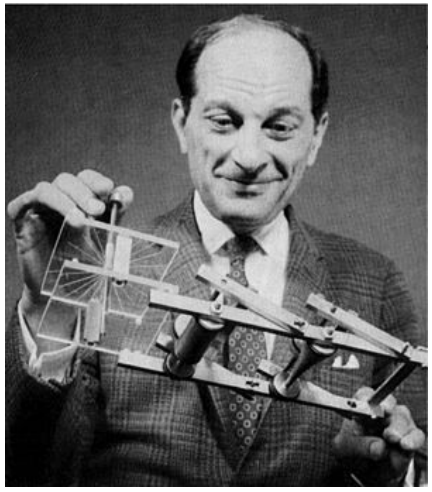
Introduction to Stan for Markov Chain Monte Carlo

Matthew Simpson

Department of Statistics, University of Missouri

March 29, 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 $\boldsymbol{p}' \sim \pi(\boldsymbol{p}|\boldsymbol{q}^{(t)})$.
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

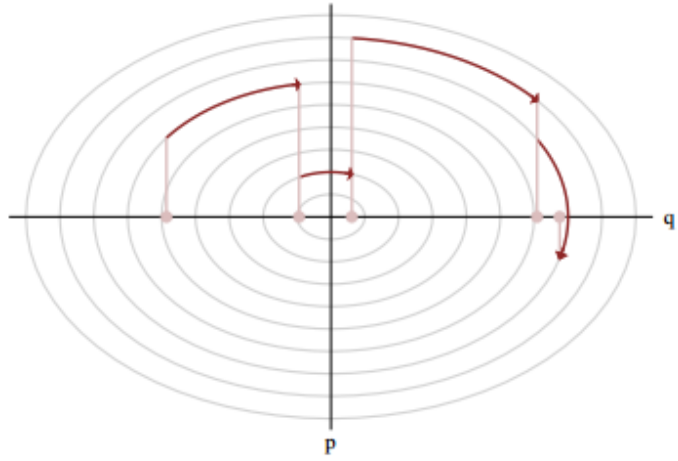


FIG 3. *Every Hamiltonian Markov chain alternatives between a deterministic Hamiltonian flow that explores a single level set (dark red) and a momentum resampling that transitions between level sets with a random walk (light red). The longer the flow is integrated the more efficiently the Markov chain 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 deterministically moves along that set.

Long integration time $\implies$ essentially 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:

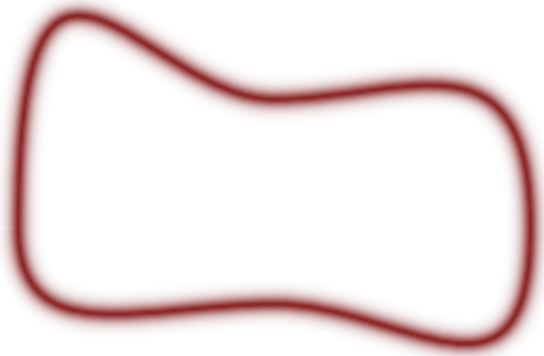


FIG 4. *In high-dimensional parameter spaces probability mass, $\pi(q)\,\mathrm{d}q$, and hence the dominant contributions to expectations, concentrates in a neighborhood called the* typical set. *In order to accurately estimate expectations we have to be able to identify where the typical set lies in 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.

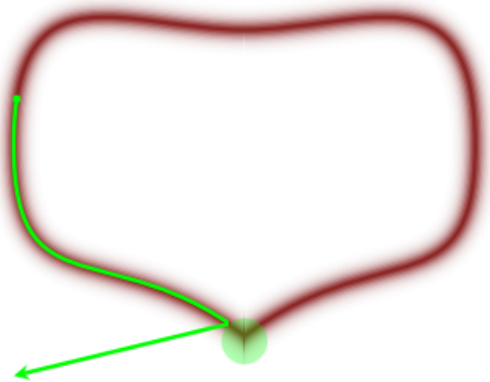


FIG 35. *Neighborhoods of high curvature in the typical set (green) that frustrate geometric ergodicity are also pathological to symplectic integrators, causing them to diverge. This confluence of pathologies is advantageous in practice because we can use the easily-observed divergences to identify the 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: modeling U.S. county level income as a function of covariates.

(See `stanintro.R` for construction of covariates.)

$$y_i = \text{mean household income in county } i,$$
$$\boldsymbol{x}_i' = \text{education and race covariates for county } i.$$

The regression model:

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

for $i = 1, 2, \ldots, N$.

Using Stan: A Simple Example

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

```
data {
  int<lower = 1> n_obs;
  int<lower = 1> n_cov;
  vector[n_obs] y;
  matrix[n_obs, n_cov] x;
}
parameters {
  real alpha;
  vector[n_cov] beta;
  real<lower = 0> sigma;
}
model {
  y ~ normal(alpha + x*beta, sigma);
  alpha ~ normal(60000, 20000);
  beta ~ normal(0, 20000);
  sigma ~ student_t(5, 0, 20000);
}
```

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. Ex: `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_mn;
  real<lower = 0> alpha_prior_sd;
  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.

Example from `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.

Example from `regression.stan`:

```
y ~ normal(alpha + x*beta, sigma);
alpha ~ normal(alpha_prior_mn, alpha_prior_sd);
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 = nrow(codata), n_cov = ncol(x.base),
               y = codata$income.mean, x = x.base,
               beta_prior_mn = 0, beta_prior_sd = 20000,
               alpha_prior_mn = 60000, alpha_prior_sd = 20000,
               sig_prior_scale = 20000, sig_prior_df = 5)

## initialize: create the model and check data constraints
## (takes a good 15-30 seconds)
regfit0 = stan("regression.stan", data = regdat,
                 chains = 1, iter = 1)
## ignore compiler warnings

## fit the model (only need to initialize once)
regfit = stan(fit = regfit0, data = regdat, cores = 4,
              chains = 4, warmup = 2000, iter = 4000)
```

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, w/ `parallel` backend.
  - 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()

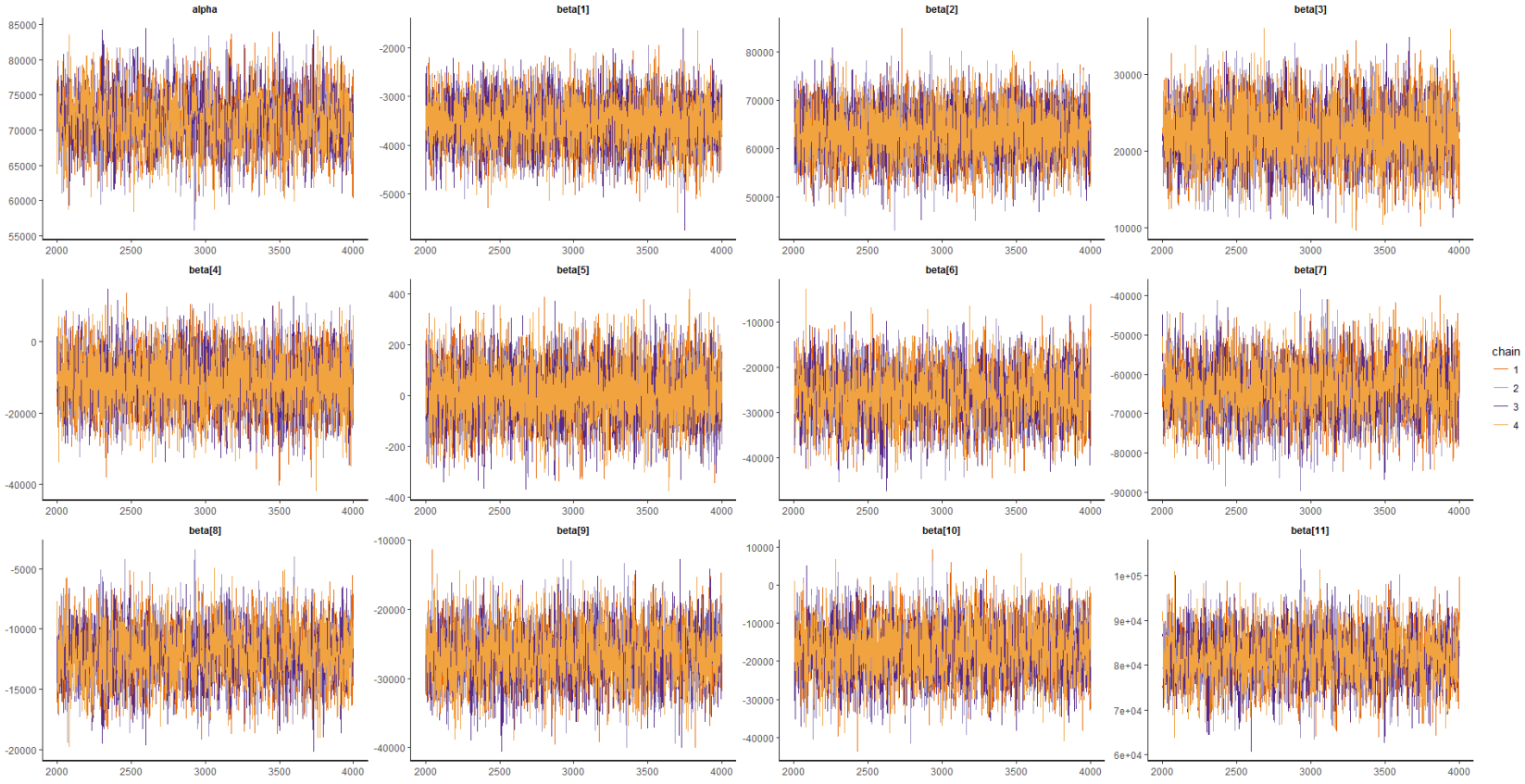
```
> regfit0 = stan("regression.stan", data = regdat,
+                chains = 1, iter = 1)
...
warning: "BOOST_NO_CXX11_RVALUE_REFERENCES" redefined
...
```

- Ignore compiler warnings like the one above.

```
> regfit = stan(fit = regfit0, data = regdat, cores = 4,
+           chains = 4, warmup = 2000, iter = 4000)
...
Chain 1, Iteration: 4000 / 4000 [100%]  (Sampling)
 Elapsed Time: 51.28 seconds (Warm-up)
               38.532 seconds (Sampling)
               89.812 seconds (Total)
```

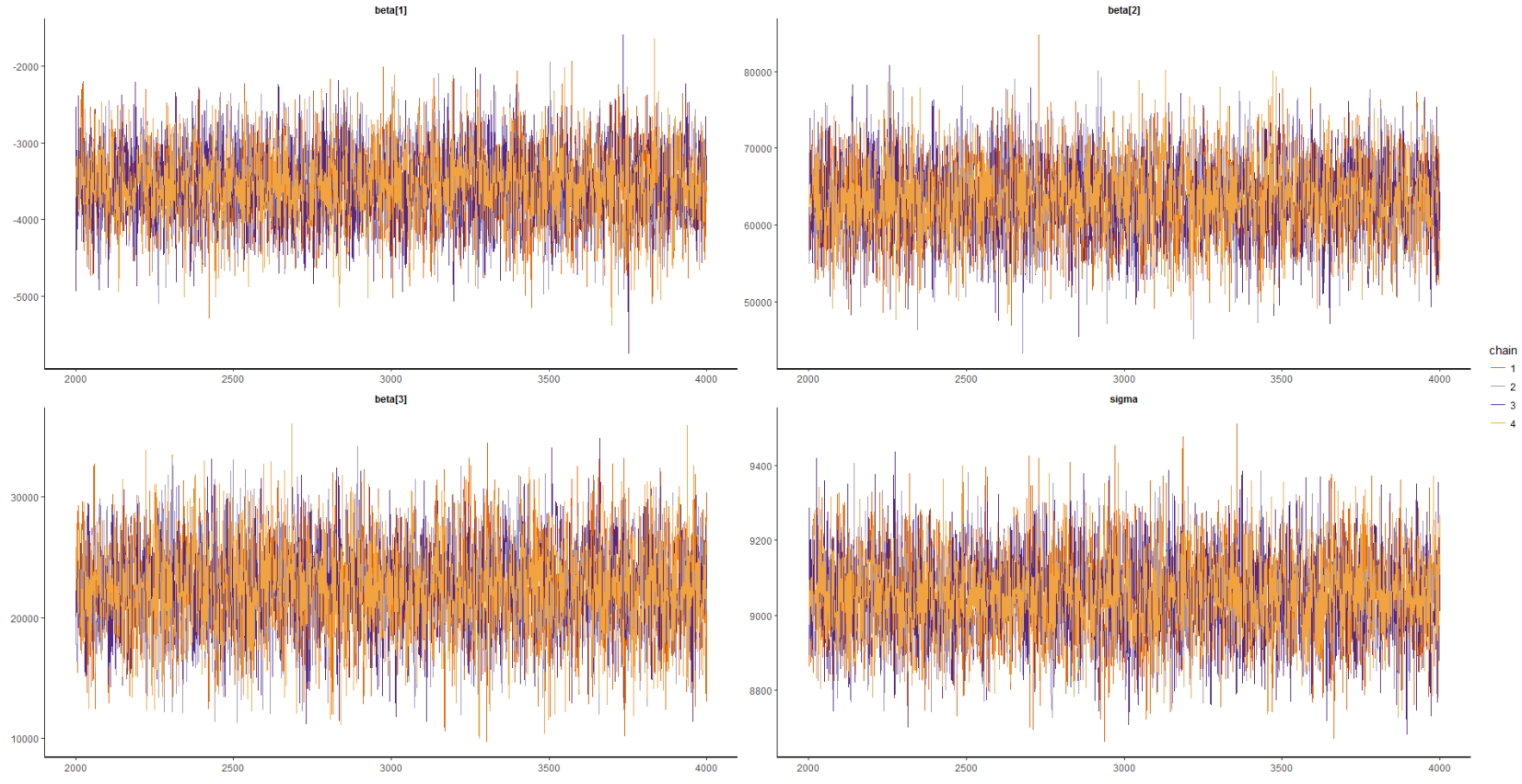
Traceplots

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



More Traceplots

```
> traceplot(regfit,
+   pars = c(paste("beta[", 1:3, "]", sep=""), "sigma"))
```



Posterior Summaries

```
> summary(regfit, pars = c("alpha", "beta", "sigma"))$summary
                mean      se_mean          sd        2.5%          25%
alpha     71399.869821  84.94908998 3938.516308  63689.9678  68746.88233
beta[1]    -3551.058747   6.26461887  489.504511  -4522.1539  -3873.28857
beta[2]    63251.705576  64.16739808 5145.175179  53114.7035  59754.63962
beta[3]    22419.975765  45.58024712 3771.151469  14939.3834  19849.17895
beta[4]   -11424.187414  91.62543117 7604.783988 -26123.4236 -16528.42301
beta[5]        8.222437   1.30817829  111.458389   -212.2165    -67.37313
beta[6]   -26285.741524  96.51512551 5748.283846 -37469.5113 -30101.28221
beta[7]   -63841.417321 120.38712152 6874.102743 -77235.3078 -68536.56421
beta[8]   -12118.477929  44.03565139 2230.542000 -16481.2903 -13615.97501
beta[9]   -26595.696705  69.79832336 3910.493582 -34157.9793 -29214.67677
beta[10] -17089.068001 112.51467271 6941.976014 -30492.6935 -21873.41458
beta[11]  81749.431910 115.67307042 5624.916358  70767.5652  77925.54102
sigma      9049.495580   1.40047729  115.141941   8831.0488   8970.70867
lp__     -30209.057602   0.04393621    2.559298 -30214.9288 -30210.55878
                 50%          75%       97.5%    n_eff      Rhat
alpha     71391.759269  74043.63345  79140.4871 2149.551 1.0008773
beta[1]    -3550.100262  -3217.50867  -2593.4421 6105.540 0.9998866
beta[2]    63251.123712  66745.33228  73248.2005 6429.415 1.0003574
beta[3]    22429.869178  24978.31383  29683.4286 6845.333 0.9996002
beta[4]   -11532.083317  -6253.10872   3440.9387 6888.770 1.0001511
beta[5]        7.641381     83.44036    228.8966 7259.248 1.0005791
beta[6]   -26343.125976 -22457.52170 -15092.0916 3547.200 0.9997183
beta[7]   -63924.700591 -59193.35899 -50219.5088 3260.408 1.0005047
beta[8]   -12103.518865 -10620.56975  -7700.9977 2565.736 1.0007448
beta[9]   -26561.026938 -23967.36268 -18947.1339 3138.869 1.0006779
beta[10] -17026.910436 -12399.28093  -3668.6178 3806.693 1.0002487
beta[11]  81754.468636  85563.39644  92635.6943 2364.654 1.0005060
sigma      9047.818999   9127.68236   9276.6519 6759.506 0.9997143
lp__     -30208.734992 -30207.18270 -30205.0836 3393.100 1.0014618
```

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)] 65076 74078 62241 78395 68850 ...
 $ beta : num [1:8000, 1:11] -2995 -3533 -3492 -3856 -3685 ...
 $ sigma: num [1:8000(1d)] 9174 8926 9124 9074 8982 ...
 $ lp__ : num [1:8000(1d)] -30212 -30208 -30211 -30206 -30210 ...
```

Transformed data and parameters

Transformed data: transform once before running MCMC.
Transformed parameters: transform once per leapfrog step.

```
data { ... }
transformed data {
  vector[n_obs] log_y;
  log_y = log(y);
}
parameters = { ... }
transformed parameters {
  real<lower = 0> sigma;
  sigma = sqrt(sigma2);
}
model { // implicit U(-Inf, Inf) prior on mu
  log_y ~ normal(mu, sigma);
  sigma2 ~ inv_gamma(0.5, 0.5); // this is a bad prior
}
```

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);
  sigma2 ~ inv_gamma(0.5, 0.5); // this is a bad prior
}
```

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 (~).
- 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
}
```

Program Block Characteristics

	data	transformed data	parameters	transformed parameters	model	generated quantities
Execution	Per chain	Per chain	NA	Per leapfrog	Per leapfrog	Per sample
Variable Declarations	Yes	Yes	Yes	Yes	Yes	Yes
Variable Scope	Global	Global	Global	Global	Local	Local
Variables Saved?	No	No	Yes	Yes	No	Yes
Modify Posterior?	No	No	No	No	Yes	No
Random Variables	No	No	No	No	No	Yes

Stolen from

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

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.

Regression with Centered and Scaled Data: Stan

From `regression_cs.stan`. Reduce fit time from 70 to 10 secs.

Note: care taken to get the correct priors for the _cs parameters, and the correct parameter estimates back on original scale.

```
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_mn = (alpha_prior_mn-y_mn)/y_sd;
  alpha_cs_prior_sd = alpha_prior_sd / 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 ~ normal(alpha_cs_prior_mn +
        dot_product(x_mnsd, beta_cs), alpha_cs_prior_sd);
  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;
}
```

Dealing with Hierarchical Models

Example: same regression, now with state-level random random intercepts.

Let $s[i]$ denote the state that county i is in, and α_s denote the intercept for state s.

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

$$y_i|\boldsymbol{\alpha} \overset{ind}{\sim} \mathrm{N}(\alpha_{s[i]} + \boldsymbol{x}_i'\boldsymbol{\beta}, \sigma^2), \qquad \alpha_s \overset{iid}{\sim} \mathrm{N}(\mu_\alpha, \sigma_\alpha^2),$$

for $i = 1, 2, \ldots, N$, $s = 1, 2, \ldots, S$.

Hierarhical Model in Stan

From `rand_intercept_reg.stan`:

```
parameters {
  vector[n_cov] beta;
  real<lower = 0> sigma;
  vector[n_state] alpha;
  real mu_alpha;
  real<lower = 0> sigma_alpha;
}
model {
  y_cs ~ normal(state*alpha + x_cs*beta, sigma);
  alpha ~ normal(mu_alpha, sigma_alpha);
  beta ~ normal(0, 10);
  sigma ~ student_t(5, 0, 10);
  mu_alpha ~ normal(0, 10);
  sigma_alpha ~ student_t(5, 0, 10);
}
```

Slides Left:

1. Results of hierarchical model
   - 1.1 divergent transitions
   - 1.2 exceptions
2. Noncentered hierarchical model
3. results of noncenetered model
4. when is stan good (sorts of models)
   - 4.1 cheap likelihoods
   - 4.2 not-too-large covariance matrices
   - 4.3 not-too-large mixtures