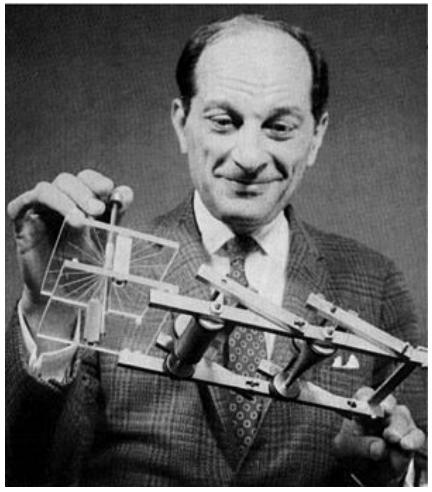
Introduction to Stan for Markov Chain Monte Carlo

Matthew Simpson

Department of Statistics, University of Missouri

April 11, 2017

Stan is...



Stanislaw Ulam, inventor of Monte Carlo methods.

`http://mc-stan.org/`

A **probabilistic programming language** that implements **Hamiltonian Monte Carlo (HMC)**, variational Bayes, and (penalized) maximum likelihood estimation.

Available on Linux, Mac, and Windows with interfaces in **R**, **Python**, shell (command line), MATLAB, Julia, Stata, and Mathematica.

Markov chain Monte Carlo (MCMC)

Goal: sample from some target density $\pi(\boldsymbol{q})$.

Create a Markov chain with transition density $k(\boldsymbol{q}'|\boldsymbol{q})$.

- Start with arbitrary $\boldsymbol{q}^{(0)}$ and repeatedly sample $\boldsymbol{q}^{(t+1)} \sim k(\boldsymbol{q}'|\boldsymbol{q}^{(t)})$.
- Under some conditions $\boldsymbol{q}^{(t)} \to \boldsymbol{q}$ in distribution.
- Additionally with geometric ergodicity:

$$\frac{1}{T}\sum_{t=1}^{T} f(\boldsymbol{q}^{(t)}) \to \mathrm{N}\left(\mathrm{E}[f(\boldsymbol{q})], \frac{\mathrm{var}[f(\boldsymbol{q})]}{ESS}\right).$$

Auxillary variable MCMC: construct a variable $\boldsymbol{p}$ with joint density $\pi(\boldsymbol{q}, \boldsymbol{p}) = \pi(\boldsymbol{p}|\boldsymbol{q})\pi(\boldsymbol{q})$.

- Construct a Markov chain for $(\boldsymbol{q}, \boldsymbol{p})$ and throw away the sampled $\boldsymbol{p}^{(t)}$s.
- Ex: data augmentation, slice sampling, HMC, etc.

HMC in Theory

Construct $\boldsymbol{p}$ in a special way. Let $\boldsymbol{q}, \boldsymbol{p} \in \Re^n$ and:

$$V(\boldsymbol{q}) = -\log \pi(\boldsymbol{q}) \text{ — } \textit{potential energy}.$$
$$T(\boldsymbol{q}, \boldsymbol{p}) = -\log \pi(\boldsymbol{p}|\boldsymbol{q}) \text{ — } \textit{kinetic energy}.$$
$$H(\boldsymbol{q}, \boldsymbol{p}) = V(\boldsymbol{q}) + T(\boldsymbol{q}, \boldsymbol{p}) \text{ — } \textit{Hamiltonian}, \text{ total energy.}$$

where $\boldsymbol{q}$ denotes *position* and $\boldsymbol{p}$ denotes *momentum*.

Energy-preserving evolution in time is defined by Hamilton's equations:

$$\frac{\mathrm{d}\boldsymbol{p}}{\mathrm{d}t} = -\frac{\partial H}{\partial \boldsymbol{q}}; \qquad \frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = +\frac{\partial H}{\partial \boldsymbol{p}}.$$

How to implement HMC (in theory):

1. Sample momenta variables: $\boldsymbol{p}' \sim \pi(\boldsymbol{p}|\boldsymbol{q}^{(t)})$.
2. Run Hamiltonian evolution forward in time from $(\boldsymbol{q}^{(t)}, \boldsymbol{p}')$ for a some amount of *integration time* to obtain $(\boldsymbol{q}^{(t+1)}, \boldsymbol{p}^{(t+1)})$.

HMC in Pictures

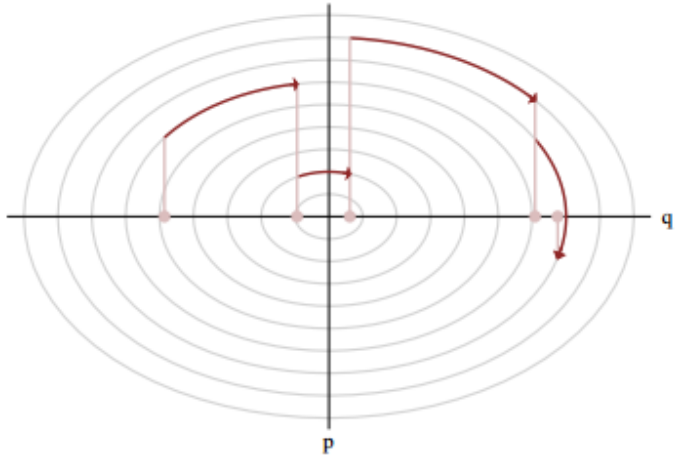


FIG 3. *Every Hamiltonian Markov chain alternatives between a deterministic Hamiltonian flow that explores a single level set (dark red) and a momentum resampling that transitions between level sets with a random walk (light red). The longer the flow is integrated the more efficiently the Markov chain can explore each level set and the smaller the autocorrelations will be. When the flow is integrated for only an infinitesimally small time the Markov chain devolves into a Langevin diffusion.*

HMC samples a level set, then moves along that set.

Long integration time $\implies$ $\approx$ zero autocorrelation in the chain.

(Picture stolen from `https://arxiv.org/pdf/1601.00225.pdf`)

HMC in Practice

To implement HMC for a **differentiable** target $\pi(\boldsymbol{q})$ you need:

1. No discrete valued parameters in $\boldsymbol{q}$.
   - Usually can integrate them out, e.g. mixture models.
2. No constrained parameters in $\boldsymbol{q}$.
   - Stan: transform and compute the log-Jacobian automatically.
3. The gradient vector of $\log \pi(\boldsymbol{q})$.
   - Stan: use C++ autodiff library to do this automatically and accurately.
4. Choose a kinetic energy, i.e. $\pi(\boldsymbol{p}|\boldsymbol{q})$.
   - Stan: $\mathrm{N}(\mathbf{0}, \boldsymbol{M})$ and tune $\boldsymbol{M}$ during warmup. (Typical HMC)
   - More intelligent: $\boldsymbol{M}(\boldsymbol{q})$. (Riemannian HMC; future Stan)

HMC in Practice (continued)

To implement HMC for a **differentiable** target $\pi(\boldsymbol{q})$ you need:

5. Numerical integrator for Hamilton's equations.
   - Need to make an adjustment to the Hamiltonian flow and use a Metropolis correction to ensure detailed balance.
   - Typically use leapfrog integration $\implies$ how many leapfrog steps?
   - Stan: adapt number of steps to hit a target Metropolis acceptance rate.

6. An integration time. How long is long enough?
   - Old Stan: No U-Turn Criterion / No U-Turn Sampler (NUTS)

     "stop when we start heading back toward where we started."
   - New Stan: eXhaustive HMC (XHMC/XMC/better NUTS)

     "stop when it looks like autocorrelation should be low."

Why Hamiltonian Monte Carlo?

The long answer:

- **Everything You *Should* Have Learned About MCMC** (Michael Betancourt)
  `https://www.youtube.com/watch?v=DJ0c7Bm5Djk&feature=youtu.be&t=4h40m10s`
- **A Conceptual Introduction to HMC** (Michael Betancourt)
  `https://arxiv.org/pdf/1701.02434.pdf`
- **Hamiltonian Monte Carlo for Hierarchical Models** (Michael Betancourt and Mark Girolami)
  `https://arxiv.org/pdf/1312.0906.pdf`

The short answer:

- Works in high dimensions.
- More robust.
- Makes noise when it fails.

Works in high dimensions

We are interested in expecations of the form $\int f(\boldsymbol{q})\pi(\boldsymbol{q})d\boldsymbol{q}$.

- Naively: focus on areas where $\pi(\boldsymbol{q})$ (density) is large.
- Better: where where $\pi(\boldsymbol{q})d\boldsymbol{q}$ (mass) is large; "typical set."

Typical Set:

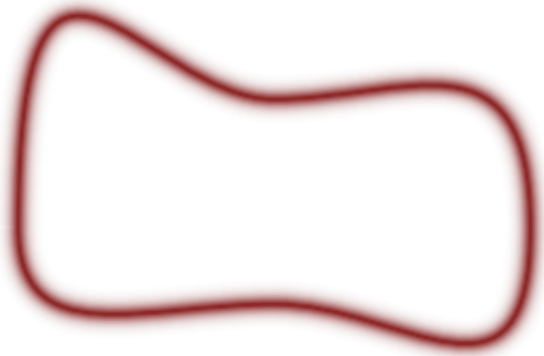


FIG 4. *In high-dimensional parameter spaces probability mass, $\pi(q)\,\mathrm{d}q$, and hence the dominant contributions to expectations, concentrates in a neighborhood called the* typical set. *In order to accurately estimate expectations we have to be able to identify where the typical set lies in parameter space so that we can focus our computational resources where they are most effective.*

In high dimensions, this is essentially a surface.

- Random walk methods have to take tiny steps.
- Gibbs methods take a long time to move around the surface.
- Modes are far away from mass $\rightarrow$ mode-based methods fail.

Robustness and Noisy Failure

HMC has guaranteed geometric ergodicity in a larger class of target densities than alternatives.

When geometric ergodicity fails, HMC often won't sample due to numerically infinite gradients ("divergent transitions").

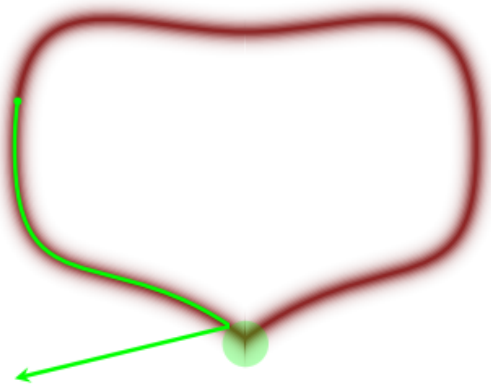


FIG 35. *Neighborhoods of high curvature in the typical set (green) that frustrate geometric ergodicity are also pathological to symplectic integrators, causing them to diverge. This confluence of pathologies is advantageous in practice because we can use the easily-observed divergences to identify the more subtle statistical pathologies.*

- Caused by weird posterior geometries (reparameterize).
- Common in hierarchical models.
- Also a problem in Gibbs samplers, but they still give output.

Using Stan: Resources

How to install Stan and `rstan` (Follow the directions carefully!):

- `https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started`

Stan manual (it's very good and constantly being improved):

- `https://github.com/stan-dev/stan/releases/download/v2.14.0/stan-reference-2.14.0.pdf`

Links to the manual, examples, tutorials, and case studies:

- `http://mc-stan.org/documentation/`

`rstan` documentation:

- `http://mc-stan.org/interfaces/rstan.html`

A brief guide to Stan's warnings:

- `http://mc-stan.org/misc/warnings.html`

A shorter intro with a different emphasis:

- `http://mlss2014.hiit.fi/mlss_files/2-stan.pdf`

Using Stan: A Simple Example

Running example: regression with observation errors.

Fake data generated in `stanintro.R`:

```
n <- 100
alpha <- 150
beta <- c(0.05, 100, -.003)
x <- cbind(runif(n, -1, 1)*100, rnorm(n)/10 + 10, rgamma(n, 1, 1/100))
sigma <- 10
ses <- 10*rep(sigma, n)
theta <- rnorm(n, alpha + x%*%beta, sigma)
y <- rnorm(n, theta, ses)
```

Start by ignoring observation errors:

$$y_i \overset{ind}{\sim} \mathrm{N}(\alpha + \boldsymbol{x}_i'\boldsymbol{\beta}, \sigma^2)$$

for $i = 1, 2, \ldots, N$, with independent priors:

$$\alpha \sim \mathrm{Cauchy}(100, 1000^2),$$

$$\beta_k \overset{iid}{\sim} \mathrm{N}(10, 100^2),$$

$$\sigma \sim \mathrm{half} - \mathrm{T}_5(0, 100^2).$$

Using Stan: A Simple Example

Define the model in a `.stan` file, e.g. `regression.stan`:

```
data {
  int<lower = 1> n_obs;
  int<lower = 1> n_cov;
  vector[n_obs] y;
  matrix[n_obs, n_cov] x;
  real beta_prior_mn;
  real<lower = 0> beta_prior_sd;
  real alpha_prior_loc;
  real<lower = 0> alpha_prior_scale;
  real<lower = 0> sig_prior_scale;
  real<lower = 0> sig_prior_df;
}
parameters {
  real alpha;
  vector[n_cov] beta;
  real<lower = 0> sigma;
}
model {
  y ~ normal(alpha + x*beta, sigma);
  alpha ~ cauchy(alpha_prior_loc, alpha_prior_scale);
  beta ~ normal(beta_prior_mn, beta_prior_sd);
  sigma ~ student_t(sig_prior_df, 0, sig_prior_scale);
}
```

The .stan File

Defines a target density as a function of data and parameters.

- Data: all things that are fixed during MCMC, including prior hyperparameters and what we normally think of as "data".
- Parameters: all things we want/need to sample from.

The file is composed of several "program blocks" in a specific order.

- "parameters" and "model" blocks are mandatory.
- "data" block is necessary to read data into Stan.
- And several others.

```
data {
  // define all variables to be read into Stan here
}
parameters {
  // define all parameters of the target density here
}
model {
  // define model as a function parameters and data here
}
```

Defining variables

A variable can be defined at the beginning of any block,
or at the beginning of any code chunk (`{...}`).

- Stan has two basic data types: `int` and `real`.
- Vectors and matrices are collections of `reals`.
- Can define arrays of `ints` or `reals`... or vectors or whatever.
- Must define every variable that Stan will use.
- Arrays, vectors, and matrices must have defined dimensions.
- Can (and should) specify any relevant bounds for all variables (error checking & more).
- Stan uses `R` style 1-based indexing.

Basic syntax: `int n;` or `real y;`.

Some example variable definitions

```
int<lower = 1> A;  // A >= 1 (constraints are inclusive)
real<upper = 0> B; // B <= 0
real<lower = 0, upper = 1> C; // 0 <= C <= 1
vector[10] D;                 // vector of 10 reals
vector<lower = 0>[10] E;      // vector of 10 reals >= 0
row_vector[5] F;              // row vector of 5 reals
matrix[10, 5] H;              // 10x5 matrix of reals
cov_matrix I[2];              // 2x2 PD matrix
corr_matrix J[2];             // 2x2 PD matrix
cholesky_factor_cov K[2];     // lower triangular
cholesky_factor_corr L[2];    // lower triangular
simplex M[5]; // each 0 < M[i] < 1; sum_i M[i] = 1
```

Note: C-style syntax: end statements with a semicolon (;),
and '//' comments the rest of the line (like # in R).

Array / matrix indexing: mostly like R

Indexing order: array dimensions, then row, then column.

```
real A[N];           // N-dim array
vector[N] B;         // N-dim vector
matrix[N, M] C;      // NxM matrix
vector[N] D[M];      // M N-dim vectors
matrix[N, M] E[K]; // K NxM matrices
A[3]; B[3];          // access 3rd element
A[1:3]; B[1:3];      // 1st - 3rd elements
A[ii]; B[ii]; // if ii = [1, 3], 1st and 3rd elements
C[1,2];              // 1st row / 2nd column
D[1,2];              // 2nd element of 1st vector
C[1];                // first row
D[1];                // first vector
C[,1];               // first column
D[,1];               // vector of 1st elements
E[1,1:4,1:4];        // top left 4x4 submatrix of 1st matrix
```

...and combinations of the above.

Data block

Define all data that will be read into Stan. `regression.stan`:

```
data {
  int<lower = 1> n_obs;
  int<lower = 1> n_cov;
  vector[n_obs] y;
  matrix[n_obs, n_cov] x;
  real beta_prior_mn;
  real<lower = 0> beta_prior_sd;
  real alpha_prior_loc;
  real<lower = 0> alpha_prior_scale;
  real<lower = 0> sig_prior_scale;
  real<lower = 0> sig_prior_df;
}
```

This block *only* consists of variable definitions.

Any constraints are checked once before the sampler is run.

Parameters block

Define all parameters in the model. `regression.stan`:

```
parameters {
  real alpha;
  vector[n_cov] beta;
  real<lower = 0> sigma;
}
```

This block *only* consists of variable definitions.

Stan automatically transforms constrained parameters to unconstrained Euclidean space and computes the relevant Jacobian.

- Stan can hand simple constraints stated in terms of lower & upper bounds, e.g. `real<lower = mu_x> mu_y;`,
- ...and certain hardcoded complex constraints, e.g. covariance and correlation matrices, Cholesky factors of both, etc.

Model block

Define the model in terms of parameters and data.
`regression.stan`:

```
y ~ normal(alpha + x*beta, sigma);
alpha ~ cauchy(alpha_prior_loc, alpha_prior_scale);
beta ~ normal(beta_prior_mn, beta_prior_sd);
sigma ~ student_t(sig_prior_df, 0, sig_prior_scale);
```

Each sampling statement ('~') adds the relevant quantity to the target log-density. The LHS must be a previously defined variable (data, parameter, transformed data, or transformed parameter blocks).

The LHS cannot be an arbitrary function of a variable.

Use transformed data/parameters blocks to solve this (more later).

Arithmetic operators

'*' is matrix multiplication when its arguments are not scalars. Similarly '/' and '\' are matrix division:

$$A * B = AB, \qquad A/B = AB^{-1}, \qquad A\backslash B = A^{-1}B$$

Use ' `.*` ' and ' `./` ' for elementwise operations.

...but give these operators some space:

`A.*B` will throw an error, `A .* B` will not.

Otherwise, most things work just like `R`, except Stan is finicky about dimensions matching (this is good for catching errors).

- other differences: `||` for 'or', `&&` for 'and', `X'` for X transpose.

Check the manual for efficient specialized functions for many common linear algebra (and other) operations, e.g.:

`crossprod(); tcrossprod(); dot_product(); quad_form();`

Vectorization vs loops

Sampling statements are vectorized when it makes sense.

I.e. the following are equivalent ways of coding y's model:

- `y ~ normal(alpha + x*beta, sigma);`
- ```
  for(i in 1:n_obs){ // same as in R
    y[i] ~ normal(alpha + x[i]*beta, sigma);
  }
  ```

Stan is written in C++ so for loops are fast...

But the `autodiff` library is much faster on vectorized models.

- Much faster gradient computations.
- Sampling is cheaper per-iteration (per-leapfrog step).

Upshot: vectorize wherever you can (see manual).
  ```

Fitting the model in R

Fitting a regression:

```
## create list of all variables in the data block
regdat <- list(n_obs = n, n_cov = length(beta), y = y, x = x,
               alpha_prior_loc = 100, alpha_prior_scale = 1000,
               beta_prior_mn = 10, beta_prior_sd = 100,
               sig_prior_df = 5, sig_prior_scale = 100)

## initialize the model; takes 15-30 seconds
regfit0 <- stan("regression.stan", data = regdat, chains = 1, iter = 1)
## ignore compiler warnings

## sample the model
regfit <- stan(fit = regfit0, data = regdat, cores = 4, chains = 4,
               warmup = 2000, iter = 4000, open_progress = FALSE)
## about 3 minutes to fit per chain
```

The stan() function in rstan

Required arguments:

- A model — ".stan" file, or `fit = stanfit`, a `stan` object.
- data — list of all variables in the data block of the model.
  (only required if the model has a data block)

Useful named arguments and their defaults:

- `cores = 1` — number of cores to use.
  - Parallelizes *across* chains, not within.
- `chains = 4` — number of chains.
- `iter = 4000` — total number of iterations per chain.
- `warmup = iter/2` — iterations used for tuning / burn-in.

Starting values, tuning, etc., taken care of automatically, but much of this is exposed in `stan()`.

Running stan()

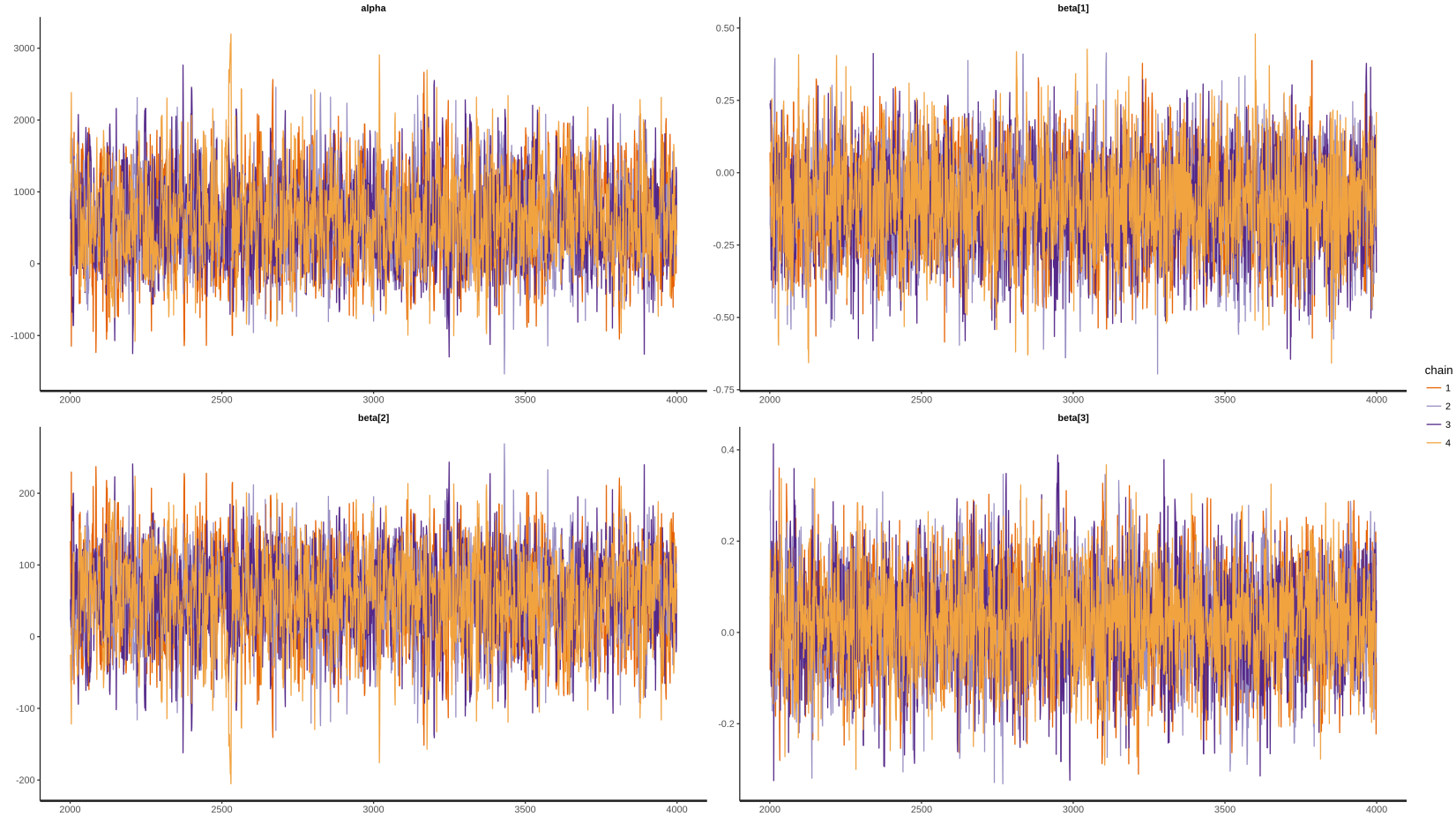
```
> regfit0 = stan("regression.stan", data = regdat,
+                chains = 1, iter = 1)
...
warning: "BOOST_NO_CXX11_RVALUE_REFERENCES" redefined
...
```

- Ignore compiler warnings like the one above.

```
> regfit = stan(fit = regfit0, data = regdat, cores = 4,
+          chains = 4, warmup = 2000, iter = 4000)
...
Chain 4, Iteration: 4000 / 4000 [100%]  (Sampling)
 Elapsed Time: 79.7899 seconds (Warm-up)
               96.0802 seconds (Sampling)
               175.87 seconds (Total)
```

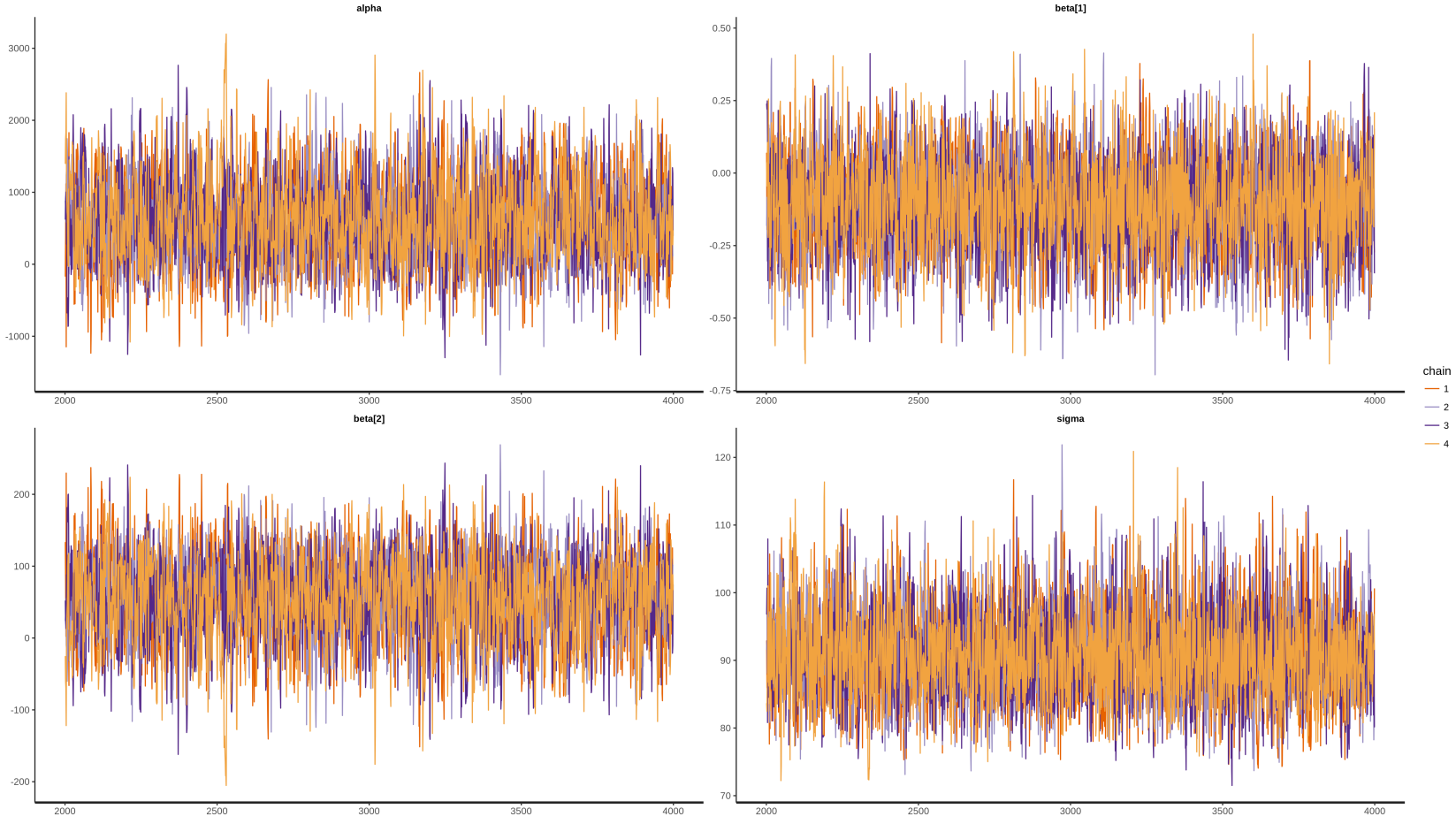
Traceplots

```
> traceplot(regfit, pars = c("alpha", "beta"))
```



More Traceplots

```
> traceplot(regfit, pars = c("alpha", paste("beta[", 1:2, "]", sep=""), "sigma"))
```



Posterior Summaries

```
> summary(regfit)$summary
                mean      se_mean          sd         2.5%          25%
alpha    601.47469242 11.155331426 593.8138899 -490.6827947  191.95702515
beta[1]   -0.10907827  0.002090211   0.1628532   -0.4288071   -0.21721217
beta[2]   54.30721025  1.113999466  59.3528808  -67.0703765   16.13727134
beta[3]    0.02146048  0.001504924   0.1060208   -0.1860389   -0.05051183
sigma     90.81032505  0.088702720   6.7013775   78.7561024   86.06563171
lp__    -496.94003128  0.028859472   1.6090604 -500.9452935 -497.77083742
                 50%           75%         97.5%    n_eff      Rhat
alpha    570.66425923  9.840412e+02 1823.5656680 2833.582 1.0024421
beta[1]   -0.10911846  4.565740e-04    0.2095149 6070.329 1.0000127
beta[2]   57.29746473  9.525990e+01  163.6458877 2838.662 1.0024554
beta[3]    0.02158836  9.293641e-02    0.2261381 4963.105 0.9998738
sigma     90.37856417  9.514879e+01  105.0946718 5707.610 0.9998136
lp__    -496.63425539 -4.957597e+02 -494.7635700 3108.622 1.0010516
```

Can also specify which parameters you want summaries of, e.g.

```
> summary(regfit, pars = c("beta"))$summary
```

More Posterior Summaries

`lp__`: value of the log posterior at every iteration.
`n_eff`: effective sample size.

- equivalent number of iid draw for that parameter.

`Rhat`: potential scale reduction factor.

- convergence diagnostic based on multiple chains.
- `Rhat` < 1.01: no evidence against convergence *for that parameter*.

`shinystan` R package:

- `http://mc-stan.org/interfaces/shinystan`

Extract posterior draws into a named list:

```
> regfitdraws <- extract(regfit)
> str(regfitdraws, 1)
List of 4
 $ alpha: num [1:8000(1d)] 1942.1 30.9 158.2 824.3 1634.2 ...
 $ beta : num [1:8000, 1:3] -0.278 -0.124 -0.1108 -0.1193 -0.0119 ...
 $ sigma: num [1:8000(1d)] 100.1 99 91.5 81.6 93.2 ...
 $ lp__ : num [1:8000(1d)] -499 -497 -495 -497 -497 ...
```

Regression with Centered and Scaled Data

Adapted from `regression_cs.stan`.

```
data {
  ...
}
transformed data {
  vector[n_obs] y_cs;
  matrix[n_obs, n_cov] x_cs;
  real y_mn;
  real<lower = 0> y_sd;
  vector[n_cov] x_mn;
  vector<lower = 0>[n_cov] x_sd;

  // center and scale y
  y_mn = mean(y);
  y_sd = sd(y);
  y_cs = (y - y_mn)/y_sd;

  // center and scale x
  for(i in 1:n_cov){
    x_mn[i] = mean(x[,i]);
    x_sd[i] = sd(x[,i]);
    x_cs[,i] = (x[,i] - x_mn[i]) / x_sd[i];
  }
}
parameters {
  real alpha_cs;
  vector[n_cov] beta_cs;
  real<lower = 0> sigma_cs;
}
```

This model is equivalent to `regression.stan` as long as:

- We translate the priors on `alpha`, `beta`, and `sigma` into priors on their `_cs` versions.
- We transform back to `alpha`, `beta`, and `sigma` if that is what we want to do inference on.

Carefully Taking Into Account Transformations

From `regression_cs.stan` (see file for full text).

```
transformed data {
  ...
  // center and scale y
  y_mn = mean(y);
  y_sd = sd(y);
  y_cs = (y - y_mn)/y_sd;

  // center and scale x
  for(i in 1:n_cov){
    x_mn[i] = mean(x[,i]);
    x_sd[i] = sd(x[,i]);
    x_cs[,i] = (x[,i] - x_mn[i]) / x_sd[i];
  }

  // priors on _cs parameters
  x_mnsd = x_mn ./ x_sd;
  beta_cs_prior_mn = x_sd * beta_prior_mn / y_sd;
  beta_cs_prior_sd = x_sd * beta_prior_sd / y_sd;
  alpha_cs_prior_loc = (alpha_prior_loc-y_mn)/y_sd;
  alpha_cs_prior_scale = alpha_prior_scale / y_sd;
  sig_cs_prior_scale = sig_prior_scale / y_sd;
}
```

```
parameters {
  real alpha_cs;
  vector[n_cov] beta_cs;
  real<lower = 0> sigma_cs;
}
model {
  y_cs ~ normal(alpha_cs + x_cs*beta_cs, sigma_cs);
  beta_cs ~ normal(beta_cs_prior_mn, beta_cs_prior_sd);
  alpha_cs ~ cauchy(alpha_cs_prior_loc + dot_product(x_mnsd
  sigma_cs ~ student_t(sig_prior_df, 0, sig_cs_prior_scale)
}
generated quantities {
  real alpha;
  vector[n_cov] beta;
  real<lower = 0> sigma;

  beta = (beta_cs ./ x_sd) * y_sd;
alpha = alpha_cs * y_sd - dot_product(x_mn, beta) + y_mn;
  sigma = sigma_cs * y_sd;
  }
```

Reduces fit time per chain from 3 minutes to 4 seconds.

Transformed data and parameters

Transformed data: transform once before running MCMC.

Transformed parameters: transform once per leapfrog step.

For example:

```
data {
  ...
}
transformed data {
  vector[n_obs] log_y;
  log_y = log(y);
}
parameters = {
  ...
}
transformed parameters {
  real<lower = 0> sigma;
  sigma = sqrt(sigma2);
}
model {
  log_y ~ normal(mu, sigma);
  ...
}
```

Other places to put transformations

Model block:

- Computed once per leapfrog step — useful for intermediate quantities you don't want MCMC draws for.
- No constraints allowed here.

```
model {
  real sigma;
  sigma = sqrt(sigma2);
  log_y ~ normal(mu, sigma);
  ...
}
```

Generated quantities block:

- Computed once per MCMC iter — useful for quantities you want draws for, but aren't needed for computing the posterior.
- Can also put random draws here, e.g. for predictive dists.
- Constraints allowed, but not necessary.

```
model { ... }
generated quantities {
  real muoversigma;
  muoversigma = mu / sqrt(sigma2);
}
```

Where should I put my transformation for max efficiency?

Transformed data block:

- All pure functions of data, or other intermediate quantities that don't depend on parameters.

Transformed parameters block:

- Only place to put transformations to automatically take into account the Jacobian.
- **Any quantity that is a function of parameters and on the LHS of a sampling statement ($\sim$) must go here.**
- Avoid putting other quantities here because computing Jacobians is slow and unnecessary.

Model block:

- Intermediate quantities used on RHS of sampling statements.

Generated quantities block:

- Quantities you want MCMC draws for that aren't already in the parameters or transformed parameters block.

All Possible Program Blocks

Only the parameters and model blocks are mandatory, but the blocks must appear in this order.

```
functions {
  // define user defined functions here (see manual)
}
data {
  // define all input (data / hyperparameter) variables here
}
transformed data {
  // create transformations of data and other intermediate
  // quantities that don't depend on parameters here
}
parameters {
  // define all model parameters here
}
transformed parameters {
  // create any needed transformations of parameters here
}
model {
  // create the log posterior density here
}
generated quantities {
  // create any other variables you want draws for here
}
```

Tricks for Better/Faster Samplers in Stan

HMC/Stan works best when:

- All parameters are on similar scales.
- Posterior geometries aren't "weird".

How to deal with these issues:

- Center and scale responses and covariates.
- Work on the log scale, e.g. `logy ~ normal(.,.);` is better than `y ~ lognormal(.,.);`
- Reparameterize the model — noncentered parameterizations.

Centering and scaling data often drastically speeds up model fits because it makes adaptation easier.

Reparameterization is often necessary to get a working sampler at all, especially in hierarchical models.

Dealing with Hierarchical Models

Example: same regression, now taking into account observation errors.

Now we specify the model directly on centered and scaled y_i and $\boldsymbol{x}_i'$:

$$y_i \overset{ind}{\sim} \mathrm{N}(\theta_i, s_i^2),$$
$$\theta_i \overset{ind}{\sim} \mathrm{N}(\alpha + \boldsymbol{x}_i'\boldsymbol{\beta}, \sigma^2),$$

for $i = 1, 2, \ldots, N$, where s_i is the known observation SE for y_i.

Known Observation Error Variances in Stan

From `reg_error.stan` (about 1 minute to fit):

```
data {
  int<lower = 1> n_obs;
  int<lower = 1> n_cov;
  vector[n_obs] y;
  vector<lower = 0>[n_obs] y_se;
  ...
}
transformed data {
  ...
  // note: have to convert y_se to y_cs_se
}
parameters {
  real alpha_cs;
  vector[n_cov] beta_cs;
  vector[n_obs] theta_cs;
  real<lower = 0> sigma_cs;
}
model {
  y_cs ~ normal(theta_cs, y_cs_se);
  theta_cs ~ normal(alpha_cs + x_cs*beta_cs, sigma_cs);
  beta_cs ~ normal(beta_prior_mn, beta_prior_sd);
  alpha_cs ~ cauchy(alpha_prior_loc, alpha_prior_scale);
  sigma_cs ~ student_t(sig_prior_df, 0, sig_prior_scale);
}
```

Problems

```
> regerrordat <- list(n_obs = n, n_cov = length(beta), y = y, x = x, y_se = ses,
+                     alpha_prior_loc = 0, alpha_prior_scale = 10,
+                     beta_prior_mn = 0, beta_prior_sd = 10,
+                     sig_prior_df = 5, sig_prior_scale = 10)
> regerrorfit <- stan(fit = regerrorfit0, data = regerrordat, cores = 4, chains = 4,
                      warmup = 2000, iter = 4000, open_progress = FALSE)
Warning messages:
1: There were 119 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help. See
http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
2: There were 4 chains where the estimated Bayesian Fraction of Missing Information was low. See
http://mc-stan.org/misc/warnings.html#bfmi-low
3: Examine the pairs() plot to diagnose sampling problems
```

Guide to Stan's Warnings

Summarizing from `http://mc-stan.org/misc/warnings.html`

Exception ... Hamiltonian proposal rejected:

```
Exception thrown at line 37: normal_log: Scale parameter is 0, but must be > 0!.
```

- Not a problem if the proportion of iters with exceptions is low.
- If the proportion is high: indication of a problem w/ the model.

  e.g. maybe a hierarchical variance should be zero.

Low Bayesian Fraction of Missing Information (BFMI):

- Indicates that the adaptation turned out poorly and the chain did not properly explore the posterior.
- Can solve by increasing `warmup` or reparameterizing.

Focus on Divergent Transitions

Divergent transitions after warmup:

```
1: There were 119 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help.
```

- Indicates that geometric ergodicity is breaking down. ***Just one divergent transition is enough to be concerning.***
- Solve via reparameterization or increasing `adapt_delta`.
  `stan(..., control = list(adapt_delta = .99))`

`adapt_delta` is the target Metropolis acceptance rate, which is controlled via the leapfrog step size.

Lowering the leapfrog step size forces the numerical integrator to follow the local curvature of the target distribution more closely...

But this doesn't always work! Reparameterization can solve it.

The Problem with Hierarchical Models

Good discussion in **HMC for Hierarchical Models**:
`https://arxiv.org/pdf/1312.0906.pdf`

In a nutshell: these posterior distributions have weird geometries.

- Parameters are highly correlated, and global and local correlations may be very different.
- Gibbs samplers have high autocorrelation.
- Random Walk Metropolis samplers have small step sizes.
- HMC more likely to have divergent transitions.
- Geometric ergodicity typically fails ***using any method***.

How to fix this?

- Gibbs: reparameterize, parameter expansion, interweaving.
- HMC: reparameterize, decrease step size, Riemannian HMC.

Centered vs. noncenteredparameterizations

$$y_i \overset{ind}{\sim} \mathrm{N}(\theta_i, s_i^2) \qquad \theta_i \overset{ind}{\sim} \mathrm{N}(\alpha + \boldsymbol{x}_i'\boldsymbol{\beta}, \sigma^2); \quad \text{(centered)}$$

$$y_i \overset{ind}{\sim} \mathrm{N}(\alpha + \boldsymbol{x}_i'\boldsymbol{\beta} + \sigma\varepsilon_i, s_i^2), \quad \varepsilon_i \overset{iid}{\sim} \mathrm{N}(0, 1). \quad \text{(noncentered)}$$

Define the signal-to-noise ratio: $\sigma^2/\bar{s}^2$.

Centered parameterization:

- Usually the natural way to write the model.
- Results in easy MCMC when the signal-to-noise ratio is large.

Noncentered parameterization:

- Usually found by centering and scaling: $\varepsilon_i = (\theta_i - \alpha - \boldsymbol{x}_i'\boldsymbol{\beta})/\sigma$.
- Results in easy MCMC when the signal-to-noise ratio is small.

Noncentered Regression with Observation Errors

From `reg_error_nc.stan` (about 30 seconds per chain):

```
parameters {
  real alpha_cs;
  vector[n_cov] beta_cs;
  vector[n_obs] theta_cs_raw;
  real<lower = 0> sigma_cs;
}
transformed parameters{
  vector[n_obs] theta_cs;
  theta_cs = theta_cs_raw*sigma_cs + alpha_cs + x_cs*beta_cs;
}
model {
  y_cs ~ normal(theta_cs, y_cs_se);
  theta_cs_raw ~ normal(0, 1);
  beta_cs ~ normal(beta_prior_mn, beta_prior_sd);
  alpha_cs ~ cauchy(alpha_prior_loc, alpha_prior_scale);
  sigma_cs ~ student_t(sig_prior_df, 0, sig_prior_scale);
}
```

No divergent transitions or low BFMI with this parameterization.

Stan vs Gibbs for Hierarchical Models

Stan's major advantages:

- Stan tends to let you know when geometric ergodicity fails.
- Stan tends to be more efficient.

Although the pathologies of the centered parameterization penalize all three algorithms, Euclidean Hamiltonian Monte Carlo proves to be at least an order-of-magnitude more efficient than both Random Walk Metropolis and the Gibbs sampler. The real power of Hamiltonian Monte Carlo, however, is revealed when those penalties are removed in the non-centered parameterization (Table IV).

Algorithm	Parameterization	Step Size	Average Acceptance Probability	Time (s)	Time/ESS (s)
Metropolis	Centered	$5.00 \cdot 10^{-3}$	0.822	$4.51 \cdot 10^{4}$	1220
Gibbs	Centered	1.50	0.446	$9.54 \cdot 10^{4}$	297
EHMC	Centered	$1.91 \cdot 10^{-2}$	0.987	$1.00 \cdot 10^{4}$	**16.2**
Metropolis	Non-Centered	0.0500	0.461	398	1.44
Gibbs	Non-Centered	2.00	0.496	817	1.95
EHMC	Non-Centered	0.164	0.763	154	$\mathbf{2.94 \cdot 10^{-2}}$

TABLE I. Euclidean Hamiltonian Monte Carlo significantly outperforms both Random Walk Metropolis and the Metropolis-within-Gibbs sampler for both parameterizations of the one-way normal model. The difference is particularly striking for the more efficient non-centered parameterization that would be used in practice.

From **HMC for Hierarchical Models**:

`https://arxiv.org/pdf/1312.0906.pdf`

Coming Soon: Riemannian HMC (and more)

Good Riemannian HMC deals with all of this without user input.

Momentum distribution $\boldsymbol{p}|\boldsymbol{q} \sim \mathrm{N}(\mathbf{0}, \boldsymbol{M}(\boldsymbol{q}))$.

From the manual:

Stan's Future

We're not done. There's still an enormous amount of work to do to improve Stan. Some older, higher-level goals are in a standalone to-do list:

`https://github.com/stan-dev/stan/wiki/Longer-Term-To-Do-List`

We are gradually weaning ourselves off of the to-do list in favor of the GitHub issue tracker (see the next section for a link).

Some major features are on our short-term horizon: Riemannian manifold Hamiltonian Monte Carlo (RHMC), transformed Laplace approximations with uncertainty quantification for maximum likelihood estimation, marginal maximum likelihood estimation, data-parallel expectation propagation, and streaming (stochastic) variational inference. The latter has been prototyped and described in papers.

We will also continue to work on improving numerical stability and efficiency throughout. In addition, we plan to revise the interfaces to make them easier to understand and more flexible to use (a difficult pair of goals to balance).

Later in the Stan 2 release cycle (Stan 2.7), we added variational inference to Stan's sampling and optimization routines, with the promise of approximate Bayesian inference at much larger scales than is possible with Monte Carlo methods. The future plans involve extending to a stochastic data-streaming implementation for very large-scale data problems.

Good results from setting $\boldsymbol{M}$ to a function of the Hessian of $\pi(\boldsymbol{q})$.

$\Longrightarrow$ need second order `autodiff`. They're working on it.

Thank you!

Questions?

Matthew Simpson themattsimpson@gmail.com