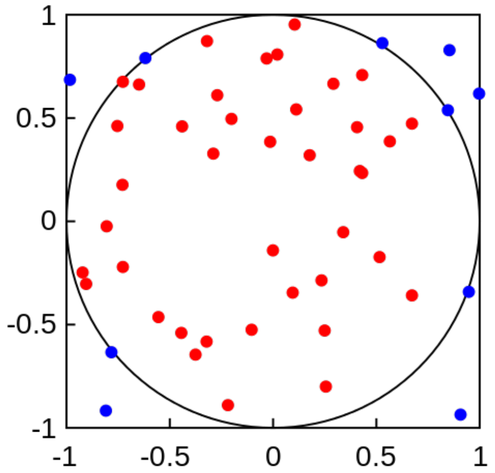
$$\frac{1}{\sqrt{N}}$$

- 3 decimal places of accuracy with sample size 1e6
- Need $100\times$ larger sample for each digit of accuracy

EXAMPLE: CALCULATE PI

Monte Carlo Calculation of π

- Computing $\pi = 3.14\ldots$ via simulation is *the* textbook application of Monte Carlo methods.
- Generate points uniformly at random within the square
- Calculate proportion within circle ($x^2 + y^2 < 1$) and multiply by square's area (4) to produce the area of the circle.
- This area is π (radius is 1, so area is $\pi r^2 = \pi$)



Plot by Mysid Yoderj courtesy of Wikipedia.

Monte Carlo Calculation of π (cont.)

- R code to calcuate π with Monte Carlo simulation:

```
> x <- runif(1e6,-1,1)
> y <- runif(1e6,-1,1)

> prop_in_circle <- sum(x^2 + y^2 < 1) / 1e6

> 4 * prop_in_circle
[1] 3.144032
```

General Monte Carlo Integration

- MC can calculate arbitrary definite integrals,

$$\int_a^b f(x)\,dx$$

- Let d upper bound $f(x)$ in (a, b); tightness determines computational efficiency
- Then generate random points uniformly in the rectangle bounded by (a, b) and $(0, d)$
- Multiply proportion of draws (x, y) where $y < f(x)$ by area of rectangle, $d \times (b - a)$.
- Can be generalized to multiple dimensions in obvious way

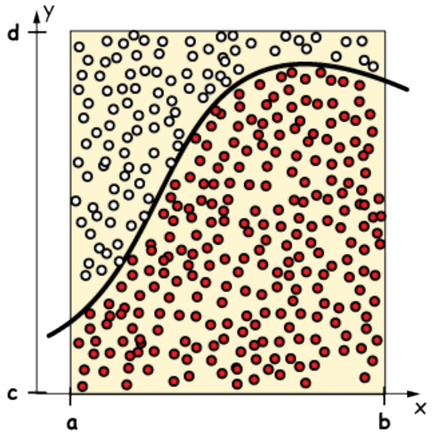


Image courtesy of Jeff Cruzan, http://www.drcruzan.com/NumericalIntegration.html

REPEAT: MONTE CARLO INTEGRATION

Expectations via Monte Carlo

- Generate draws $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(M)}$ drawn from $p(\theta)$
- Monte Carlo Estimator **plugs in average** for expectation:

$$\mathbb{E}[f(\theta)|y] \approx \frac{1}{M} \sum_{m=1}^{M} f(\theta^{(m)})$$

- Can be made **as accurate as desired**, because

$$\mathbb{E}[f(\theta)] = \lim_{M \to \infty} \frac{1}{M} \sum_{m=1}^{M} f(\theta^{(m)})$$

- *Reminder*: By CLT, error goes down as $1 / \sqrt{M}$

MARKOV CHAIN MONTE CARLO

Markov Chain Monte Carlo

- Standard Monte Carlo draws i.i.d. draws

$$\theta^{(1)}, \ldots, \theta^{(M)}$$

  according to a probability function $p(\theta)$

- Drawing an i.i.d. sample is often impossible when dealing with complex densities like Bayesian posteriors $p(\theta|y)$

- So we use Markov chain Monte Carlo (MCMC) in these cases and draw $\theta^{(1)}, \ldots, \theta^{(M)}$ from a Markov chain

Markov Chains

- A Markov Chain is a sequence of random variables

$$\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(M)}$$

  such that $\theta^{(m)}$ only depends on $\theta^{(m-1)}$, i.e.,

$$p(\theta^{(m)}|y, \theta^{(1)}, \ldots, \theta^{(m-1)}) \;=\; p(\theta^{(m)}|y, \theta^{(m-1)})$$

- Drawing $\theta^{(1)}, \ldots, \theta^{(M)}$ from a Markov chain according to $p(\theta^{(m)} \mid \theta^{(m-1)}, y)$ is more tractable
- Require marginal of each draw, $p(\theta^{(m)}|y)$, to be equal to true posterior

WHAT IS MCMC?

- Markov chain Monte Carlo is a class of algorithms
- Goal:
  Create a Markov chain where the sequence of draws is from the target distribution.
- The target distribution, for us, is the posterior distribution

$$p(\theta \mid y, x)$$

Applying MCMC

- Plug in just like ordinary (non-Markov chain) Monte Carlo
- Adjust standard errors for dependence in Markov chain

MCMC for Posterior Mean

- Standard Bayesian estimator is posterior mean

$$\hat{\theta} = \int_{\Theta} \theta \, p(\theta|y) \, d\theta$$

  - Posterior mean minimizes expected square error

- Estimate is a conditional expectation

$$\hat{\theta} = \mathbb{E}[\theta|y]$$

- Compute by averaging

$$\hat{\theta} \approx \frac{1}{M} \sum_{m=1}^{M} \theta$$

MCMC for Posterior Variance

- Posterior variance works the same way,

$$\mathbb{E}[(\theta - \mathbb{E}[\theta \mid y])^2 \mid y] \quad = \quad \mathbb{E}[(\theta - \hat{\theta})^2]$$

$$\approx \quad \frac{1}{M} \sum_{m=1}^{M} (\theta^{(m)} - \hat{\theta})^2$$

MCMC for Event Probability

- Event probabilities are also expectations, e.g.,

$$\Pr[\theta_1 > \theta_2] \;=\; \mathbb{E}[\mathsf{I}[\theta_1 > \theta_2]] \;=\; \int_{\Theta} \mathsf{I}[\theta_1 > \theta_2]\, p(\theta|y) d\theta.$$

- Estimation via MCMC just another plug-in:

$$\Pr[\theta_1 > \theta_2] \approx \frac{1}{M} \sum_{m=1}^{M} \mathsf{I}[\theta_1^{(m)} > \theta_2^{(m)}]$$

- Again, can be made as accurate as necessary

MCMC for Quantiles (incl. median)

- These are not expectations, but still plug in
- Alternative Bayesian estimator is posterior median
  - Posterior median minimizes expected absolute error
- Estimate as median draw of $\theta^{(1)}, \ldots, \theta^{(M)}$
  - just sort and take halfway value
  - e.g., Stan shows 50% point (or other quantiles)
- Other quantiles including interval bounds similar
  - estimate with quantile of draws
  - estimation error goes up in tail (based on fewer draws)

MCMC ALGORITHMS

Random-Walk Metropolis

- Draw random initial parameter vector $\theta^{(1)}$ (in support)
- For $m \in 2{:}M$
  - Sample proposal from a (symmetric) jumping distribution, e.g.,

$$\theta^* \sim \text{MultiNormal}(\theta^{(m-1)}, \sigma\mathbf{I})$$

    where $\mathbf{I}$ is the identity matrix

  - Draw $u^{(m)} \sim \text{Uniform}(0, 1)$ and set

$$\theta^{(m)} = \begin{cases} \theta^* & \text{if } u^{(m)} < \dfrac{p(\theta^*|y)}{p(\theta^{(m)}|y)} \\ \theta^{(m-1)} & \text{otherwise} \end{cases}$$

Metropolis and Normalization

- Metropolis only uses posterior in a ratio:

$$\frac{p(\theta^* \mid y)}{p(\theta^{(m)} \mid y)}$$

- This **allows** the use of **unnormalized densities**

- Recall Bayes's rule:

$$p(\theta|y) \propto p(y|\theta)\, p(\theta)$$

- Thus we only need to evaluate sampling (likelihood) and prior
  - i.e., no need to compute normalizing integral for $p(y)$,

$$\int_\Theta p(y|\theta)\, p(\theta) d\theta$$

Metropolis-Hastings

- Generalizes Metropolis to asymmetric proposals
- Acceptance ratio is

$$\frac{J(\theta^{(m)}|\theta^*) \times p(\theta^*|y)}{J(\theta^*|\theta^{(m-1)}) \times p(\theta^{(m)}|y)}$$

  where J is the (potentially asymmetric) proposal density

- i.e.,

$$\frac{\text{probabilty of being at } \theta^* \text{ and jumping to } \theta^{(m-1)}}{\text{probability of being at } \theta^{(m-1)} \text{ and jumping to } \theta^*}$$

Metropolis-Hastings (cont.)

- General form ensures equilibrium by maintaining *detailed balance*
- Like Metropolis, only requires ratios
- Many algorithms involve a Metropolis-Hastings "correction"
    - Including vanilla HMC and RHMC and ensemble samplers

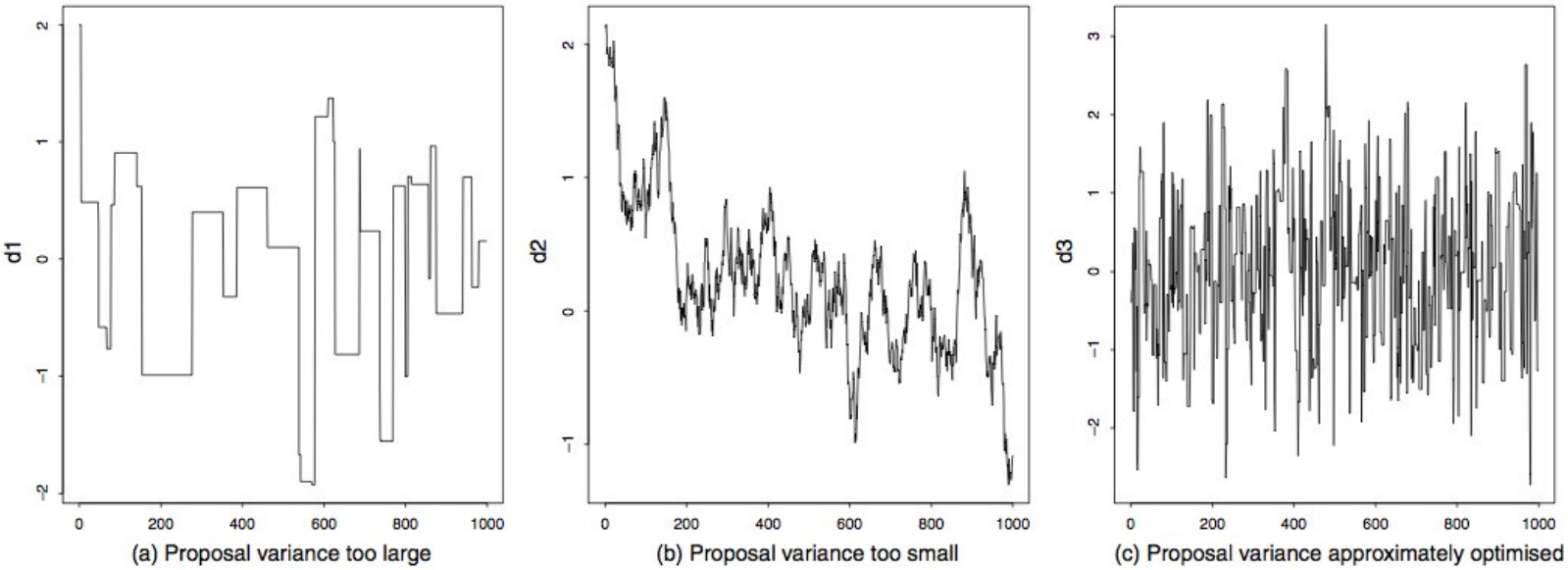
Detailed Balance & Reversibility

- Definition is measure theoretic, but applies to densities
  - just like Bayes's rule
- Assume Markov chain has stationary density $p(a)$
- Suppose $\pi(a|b)$ is density of transitioning from b to a
  - use of π to indicates different measure on Θ than p
- Detailed balance is a reversibility equilibrium condition

$$p(a)\,\pi(b|a) \;=\; p(b)\,\pi(a|b)$$

Optimal Proposal Scale?

- Proposal scale σ is a free; too low or high is inefficient



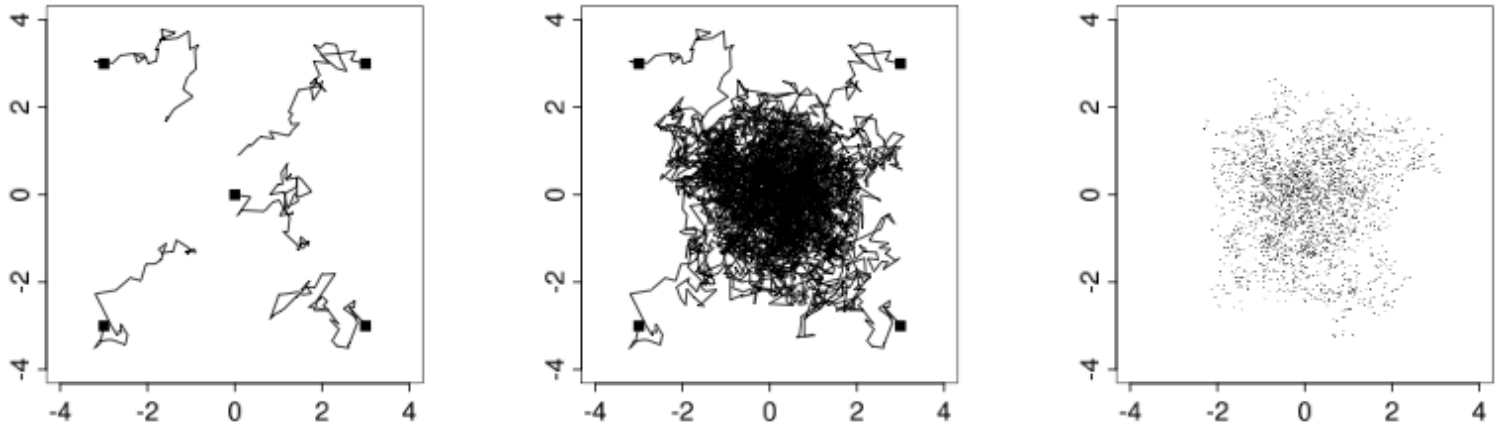
- *Traceplots* show parameter value on y axis, iterations on x
- Empirical tuning problem; theoretical optima exist for some cases

Roberts and Rosenthal (2001) Optimal Scaling for Various Metropolis-Hastings Algorithms. *Statistical Science*.

Convergence

- Imagine releasing a hive of bees in a sealed house
  - they disperse, but eventually reach equilibrium where the same number of bees leave a room as enter it (on average)
- May take many iterations for Markov chain to reach equilibrium

Convergence: Example



- Four chains with different starting points
  - *Left*: 50 iterations
  - *Center*: 1000 iterations
  - *Right*: Draws from second half of each chain

Gelman et al., *Bayesian Data Analysis*

Potential Scale Reduction ($\hat{R}$)

- Gelman & Rubin recommend M chains of N draws with **diffuse initializations**
- Measure that each chain has same posterior mean and variance
- If not, may be stuck in multiple modes or just not converged yet
- Define statistic $\hat{R}$ of chains s.t. **at convergence**, $\hat{R} \rightarrow 1$
  - $\hat{R} >> 1$ implies non-convergence
  - $\hat{R} \approx 1$ **does not guarantee convergence**
  - Only measures marginals

Split $\hat{R}$

- Vanilla $\hat{R}$ may not diagnose non-stationarity
  - e.g., a sequence of chains with an increasing parameter
- **Split $\hat{R}$**: Stan splits each chain into first and second half
  - start with M Markov chains of N draws each
  - split each in half to creates $2M$ chains of $N/2$ draws
  - then apply $\hat{R}$ to the $2M$ chains

Calculating $\hat{R}$ Statistic: Between

- **Between-sample variance** estimate

$$B = \frac{N}{M-1} \sum_{m=1}^{M} (\bar{\theta}_m^{(\bullet)} - \bar{\theta}_\bullet^{(\bullet)})^2,$$

where

$$\bar{\theta}_m^{(\bullet)} = \frac{1}{N} \sum_{n=1}^{N} \theta_m^{(n)} \quad \text{and} \quad \bar{\theta}_\bullet^{(\bullet)} = \frac{1}{M} \sum_{m=1}^{M} \bar{\theta}_m^{(\bullet)}.$$

Calculating $\hat{R}$ (cont.)

- **Within-sample variance** estimate:

$$W = \frac{1}{M} \sum_{m=1}^{M} s_m^2,$$

where

$$s_m^2 = \frac{1}{N-1} \sum_{n=1}^{N} (\theta_m^{(n)} - \bar{\theta}_m^{(\bullet)})^2.$$

Calculating $\hat{R}$ Statistic (cont.)

- **Variance** estimate:

$$\widehat{\text{var}}^{+}(\theta|y) = \frac{N-1}{N} W + \frac{1}{N} B.$$

recall that W is within-chain variance and B between-chain

- **Potential scale reduction** statistic ("R hat")

$$\hat{R} = \sqrt{\frac{\widehat{\text{var}}^{+}(\theta|y)}{W}}.$$

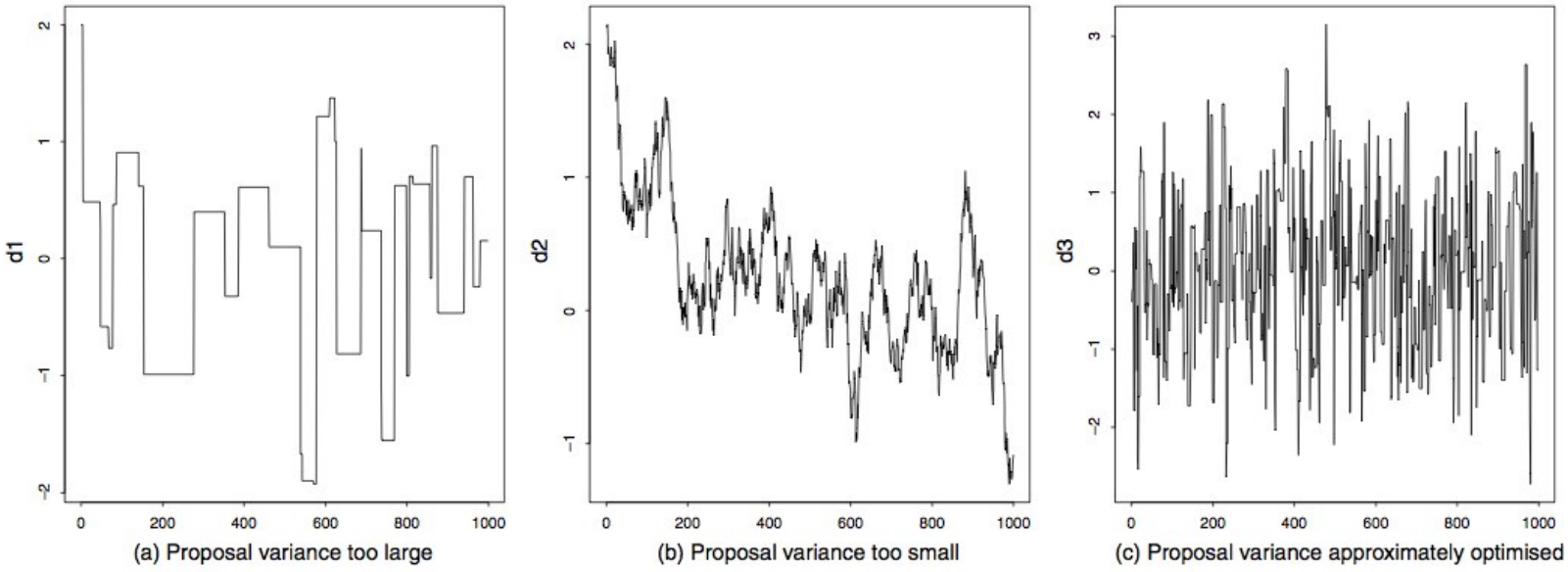
Correlations in Posterior Draws

- Markov chains typically display autocorrelation in the series of draws $\theta^{(1)}, \ldots, \theta^{(m)}$
- Without i.i.d. draws, central limit theorem *does not apply*
- Effective sample size N_{eff} divides out autocorrelation
- N_{eff} must be estimated from sample
  - Fast Fourier transform computes correlations at all lags
- Estimation accuracy proportional to

$$\frac{1}{\sqrt{N_{\text{eff}}}}$$

Reducing Posterior Correlation

- Tuning algorithm parameters to ensure good mixing
- Recall Metropolis traceplots of Roberts and Rosenthal:



- Good jump scale σ produces good mixing and high N_{eff}

Effective Sample Size

- Autocorrelation at lag t is correlation between subseqs
  - $(\theta^{(1)}, \ldots, \theta^{(N-t)})$ and $(\theta^{(1+t)}, \ldots, \theta^{(N)})$
- Suppose chain has density $p(\theta)$ with
  - $\mathbb{E}[\theta] = \mu$ and $\mathrm{Var}[\theta] = \sigma^2$
- Autocorrelation ρ_t at lag $t \geq 0$:

$$\rho_t = \frac{1}{\sigma^2} \int_{\Theta} (\theta^{(n)} - \mu)(\theta^{(n+t)} - \mu)\, p(\theta)\, d\theta$$

- Because $p(\theta^{(n)}) = p(\theta^{(n+t)}) = p(\theta)$ at convergence,

$$\rho_t = \frac{1}{\sigma^2} \int_{\Theta} \theta^{(n)}\, \theta^{(n+t)}\, p(\theta)\, d\theta$$

Estimating Autocorrelations

- Effective sample size is defined by

$$N_{\text{eff}} = \frac{N}{\sum_{t=-\infty}^{\infty} \rho_t} = \frac{N}{1 + 2\sum_{t=1}^{\infty} \rho_t}$$

- Estimate in terms of variograms at lag t (calc with FFT),

$$V_t = \frac{1}{M} \sum_{m=1}^{M} \left(\frac{1}{N_m - t} \sum_{n=t+1}^{N_m} \left(\theta_m^{(n)} - \theta_m^{(n-t)} \right)^2 \right)$$

- Adjust autocorrelation at lag t using cross-chain variance as

$$\hat{\rho}_t = 1 - \frac{V_t}{2\,\widehat{\text{var}}^{+}}$$

- If not converged, $\widehat{\text{var}}^{+}$ overestimates variance

Estimating N_{eff}

- Let T' be first lag s.t. $\rho_{T'+1} < 0$,
- Estimate autocorrelation by

$$\hat{N}_{\text{eff}} = \frac{MN}{1 + \sum_{t=1}^{T'} \hat{\rho}_t}.$$

- NUTS avoids negative autocorrelations, so first negative autocorrelation estimate is reasonable

- See Charles Geyer (2013) Introduction to MCMC. In *Handbook of MCMC*. (free online at `http://www.mcmchandbook.net/index.html`)

Gibbs Sampling

- Draw random initial parameter vector $\theta^{(1)}$ (in support)
- For $m \in 2{:}M$
  - For $n \in 1{:}N$:
    - draw $\theta_n^{(m)}$ according to conditional

$$p(\theta_n | \theta_1^{(m)}, \ldots, \theta_{n-1}^{(m)}, \theta_{n+1}^{(m-1)}, \ldots, \theta_N^{(m-1)}, y).$$

- e.g, with $\theta = (\theta_1, \theta_2, \theta_3)$:
  - draw $\theta_1^{(m)}$ according to $p(\theta_1 | \theta_2^{(m-1)}, \theta_3^{(m-1)}, y)$
  - draw $\theta_2^{(m)}$ according to $p(\theta_2 | \theta_1^{(m)}, \theta_3^{(m-1)}, y)$
  - draw $\theta_3^{(m)}$ according to $p(\theta_3 | \theta_1^{(m)}, \theta_2^{(m)}, y)$

Generalized Gibbs

- "Proper" Gibbs requires conditional Monte Carlo draws
  - typically works only for conjugate priors
- In general case, may need to use less efficient conditional draws
  - Slice sampling is a popular general technique that works for discrete or continuous θ_n (JAGS)
  - Adaptive rejection sampling is another alternative (BUGS)
  - Very difficult in more than one or two dimensions

Sampling Efficiency

- We care only about N_{eff} per second
- Decompose into
    1. Iterations per second
    2. Effective sample size per iteration
- Gibbs and Metropolis have high iterations per second (especially Metropolis)
- But they have low effective sample size per iteration (especially Metropolis)
- Both are particular weak when there is high correlation among the parameters in the posterior

Sample: HMC/NUTS Sampling

- Fix stepsize and and mass matrix
- For sampling iterations
  - generate random kinetic energy
  - simulate Hamiltonian flow
  - apply Metropolis accept/reject (HMC) or slice (NUTS)

Euclidean Hamiltonian

- **Phase space**: q position (parameters); p momentum
- **Posterior density**: $\pi(q)$
- **Mass matrix**: M
- **Potential energy**: $V(q) = -\log \pi(q)$
- **Kinetic energy**: $T(p) = \frac{1}{2} p^\top M^{-1} p$
- **Hamiltonian**: $H(p, q) = V(q) + T(p)$
- **Diff eqs**:

$$\frac{dq}{dt} = +\frac{\partial H}{\partial p} \qquad \frac{dp}{dt} = -\frac{\partial H}{\partial q}$$

Leapfrog Integrator Steps

- Solves Hamilton's equations by **simulating dynamics**
  (symplectic [volume preserving]; ϵ^3 error per step, ϵ^2 total error)
- Given: **step size** ϵ, **mass matrix** M, **parameters** q
- **Initialize kinetic** energy, $p \sim \text{Normal}(0, \mathbf{I})$
- **Repeat** for L leapfrog steps:

$$p \leftarrow p - \frac{\epsilon}{2} \frac{\partial V(q)}{\partial q} \qquad \text{[half step in momentum]}$$

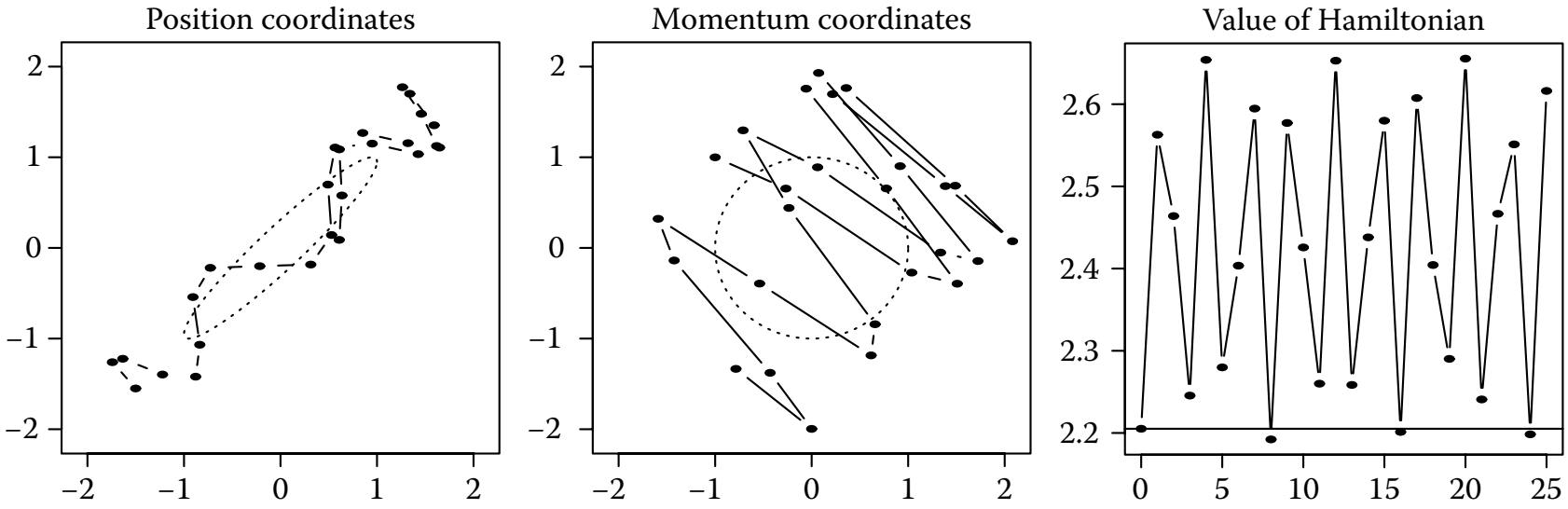
$$q \leftarrow q + \epsilon M^{-1} p \qquad \text{[full step in position]}$$

$$p \leftarrow p - \frac{\epsilon}{2} \frac{\partial V(q)}{\partial q} \qquad \text{[half step in momentum]}$$

Sample: Hamiltonian Flow

- Generate random **kinetic energy**
  - random Normal$(0, 1)$ in each parameter
- Use negative log posterior as **potential energy**
- Hamiltonian is kinetic plus potential energy
- **Leapfrog Integration**: for *fixed* stepsize (time discretization), number of steps (total time), and mass matrix,
  - update momentum half-step based on potential (gradient)
  - update position full step based on momentum
  - update momentum half-step based on potential
- Numerical solution of Hamilton's first-order version of Newton's second-order diff-eqs of motion (force $=$ mass $\times$ acceleration)

Sample: Leapfrog Example



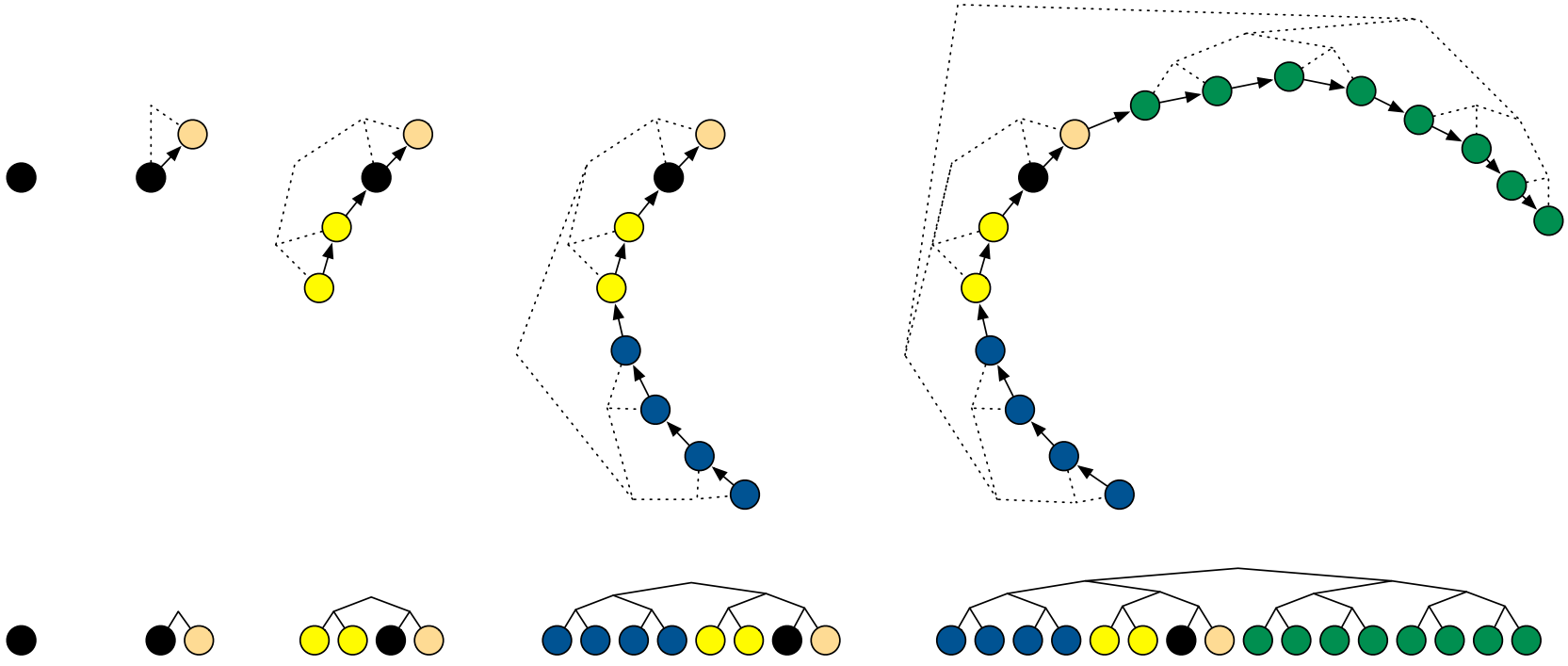
- Trajectory of 25 leapfrog steps for correlated 2D normal (ellipses at 1 sd from mean), stepsize of 0.25, initial state of $(-1, 1)$, and initial momentum of $(-1.5, -1.55)$.

Radford Neal (2013) MCMC using Hamiltonian Dynamics. In *Handbook of MCMC*. (free online at `http://www.mcmchandbook.net/index.html`)

Sample: No-U-Turn Sampler (NUTS)

- Adapts Hamiltonian simulation time
  - goal to maximize mixing, maintaining detailed balance
  - too short devolves to random walk
  - too long does extra work (i.e., orbits)
- For exponentially increasing number of steps up to max
  - Randomly choose to extend forward or backward in time
  - Move forward or backward in time number of steps
    - stop if any subtree (size 2, 4, 8, ...) makes U-turn
    - remove all current steps if subtree U-turns (not ends)
- Randomly select param with density above slice (or reject)

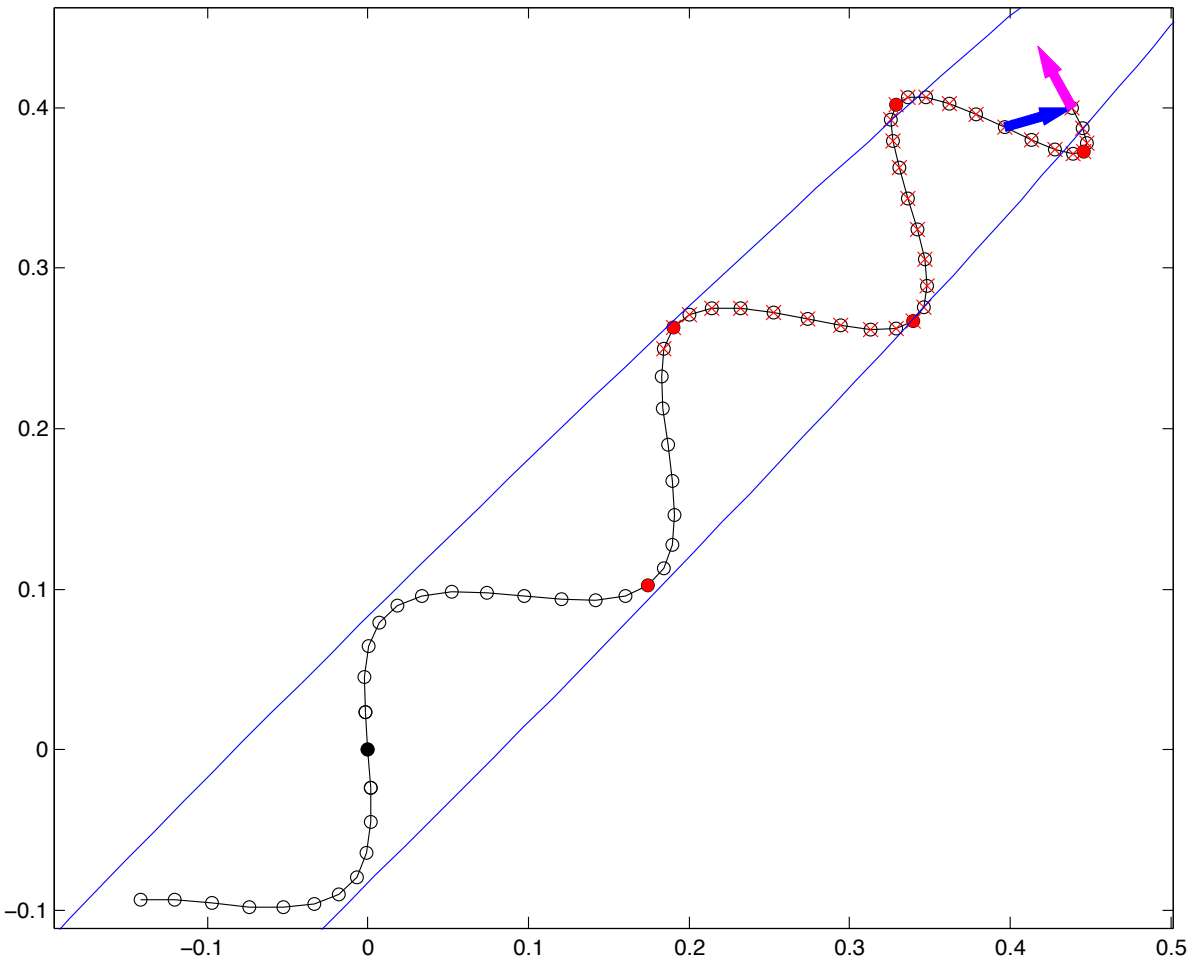
Sample: NUTS Binary Tree



- Example of repeated doubling building binary tree forward and backward in time until U-turn.

  Hoffman and Gelman. 2014. The No-U-Turn Sampler. *JMLR*. (free online at http://jmlr.org/papers/v15/hoffman14a.html)

Sample: NUTS U-Turn



- Example of trajectory from one iteration of NUTS.
- Blue ellipse is contour of 2D normal.
- Black circles are leapfrog steps.
- Solid red circles excluded below slice
- U-turn made with blue and magenta arrows
- Red crossed circles excluded for detailed balance

Sample: HMC/NUTS Warmup

- Estimate stepsize
  - too small requires too many leapfrog steps
  - too large induces numerical inaccuracy
  - need to balance
- Estimate mass matrix
  - Diagonal accounts for parameter scales
  - Dense optionally accounts for rotation

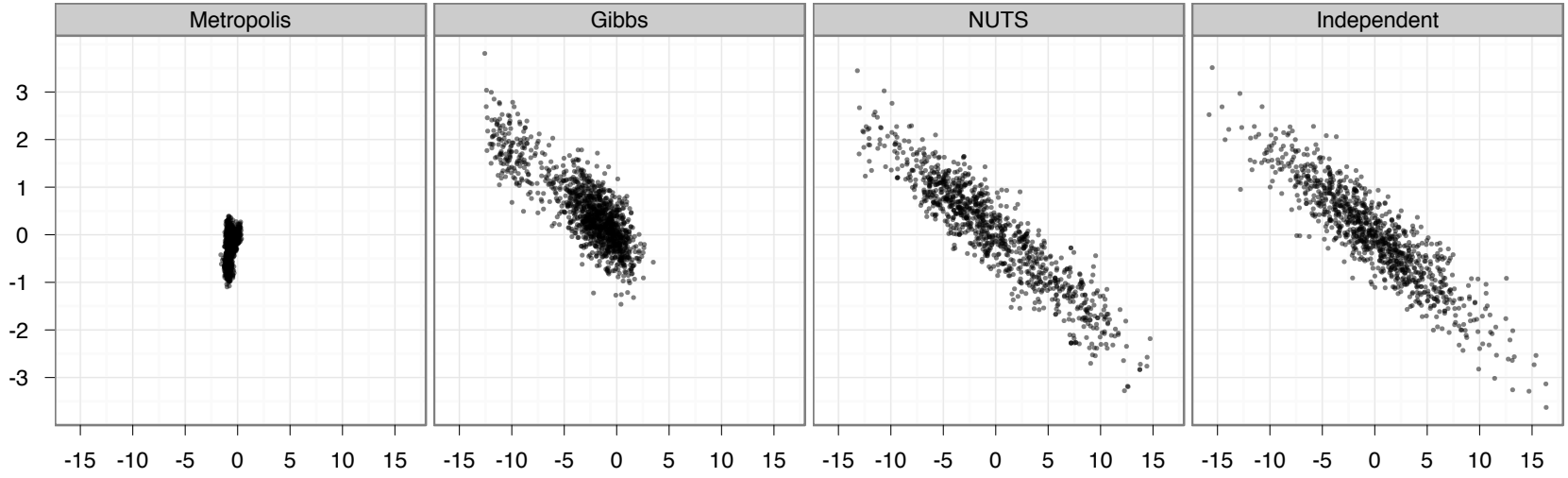
Sample: Warmup (cont.)

- Initialize unconstrained parameters as for optimization
- For exponentially increasing block sizes
  - for each iteration in block
    - generate random kinetic energy
    - simulate Hamiltonian flow (HMC fixed time, NUTS adapts)
    - choose next state (Metroplis for HMC, slice for NUTS)
  - update regularized point estimate of mass matrix
    - use parameter draws from current block
    - shrink diagonal toward unit; dense toward diagonal
  - tune stepsize (line search) for target acceptance rate

Sample: HMC/NUTS Sampling

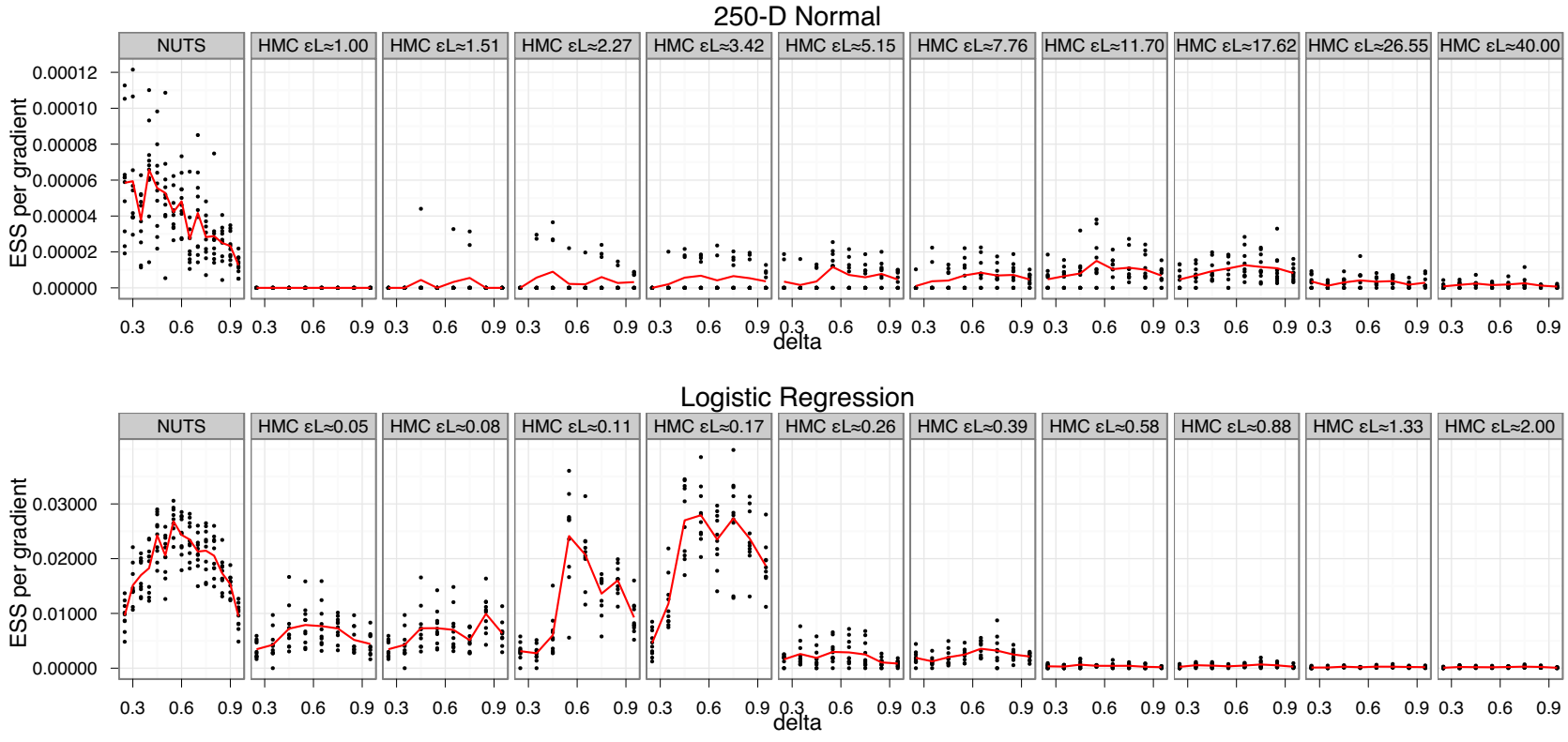
- Fix stepsize and and mass matrix
- For sampling iterations
  - generate random kinetic energy
  - simulate Hamiltonian flow
  - apply Metropolis accept/reject (HMC) or slice (NUTS)

NUTS vs. Gibbs and Metropolis



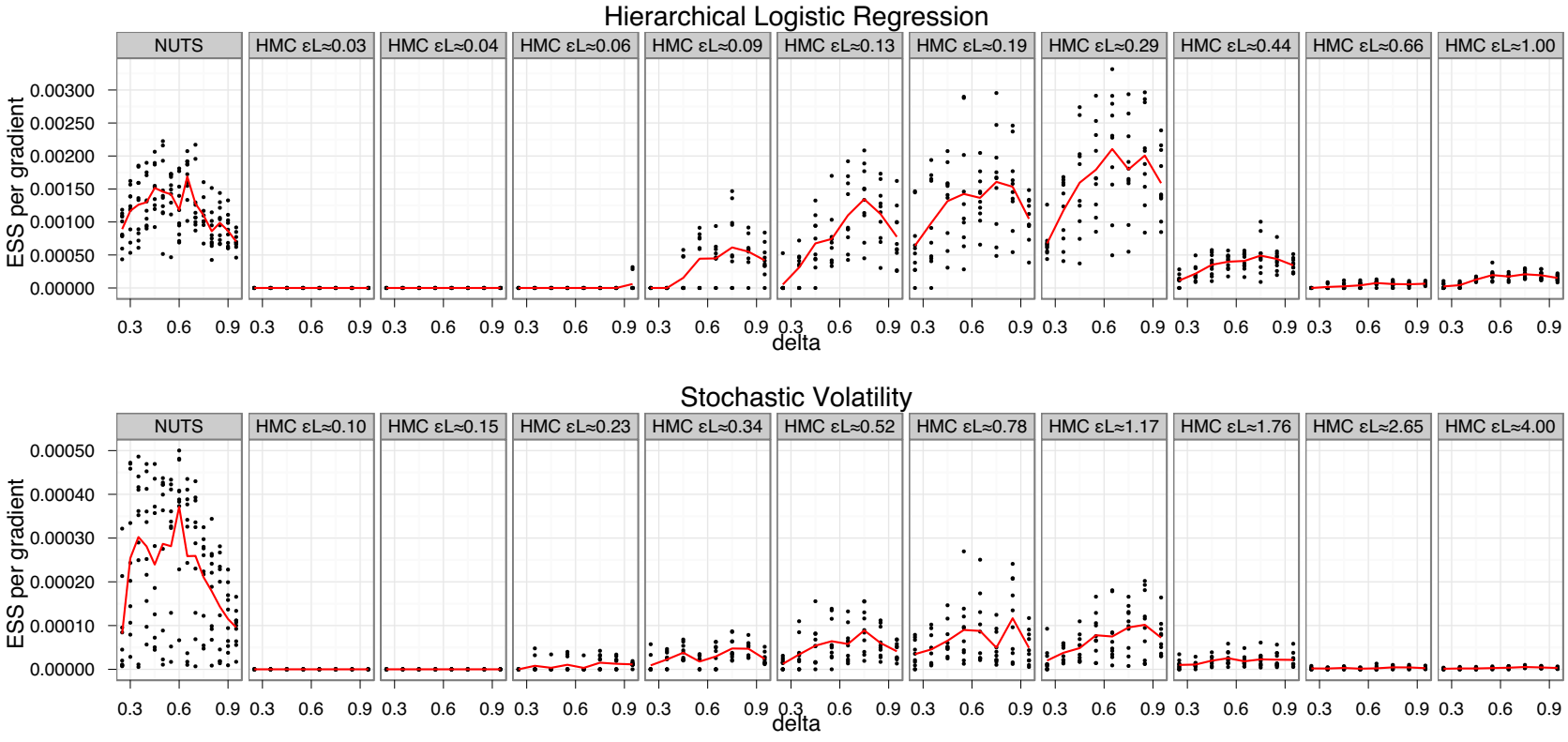
- Two dimensions of highly correlated 250-dim normal
- **1,000,000 draws** from Metropolis and Gibbs (thin to 1000)
- **1000 draws** from NUTS; 1000 independent draws

NUTS vs. Basic HMC



- 250-D normal and logistic regression models
- Vertical axis is effective sample size per sample (bigger better)
- Left) NUTS; Right) HMC with increasing $t = \epsilon L$

NUTS vs. Basic HMC II



- Hierarchical logistic regression and stochastic volatility
- Simulation time t is $\epsilon\ L$, step size (ϵ) times number of steps (L)
- NUTS can beat optimally tuned HMC (latter very expensive)

WHAT DO THESE ALGORITHMS NEED?

- Gibbs Sampling
  - Conditional distributions.
    To do this automatically, conjugacy is key!
    Limits the space of models.
- Metropolis Hastings
  - Needs a function that's proportional to the joint distribution!
    Awesome. We can do that.
- HMC / NUTS
  - Needs a function that's proportional to the joint distribution.
  - Needs gradients with respect to its parameters!
    Boo. This is hard. Limits the models to continuous parameters.
    Yay. Stan computes gradients using automatic differentiation!

3 STEPS OF BAYESIAN DATA ANALYSIS

3 STEPS OF BAYESIAN DATA ANALYSIS (FROM BDA3)

1. Set up a full probability model.
   Joint probability distribution of all observables and unobservables.

2. Condition on observed data.
   Calculate and interpret the posterior distribution.

3. Evaluate fit of the model and the implications.
   Are the conclusions reasonable?

3 STEPS OF BAYESIAN DATA ANALYSIS (FROM BDA3)

1. Set up a full probability model.
   Write a Stan program.

2. Condition on observed data.
   Run the Stan program using RStan.
   Check to see if everything went ok in RStan and ShinyStan.

3. Evaluate fit of the model and the implications.
   Evaluate in RStan and ShinyStan.
   Question assumptions.
   Go back to 1.

BAYESIAN INFERENCE IN STAN

WHAT IS STAN?

http://mc-stan.org

WHAT IS STAN?

STAN

1. Language

2. Inference algorithms

3. Interfaces

STAN LANGUAGE

- Statistical model specification language
  - specify: data, parameters, joint probability distribution
- Domain specific language

  not C++, R, Python, BUGS, JAGS...
- Imperative language

  c.f. declarative (BUGS, JAGS), object oriented (Figaro)

INFERENCE ALGORITHMS

- Bayesian inference; Markov Chain Monte Carlo (MCMC)
  - **No-U-Turn Sampler (NUTS)**
  - Hamiltonian Monte Carlo (HMC)
- Approximate Bayesian inference
  - Automatic Differentiation Variational Inference (ADVI)
- Optimization
  - Limited memory Brayden-Fletcher-Goldfarb-Shanno (L-BFGS)

PLATFORMS AND INTERFACES

- Runs in Windows, Mac OS X, and Linux
- C++ API: portable, standards compliant (C++03)
- Interfaces
  - RStan: R interface (Rcpp in memory)
  - CmdStan: command-line or shell access (direct executable)
  - PyStan: Python interface (Cython in memory)
  - MatlabStan: Matlab interface (external process)
  - Stan.jl: Julia interface (external process)
  - StataStan: Stata interface (external process)

WHO'S USING STAN?

- 1900 user group registrations
  1900 downloads of v2.11 manual (07/16)
  15k downloads of v2.9 manual (4/15)
  450 Google Scholar citation (100+ fitting)

- Biological sciences: clinical drug trials, entomology, ophthalmology, neurology, genomics, agriculture, botany, fisheries, cancer biology, epidemiology, population ecology, neurology

- Physical sciences: astrophysics, molecular biology, oceanography, climatology

- Social sciences: population dynamics, psycholinguistics, social networks, political science

- Other: materials engineering, finance, actuarial, sports, public health, recommender systems, educational testing

DOCUMENTATION

- *Stan User's Guide and Reference Manual*
  - 500+ pages
  - Example models, modeling and programming advice
  - Intro to Bayesian and frequentist statistics
  - Complete language specification and execution guide
  - Description of algorithms (NUTS, R-hat, n_eff)
  - Guide to built-in distributions and functions

- Installation and getting started manuals by interface

BOOKS AND MODEL SETS

- **Model Sets** translated to Stan
  https://github.com/stan-dev/example-models
  - BUGS and JAGS examples (most of all 3 volumes)
  - Gelman and Hill (2009)
    Data Analysis Using Regression and Multilevel/Hierarchical Models
  - Wagenmakers and Lee (2014)
    Bayesian Cognitive Modeling
  - Kéry and Schaub (2012)
    Bayesian Population Analysis using WinBUGS --- A Hierarchical Perspective
- **Books** with Sections on Stan
  - Gelman et al. (2013) *Bayesian Data Analysis*. 3rd Edition.
  - Krusche (2014) *Doing Bayesian Data Analysis*.
  - Korner-Nievergelt et al. (2015) *Bayesian Data Analysis in Ecology Using Linear Models with R, BUGS, and Stan*

OPEN SOURCE

- Stan is freedom-respecting, open-source software
  - Math library, Stan library, CmdStan:
    new BSD license
  - RStan, PyStan: GPL
- Hosted on GitHub
  http://github.com/stan-dev/
- Over 40 contributors since 2011
- Active development

HOW STAN WORKS

STEPS

1. Stan program read into memory

```
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  y ~ bernoulli(theta);
}
```

STEPS

1. Stan program read into memory

2. Source-to-source transformation to C++

```
// Code generated by Stan version 2.9

#include <stan/model/model_header.hpp>

namespace bernoulli_model_namespace {

using std::istream;
using std::string;
using std::stringstream;
using std::vector;
using stan::io::dump;
using stan::math::lgamma;
using stan::model::prob_grad;
using namespace stan::math;

typedef Eigen::Matrix<double,Eigen::Dynamic,1> vector_d;
typedef Eigen::Matrix<double,1,Eigen::Dynamic> row_vector_d;
typedef Eigen::Matrix<double,Eigen::Dynamic,Eigen::Dynamic> matrix_d;

static int current_statement_begin__;

class bernoulli_model : public prob_grad {
private:
    int N;
    vector<int> y;
public:
    bernoulli_model(stan::io::var_context& context__,
        std::ostream* pstream__ = 0)
        : prob_grad(0) {
        current_statement_begin__ = -1;

        static const char* function__ = "bernoulli_model_namespace::bernoulli_model";
        (void) function__; // dummy call to supress warning
        size_t pos__;
        (void) pos__; // dummy call to supress warning
        std::vector<int> vals_i__;
        std::vector<double> vals_r__;
        context__.validate_dims("data initialization", "N", "int", context__.to_vec());
        N = int(0);
        vals_i__ = context__.vals_i("N");
        pos__ = 0;
        N = vals_i__[pos__++];
        context__.validate_dims("data initialization", "y", "int", context__.to_vec(N));
        validate_non_negative_index("y", "N", N);
        y = std::vector<int>(N,int(0));
        vals_i__ = context__.vals_i("y");
        pos__ = 0;
        size_t y_limit_0__ = N;
        for (size_t i_0__ = 0; i_0__ < y_limit_0__; ++i_0__) {
            y[i_0__] = vals_i__[pos__++];
        }

        // validate data
        check_greater_or_equal(function__,"N",N,0);
        for (int k0__ = 0; k0__ < N; ++k0__) {
            check_greater_or_equal(function__,"y[k0__]",y[k0__],0);
            check_less_or_equal(function__,"y[k0__]",y[k0__],1);
        }

        double DUMMY_VAR__(std::numeric_limits<double>::quiet_NaN());
        (void) DUMMY_VAR__;  // suppress unused var warning


        // initialize transformed variables to avoid seg fault on val access

        try {
        } catch (const std::exception& e) {
            stan::lang::rethrow_located(e,current_statement_begin__);
            // Next line prevents compiler griping about no return
            throw std::runtime_error("*** IF YOU SEE THIS, PLEASE REPORT A BUG ***");
        }

        // validate transformed data

        // set parameter ranges
        num_params_r__ = 0U;
        param_ranges_i__.clear();
        ++num_params_r__;
    }

    ~bernoulli_model() { }


    void transform_inits(const stan::io::var_context& context__,
                         std::vector<int>& params_i__,
                         std::vector<double>& params_r__,
                         std::ostream* pstream__) const {
        stan::io::writer<double> writer__(params_r__,params_i__);
        size_t pos__;
        (void) pos__; // dummy call to supress warning
        std::vector<double> vals_r__;
        std::vector<int> vals_i__;

        if (!(context__.contains_r("theta")))
            throw std::runtime_error("variable theta missing");
        vals_r__ = context__.vals_r("theta");
        pos__ = 0U;
        context__.validate_dims("initialization", "theta", "double", context__.to_vec());
```

STEPS

1. Stan program read into memory
2. Source-to-source transformation to C++
3. C++ compiled and linked
   (takes a while)

STEPS

1. Stan program read into memory

2. Source-to-source transformation to C++

3. C++ compiled and linked
   (takes a while)

4. Run Stan program
   run inference algorithm on the Stan program provided
   (data already loaded in memory)

WHY USE STAN?

- vs. BUGS and JAGS
  - Time to converge and per effective sample size:
    0.5 – ∞ times faster
  - Memory usage: 1 – 10%
  - Language features
- vs. writing own samplers
  - Iterating over models is easy:
    no writing gradients / Gibbs sampler
  - We're still working on new inference algorithms

WHERE TO GET HELP?

- Stan Home Page
  http://mc-stan.org

- Stan Users Google Group
  https://groups.google.com/d/forum/stan-users
  (We'll eventually migrate to discourse.mc-stan.org)

- Stan User Guide and Reference Manual
  http://mc-stan.org/documentation/

  https://github.com/stan-dev/stan/releases/download/v2.11.0/stan-reference-2.11.0.pdf

RECAP

- Although Stan's the new kid on the block, you're in good company
- Active development
- No Stan book yet, but examples are out there

Coming up next:

- Stan language
- RStan usage

STAN LANGUAGE

GOALS

- Write statistical models in Stan language
  - learn language
  - write Stan programs
- Diagnose models

RECAP

STATISTICAL INFERENCE

Want

posterior distribution of parameters given data

$$p(\theta \mid x)$$

Given

joint model

$$p(\theta, x)$$

data

$$x$$

parameters

$$\theta$$

WHY MCMC?

$$
\begin{aligned}
p(\theta \mid x) &= \frac{p(\theta, x)}{p(x)} \\
&= \frac{p(\theta, x)}{\int p(\theta, x) d\theta}
\end{aligned}
$$

WHY MCMC?

$$p(\theta \mid x) = \frac{p(\theta, x)}{p(x)}$$

$$= \frac{p(\theta, x)}{\int p(\theta, x) d\theta}$$

WHY MCMC?

$$p(\theta \mid x) \propto \boldsymbol{p(\theta, x)}$$

- MCMC generates draws from the posterior distribution
- We write the joint distribution, Stan does the MCMC
- Stan estimates expectations

$$\mathbb{E}[f(x, \theta)] = \int f(x, \theta) \times p(\theta \mid x) dx$$

IN STAN, WE DEFINE

- Joint model of data and parameters: $\log p(\theta, x)$
  - Define data x
  - Define parameters θ

STAN LANGUAGE

- Specify (differentiable) statistical models
- imperative language
- statically typed

STAN IS A PROGRAMMING LANGUAGE

- Higher level language for specifying **differentiable** log density functions
- Turing-complete
- full conditionals and loops, functions (including recursion)
- strong, statically typed
- reassignable local variables and scoping

- Not a graphical model specification language
- “black-box” inference is built on top of the language

SIMPLE EXAMPLE

COIN FLIPS (EARLIER EXAMPLE)

- Look at the flips: mean, count
- Generative model: assumptions?

```
data {
  int N;
  int x[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  x ~ bernoulli(theta);
}
```

COIN FLIP MODEL

- Create a file "bernoulli.stan"

```
data {
  int N;
  int x[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  x ~ bernoulli(theta);
}
```

- Run:

```
R:      fit <- stan("bernoulli.stan", data=c("N", "x"))
```

NOT THE ONLY WAY TO WRITE THIS MODEL

```
data {
  int N;
  int x[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  for (n in 1:N)
    x[n] ~ bernoulli(theta);
}
```

NOT THE ONLY WAY TO WRITE THIS MODEL

```
data {
  int N;
  int x[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  for (n in 1:N)
    target += bernoulli_lpmf(x[n] | theta));
}
```

THINGS WE CAN DO

- With the fit:
  - R: summary, print, traceplot, extract
    - shinystan: launch_shinystan(fit)
  - Python: print, extract, plot
- Change the model!

STEPS

1. Write Stan program (.stan file)
2. Translate Stan program to C++ (Stan interface)
3. Compile an executable (Stan interface & C++ compiler)
   - CmdStan – command line program
   - RStan & PyStan – exposed through functions
4. Run MCMC (Stan interface)
5. Posterior analysis (Stan interface)

STAN LANGUAGE

GENERAL PROPERTIES OF STAN LANGUAGE

- Whitespace does not matter
- Comments
  - `//` or `#`
  - `/* … */`
- semicolon (;)
- Variables are typed and scoped
- Compile-time vs run-time errors

BLOCKS: STRUCTURE OF A STAN PROGRAM

```
functions

data

transformed data

parameters

transformed parameters

model

generated quantities
```

- Blocks start and end with braces (`{ }`)
- `model` block is required
- Variables declared in each block have scope over all subsequent statements

STAN TYPES

VARIABLE DECLARATION

- Each variable has a type (static type)
- Only values of that type can be assigned to the variable (strongly typed)
- Declaration of variables happen at the top of a block (including local blocks)

- Start by learning about the types in the context of the data block

SCALAR DATA TYPES

`real`

- scalar
- continuous

```
data {
  real y;
}
```

`int`

- scalar
- integer
- can't be used in `parameters` or `transformed parameters` blocks

```
data {
  int n;
}
```

EXAMPLE: SCALAR DECLARATION

```
data {
  real y;
  int n;
}
parameters {
  real theta;
}
model {
  theta ~ normal(0, 1);
}
```

1. Generate fake data and run

```
dat <- list(y = 10.1, n = 3)
fit <- stan("example.stan",
            data=dat)
```

2. Run with different data (no recompiling)

```
fit <- stan(fit = fit,
            data=dat)
```

3. Try with:
   - missing semicolon
   - missing data
   - non-int data for n

CONSTRAINING SCALAR VARIABLES

- Validates data is within range
- lower bound, upper bound, both
- inclusive
- can use infinite constraints:
  `positive_infinity()`
  `negative_infinity()`
- bounds can be expressions

```
data {
  int<lower=1> m;
  int<lower=0,upper=1> n;
  real<lower=0> x;
  real<upper=0> y;
  real<lower=-1,upper=1> rho;
}
```

EXAMPLE: CONSTRAINED SCALAR

```
data {
  int<lower=1> m;
  int<lower=0,upper=1> n;
  real<lower=0> x;
  real<upper=0> y;
  real<lower=x,upper=1> z;
}
parameters {
  real theta;
}
model {
  theta ~ normal(0, 1);
}
```

1. Generate fake data and run

```
dat <- list(m = 1, n = 0,
            x = 0.1, y = -0.1,
            rho = 0.5, z = 0.2)
fit <- stan("example.stan",
            data=dat)
```

2. Try with

- x greater than 1
- real variables on boundaries

VECTOR DATA TYPES

- Contains real values
- Indexing starts at 1
- Declared with size

- `vector[3] a;`
  column vector
- `row_vector[4] b;`
  row vector
- `simplex[5] c;`
  vector, sums to 1, non-negative entries
- `unit_vector[5] d;`
  vector with norm of 1
- `ordered[6] e;`
  vector in ascending order
- `positive_ordered[7] f;`
  positive, ordered vector

MATRIX DATA TYPES

- Contains real values
- Indexing starts at 1
- Declared with size

- ```
  matrix[3,4] A;
  3x4 matrix
  A[1] returns a 4-row_vector
  ```

- ```
  corr_matrix[3] Sigma;
  square, symmetric matrix
  positive definite
  entries between -1 and 1
  diagonal 1
  ```

- ```
  cholesky_factor_corr[K] L;
  represents Cholesky factor
    of correlation matrix
  K x K lower-triangular
  positive diagonal entries
  rows are length 1
  L L^T is a correlation matrix
  ```

- ```
  cov_matrix[3] Omega;
  symmetric, square, positive definite
  ```

- ```
  cholesky_factor_cov[M,N] L;
  cholesky_factor_cov[4] L; (square matrix)
  L L^T is a covariance matrix
  ```
  ```

PASSING DATA FROM RSTAN

- vector and row_vector are represented as vector in R
  `x <- c(1, 2, 3)`
- matrix is represented as a matrix in R

- Try passing data from RStan to vector types
  - ```
    vector[3] a;
    row_vector[4] b;
    simplex[5] c;
    unit_vector[5] d;
    ordered[6] e;
    positive_ordered[7] f;
    ```
- Matrix types
  - ```
    matrix[3,4] A;
    corr_matrix[3] Sigma;
    cholesky_factor_corr[K] L;
    cov_matrix[3] Omega;
    cholesky_factor_cov[M,N] L;
    cholesky_factor_cov[4] L;
    ```

CONSTRAINING VECTOR AND MATRIX TYPES

- Add upper and lower bounds to vectors and matrices

```
vector<lower=0,upper=1>[5] rhos;
row_vector<lower=0>[4] sigmas;
matrix<lower=-1, upper=1>[3,4] Sigma;
```

- Bounds can be expressions
- Can’t put bounds on constrained vector or matrix types
  e.g. unit_vector, simplex, ordered, …
- Try it, see messages when data does not respect the constraints

ARRAYS

- All types can be made arrays
- Arrays can be multi-dimensional
- Examples
  - `real a[5];`
  - `vector[5] b[3];`
  - `int N[2,3];`
  - `vector<lower=0>[5] c[L,M,N];`

RECAP: STAN TYPES

Scalar types

- `real`
- `int`

Vector types

- `vector, row_vector`
- `simplex, unit_vector`
- `ordered, positive_ordered`

Matrix types

- `matrix`
- `corr_matrix, cov_matrix`
- `cholesky_factor_corr, cholesky_factor_cov`

Bounds

- `lower, upper, both`

Arrays

DATA

THE DATA BLOCK

- **Declare** data only
- Within the block, can’t do anything else
- Data read in from Stan interface in order declared
- All data declared must be passed by the Stan interface
- Data is validated; happens once per execution

THE TRANSFORMED DATA BLOCK

- **Declare** and **define** data; validated at end of block
- Transformed data is not read from the Stan interface
- Data variables are in scope
- Variables are initialized to
  - NaN (not a number) for real variables
  - 0 for integer variables

DEFINING TRANSFORMED DATA

- Access to
  - operators
  - built-in math functions
  - built-in matrix functions
- Assignment
  - =
    (will be deprecated in v2.10)
  - Statically typed
- Stan compiler will prevent type mismatches

OPERATORS

- Logical operators
  - `||, &&, ==, !=, <, <=, >, >=`
- Arithmetic and matrix
  - `!, -, +`
  - `+, -, *, /, \, %, ^`
  - `.*, ./, ‘`

BUILT-IN MATH FUNCTIONS

- All built-in C++ functions and operators
- Extensive library of statistical functions
  - `softmax()`, `lgamma()`, `digamma()`, `beta()`, …
- Efficient, arithmetically stable compound functions
  - `multiply_log()`, `log_sum_exp()`, `log_inv_logit()`, …

BUILT-IN MATRIX FUNCTIONS

- Basic arithmetic
- Element-wise arithmetic: ./ .*
- Solvers: matrix division, (log) determinant, inverse
- Decompositions: QR, Eigenvalues, Eigenvectors
- Compound operations: quadratic forms
- Ordering, slicing, broadcasting: sort, rank, block, rep
- Reductions: sum, product, norm
- Specialization: triangular, positive-definite, etc.

LOOPS, CONDITIONALS, BLOCKS, HELPER FUNCTIONS

- For loop: `for (n in 1:N) …`
- while loop: `while (cond) …`
- conditional: `if (cond) … else if (cond) … else …`
- blocks: `{ … }` (local variables can be declared at the top)
- helper functions:
  - `print("message", expression, …)` – prints message and expressions
  - `reject("message")` – throws an error (used for debugging) with the message specified

EXERCISES

1. Transformed data only:
   - create an int 10-array, set 3 of them to 1, the rest to 0
   - print uninitialized variable (real, int, vector)
   - use an uninitialized variable in an expression
2. Define data block for this data:

```
N <- 10
y <- c(0, 1, 1, 0, 0, 1, 0, 0, 0, 1)
```

   - Use transformed data to count the number of 1's in y

RECAP: DATA AND TRANSFORMED DATA

- data and transform data is the data in $p(\theta, x)$

- Both blocks are executed once for the whole program
- Execution is fast
- Both blocks validate data
- Variables in transformed data are not saved

PARAMETERS AND TRANSFORMED PARAMETERS

PARAMETERS

- **Declare** variables in the same way as data
- int parameters are not allowed
  Not differentiable; Stan language limited by inference algorithms
- Parameters with constraints has implicit transforms;
  changes constrained parameters to unconstrained parameters:
  guarantees sampling is over the range provided
- Can’t define the value of a parameter
  (no assignment)

TRANSFORMED PARAMETERS

- **Declare** and **define** variables
- int transformed parameters not allowed
- Define transformed variables as a function of data, transformed data, and parameters
- Transformed parameters are saved, by default (alternative: define as a local variable in model block)

EXERCISES

1. Improper posterior. Run this model. (no data)

```
parameters {
  real theta;
}
model {
}
```

```
fit <- stan("example.stan")
```

2. Proper posterior. Put lower and upper bound on theta.
   Run new model.

MODEL

WRITING A JOINT MODEL

- Data and parameters of model defined
- Model block: log joint probability

- `target +=`
  – directly increments the log probability
- sampling statements provide convenient, often efficient shortcuts

EXAMPLE: BERNOULLI COIN FLIP

One way of writing this model:

$$\theta \sim \text{Beta}(1, 1)$$
$$y_i \sim \text{Bernoulli}(\theta)$$

Another is:

$$\Pr(\theta, y_1, y_2, \ldots) = \prod_i \theta^{y_i} (1 - \theta)^{1 - y_i}$$

Let's start by coding it directly.

EXAMPLE MODEL: BERNOULLI COIN

- Start with data:
  - N: number of flips
  - y: N-array of int between 0 and 1
- Parameters:
  - theta: real between 0 and 1
- Model
  - use "target += " within a loop

EXAMPLE MODEL: BERNOULLI COIN

```
data {
  int N;
  int<lower=0, upper=1> y[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  target += log(1);
  for (n in 1:N)
    target += log(if_else(y[n] == 1,
                          theta, 1 - theta)));
}
```

ANY ARBITRARY MODEL CAN BE WRITTEN USING

```
target +=
```

EXAMPLE MODEL: EMPTY MODEL

Suppose you have $\theta \in (0, 1)$

and the model is empty: $\log(p(\theta)) = 0$

$\propto \log(1)$

or equivalently: $p(\theta) \propto 1$

What is this model?

EXAMPLE MODEL: EMPTY MODEL

```
parameters {
  real<lower=0, upper=1> theta;
}
model {
}

/* or equivalently
target += 0; // or any constant
*/
```

EXAMPLE MODEL: EMPTY MODEL

- Not defining prior is adding a 0 to joint probability
- Unconstrained parameters have improper prior
- Posterior must be proper

TARGET +=

- define models by incrementing log probability of model
- why log probability?
  - numeric stability
- Available functions
  - math
  - matrix
  - probability distributions

USING THE REFERENCE MANUAL

- Pull up the manual

- distribution functions
- other functions

EXAMPLE MODEL: BERNOULLI COIN (ORIGINAL)

```
data {
  int N;
  int<lower=0, upper=1> y[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  for (n in 1:N)
    target += if_else(y[n] == 1,
                      theta, 1-theta));
}
```

EXAMPLE MODEL: BERNOULLI COIN (DISTRIBUTION FUNCTION)

```
data {
  int N;
  int<lower=0, upper=1> y[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  for (n in 1:N)
    target += bernoulli_lpmf(y[n] | theta);
}
```

VECTORIZATION

- Vectorized statements save calculations
  (where it can be saved)
- What functions are vectorized?
  Plural in manual: reals, vectors, ints

EXAMPLE MODEL: BERNOULLI COIN (VECTORIZED)

```
data {
  int N;
  int<lower=0, upper=1> y[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  target += bernoulli_lpmf(y | theta));
}
```

"SAMPLING" STATEMENTS

- Syntactic sugar for distribution functions:

```
target += foo_lpdf(lhs | arg1, arg2, …))
target += foo_lpmf(lhs | arg1, arg2, …))
```

- Sampling statements:

```
lhs ~ foo(arg1, arg2, …)
```

- Drops constant term

- Stan **does not** do rejection sampling.
  There is no "drawing" from the distribution

EXAMPLE MODEL: BERNOULLI COIN (SAMPLING)

```
data {
  int N;
  int<lower=0, upper=1> y[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  for (n in 1:N)
    y[n] ~ bernoulli(theta);
}
```

EXAMPLE MODEL: BERNOULLI COIN (VECTORIZED SAMPLING)

```
data {
  int N;
  int<lower=0, upper=1> y[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  y ~ bernoulli(theta);
}
```

QUESTIONS? NEXT

- More hands-on examples
- Truncation
- Functions
- Generated Quantities

FUNCTIONS

USER-DEFINED FUNCTIONS

```
functions {

}
data { ... }
```

- First block in Stan program
- Types in signature a little different: lose dimensions
- All arguments mandatory
- Must return
- Can forward declare, if necessary

EXAMPLE FUNCTION

```
functions {
  int fib(int n);

  int fib(int n) {
    if (n > 2)
      return n;
    else
      return fib(n - 1) + fib(n - 2);
  }

}
...
```

EXAMPLE DISTRIBUTION (WILL ERROR IN 2.11, BUT THIS IS THE SYNTAX)

```
functions {
  real foo_lpdf(real y, real theta) {
    …
  }
}
...
model {
   ...
   y ~ foo(theta);
   target += foo_lpdf(y | theta);
}
```

GENERATED QUANTITIES

GENERATED QUANTITIES

- Most often used for posterior predictive checks
- Variables in earlier blocks in scope (not local variables)
- Can use random number generators
- Will be saved in output

RSTAN

RSTAN

- stan
  - chains
  - iter
  - warmup
  - seed
- plot(fit)
- traceplot(fit)
- extract(fit)
- as.matrix(), as.data.frame(), as.array()
- sampler_params <- get_sampler_params(fit1, inc_warmup = TRUE)
- summary(do.call(rbind, sampler_params), digits = 2)
- pairs(fit1, pars = c("mu", "tau", "lp__"), las = 1)
- lookup("dnorm")

- stan_rdump()
- read_stan_csv()

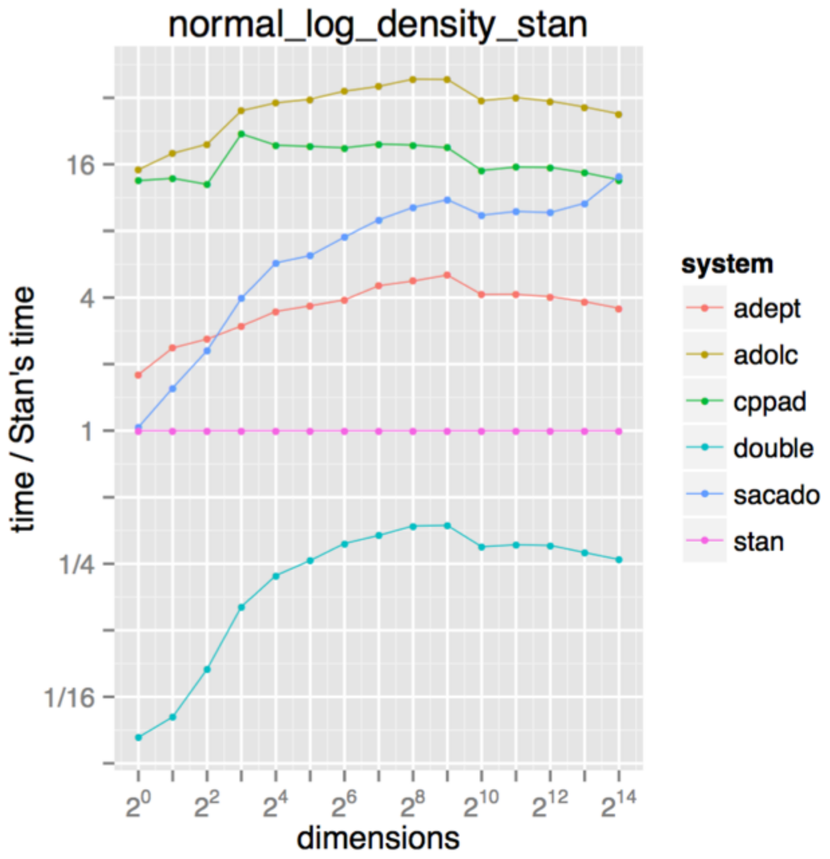
STAN'S BREAKTHROUGHS

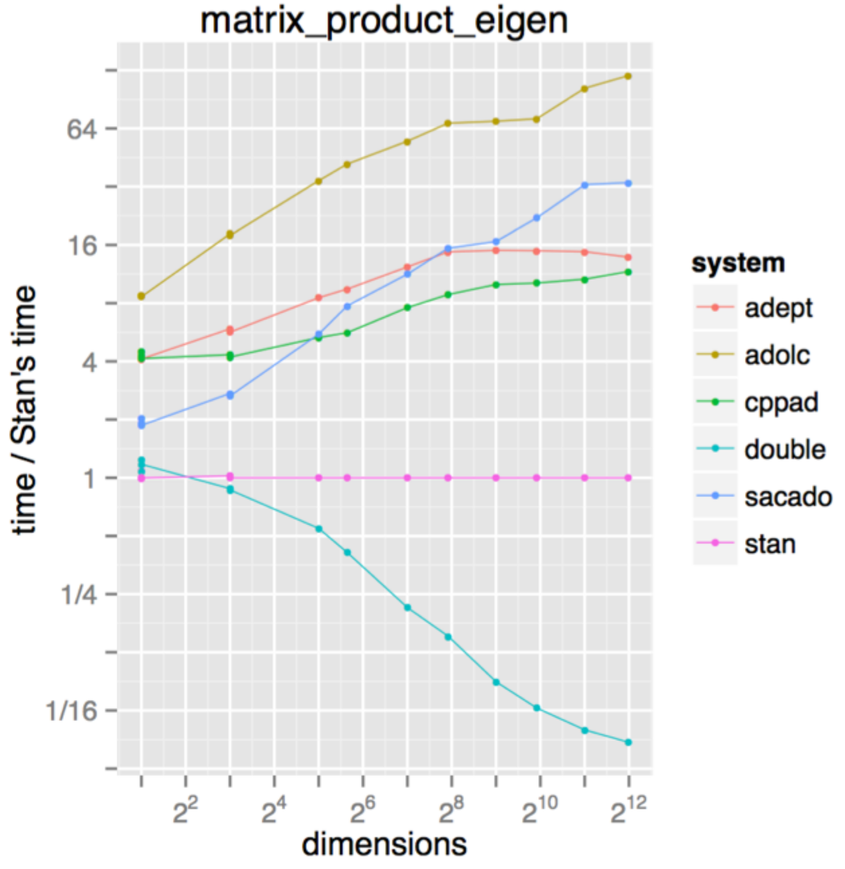
STAN'S BREAKTHROUGHS

- **Automatic differentiation**
  - Works for **all models** written in the Stan language
  - Efficient implementation, i.e. fast
- **Transforms**
  - Efficient HMC requires unbounded parameters in $\mathbb{R}^N$
  - Constraints in variable declaration automatically transform; includes change of variables term (log abs determinant)
- **No-U-Turn sampler**
  - Adapts the number of steps per iteration
  - Allows Stan to run black-box for most programs

AUTODIFF COMPARISON

- For open-source C++ packages: Stan is **fastest** (for gradients), most **general** (functions supported), and most easily **extensible** (simple OO)





NUTS VS HMC

- 250-D normal and logistic regression models
- Vertical axis: ess / sample (bigger is better)
- Left: NUTS, right: HMC with increasing time