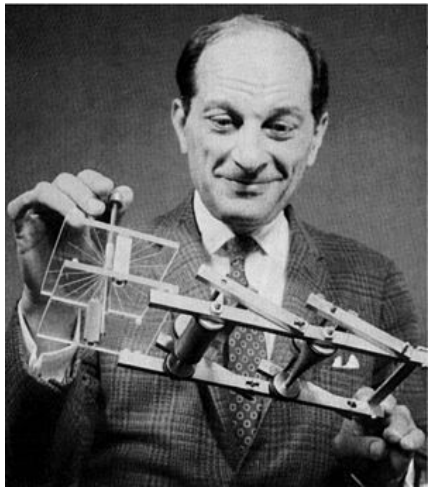
Introduction to Stan for Markov Chain Monte Carlo

Matthew Simpson

Department of Statistics, University of Missouri

March 28, 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 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 $\boldsymbol{p}$ denotes *position* and $\boldsymbol{q}$ 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

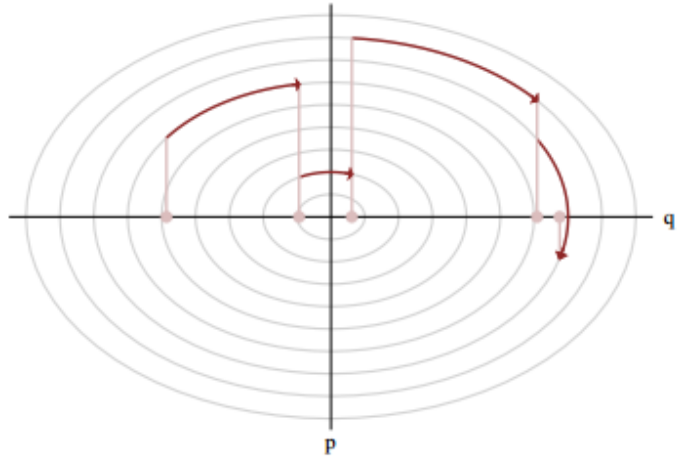


FIG 3. *Every Hamiltonian Markov chain alternatives between a deterministic Hamiltonian flow that explores a single level set (dark red) and a momentum resampling that transitions between level sets with a random walk (light red). The longer the flow is integrated the more efficiently the Markov chain 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 Hamiltonian's equations.
   - Need to make an adjustment to the Hamiltonian flow and use a Metropolis correction to ensure detailed balance.
   - Typically use leapfrog integration $\Longrightarrow$ 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 $f(\boldsymbol{q})\pi(\boldsymbol{q})$ (density) is large.
- Better: where where $f(\boldsymbol{q})\pi(\boldsymbol{q})d\boldsymbol{q}$ (mass) is large.

Typical Set:

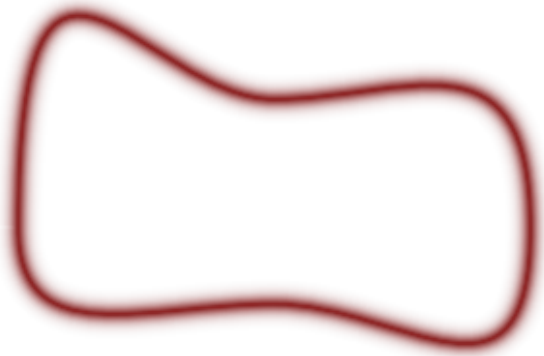


FIG 4. *In high-dimensional parameter spaces probability mass, $\pi(q)\,\mathrm{d}q$, and hence the dominant contributions to expectations, concentrates in a neighborhood called the* typical set. *In order to accurately estimate expectations we have to be able to identify where the typical set lies in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

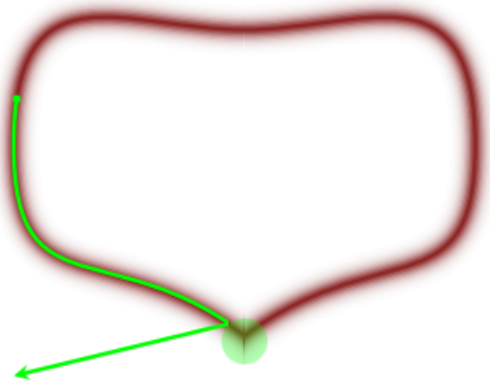


FIG 35. *Neighborhoods of high curvature in the typical set (green) that frustrate geometric ergodicity are also pathological to symplectic integrators, causing them to diverge. This confluence of pathologies is advantageous in practice because we can use the easily-observed divergences to identify the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

Using Stan: A Simple Example

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

```
data {
  int<lower = 1> n_obs;
  int<lower = 1> n_cov;
  vector[n_obs] y;
  matrix[n_obs, n_cov] x;
}
parameters {
  vector[n_cov] beta;
  real<lower = 0> sigma;
}
model {
  y ~ normal(x*beta, sigma);
  beta ~ normal(0, 10);
  sigma ~ student_t(5, 10);
}
```

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 "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.
- 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 (int constraints are inclusive)
real<upper = 0> B; // B <= 0 (real constraints are inclusive
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.

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, including any prior hyperparameters that are not hardcoded.

Example from `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; // assumes iid betas a priori
  real<lower = 0> beta_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 {
  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 constaints 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(x*beta, sigma);
beta ~ normal(beta_prior_mn, beta_prior_sd);
sigma ~ student_t(sig_prior_df, 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, or transformed parameter blocks).

'*' is matrix multiplication.

Sampling statements are vectorized when it makes sense.

Fitting the model in R

Fitting a regression:

```
library(rstan)
...
## create list of all variables in data block
regdat = list(n_obs = n, n_cov = p, y = y, x = x,
              beta_prior_mn = 0, beta_prior_sd = 10,
              sig_prior_scale = 10, sig_prior_df = 5)

## initialize: create the model and check data constraints
## (takes a good 30 seconds)
fit0 = stan("regression.stan", data = regdat,
            chains = 1, iter = 1)

## fit the model (won't need to initialize again)
fit = stan(fit0, data = regdat, cores = 4, chains = 4,
           warmup = 2000, iter = 4000)
```

The stan() function in rstan

Required arguments:

- A model — ".stan" file, or a previous `stan` object.
- Data — list of all variables in the data block of the model.

Useful named arguments and their defaults:

- cores = 1 — number of cores to use (uses `parallel` backend).
- chains = 4 — number of chains.
- iter = 4000 — total number of iterations per chain.
- warmup = iter/2 — number of iterations to use for tuning / burn-in.

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

Transformed data and parameters

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

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

Transformations in the model block

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(.,.); ...
}
```