Hamiltonian Monte Carlo and Stan

- Hamiltonian Monte Carlo uses gradient information and dynamic simulation to reduce random-walk and increase acceptance rate
  - the performance scales well with the number of dimensions
  - this lecture introduces the basic HMC and No-U-Turn-Sampler based dynamic HMC
  - other useful variants have been developed recently

Hamiltonian Monte Carlo and Stan

- Hamiltonian Monte Carlo uses gradient information and dynamic simulation to reduce random-walk and increase acceptance rate
  - the performance scales well with the number of dimensions
  - this lecture introduces the basic HMC and No-U-Turn-Sampler based dynamic HMC
  - other useful variants have been developed recently
- Stan is the most popular probabilistic programming framework
  - many recent probprog frameworks use dynamic HMC samplers
  - this lecture introduces Stan language and main features
  - later you can also use higher level packages built on top of Stan

BDA Chapter 12

- 12.1 Efficient Gibbs samplers (not part of the course)
- 12.2 Efficient Metropolis jump rules (not part of the course)
- 12.3 Further extensions to Gibbs and Metropolis (not part of the course)
- 12.4 Hamiltonian Monte Carlo (important)
- 12.5 Hamiltonian dynamics for a simple hierarchical model (useful example)
- 12.6 Stan: developing a computing environment (useful intro)

Extra material for HMC / NUTS

- An introduction for applied users with good visualizations: Monnahan, Thorson, and Branch (2016) Faster estimation of Bayesian models in ecology using Hamiltonian Monte Carlo. https://dx.doi.org/10.1111/2041-210X.12681
- A technical review of why HMC works: Neal (2012). MCMC using Hamiltonian dynamics. https://arxiv.org/abs/1206.1901
- The No-U-Turn Sampler: Hoffman and Gelman (2014). The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. https://jmlr.csail.mit.edu/papers/v15/hoffman14a.html
- Multinomial variant of NUTS: Betancourt (2018). A Conceptual Introduction to Hamiltonian Monte Carlo. https://arxiv.org/abs/1701.02434

Extra material for Stan

- Gelman, Lee, and Guo (2015) Stan: A probabilistic programming language for Bayesian inference and optimization. http://www.stat.columbia.edu/~gelman/research/published/stan_jebs_2.pdf
- Carpenter et al (2017). Stan: A probabilistic programming language. Journal of Statistical Software 76(1). https://dox.doi.org/10.18637/jss.v076.i01
- Stan User's Guide, Language Reference Manual, and Language Function Reference (in html and pdf) https://mc-stan.org/users/documentation/
    - easiest to start from Example Models in User's guide
- Basics of Bayesian inference and Stan, part 1 Jonah Gabry & Lauren Kennedy (StanCon 2019 Helsinki tutorial)
    - https://www.youtube.com/watch?v=ZRpo41l02KQ&index=6&list=PLuwyh42iHquU4hUBQs20hkBsKSMrp6H0J
    - https://www.youtube.com/watch?v=6cc4N1vT8pk&index=7&list=PLuwyh42iHquU4hUBQs20hkBsKSMrp6H0J

Chapter 12 demos

- demo12_1: HMC
- https://chi-feng.github.io/mcmc-demo/
- http://elevanth.org/blog/2017/11/28/build-a-better-markov-chain/
- cmdstanr_demo, rstan_demo
- http://sumsar.net/blog/2017/01/bayesian-computation-with-stan-and-farmer-jons/
- http://mc-stan.org/documentation/case-studies.html
- https://mc-stan.org/cmdstanr/
- https://mc-stan.org/rstan/

Hamiltonian Monte Carlo

- Originally for quantum-chromo-dynamic simulation (Duane et al., 1987)

Hamiltonian Monte Carlo

- Originally for quantum-chromo-dynamic simulation (Duane et al., 1987)
- Radford Neal started using for Bayesian neural networks in 1990's

Hamiltonian Monte Carlo

- Originally for quantum-chromo-dynamic simulation (Duane et al., 1987)
- Radford Neal started using for Bayesian neural networks in 1990's
- More recent variants are robust wrt the algorithm parameters

Hamiltonian Monte Carlo

- Originally for quantum-chromo-dynamic simulation (Duane et al., 1987)
- Radford Neal started using for Bayesian neural networks in 1990's
- More recent variants are robust wrt the algorithm parameters
- The performance scales well with the number of dimensions

Hamiltonian Monte Carlo

- Originally for quantum-chromo-dynamic simulation (Duane et al., 1987)
- Radford Neal started using for Bayesian neural networks in 1990's
- More recent variants are robust wrt the algorithm parameters
- The performance scales well with the number of dimensions
- The most popular MCMC algorithm in probabilistic programming frameworks (Stan, PyMC, TFP, Pyro, etc.)

Hamiltonian Monte Carlo

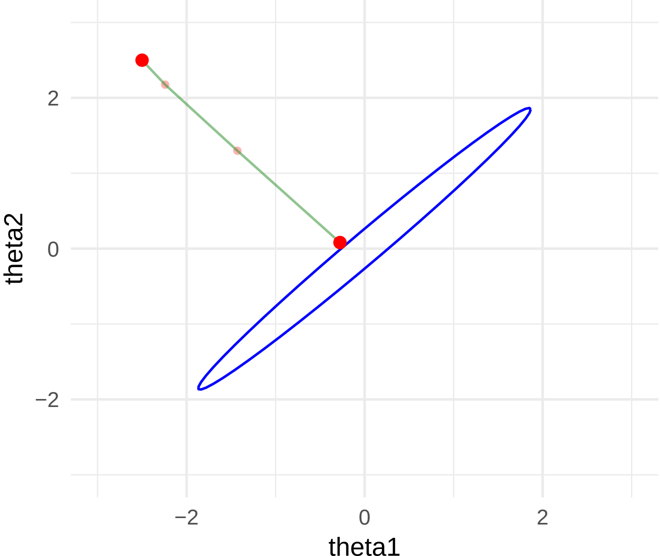
- Originally for quantum-chromo-dynamic simulation (Duane et al., 1987)
- Radford Neal started using for Bayesian neural networks in 1990's
- More recent variants are robust wrt the algorithm parameters
- The performance scales well with the number of dimensions
- The most popular MCMC algorithm in probabilistic programming frameworks (Stan, PyMC, TFP, Pyro, etc.)
- Also used as the a high-fidelity reference in Approximate Inference in Bayesian Deep Learning competition https://izmailovpavel.github.io/neurips_bdl_competition/

Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling

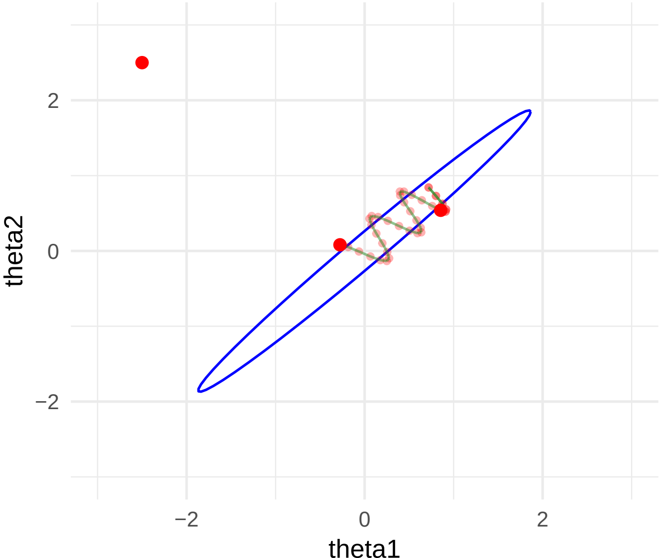
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



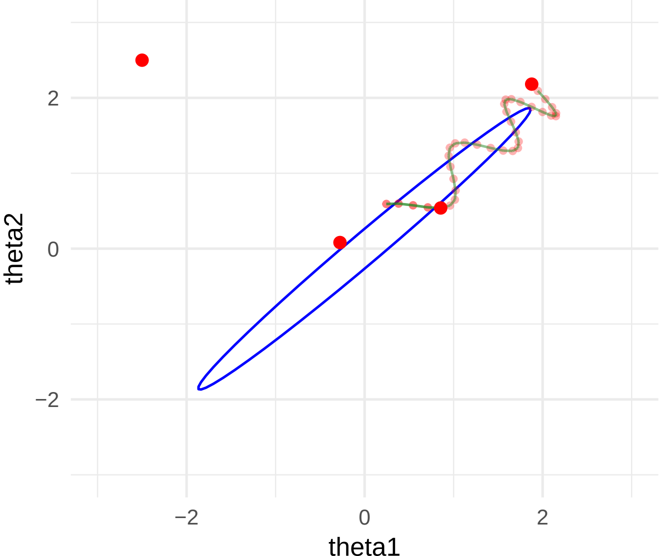
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



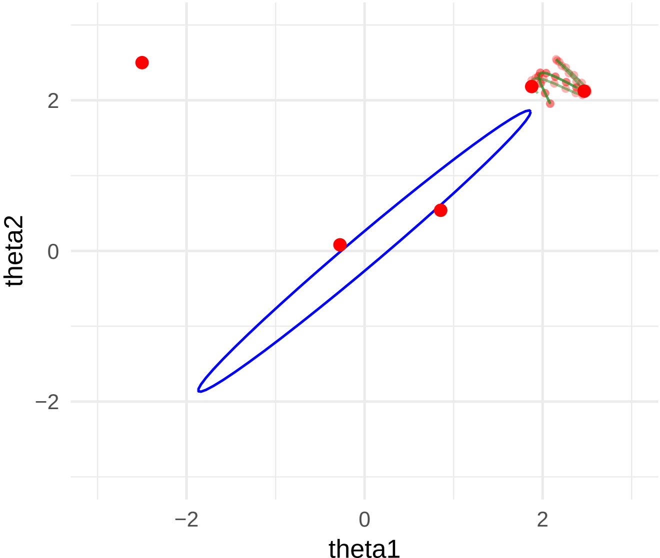
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



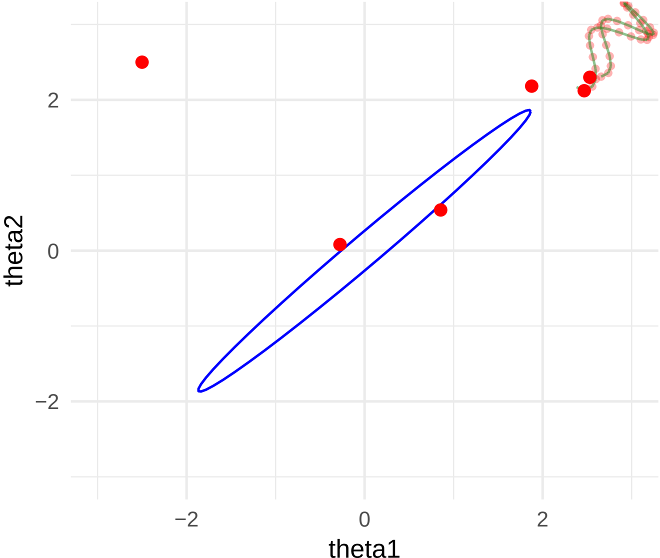
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



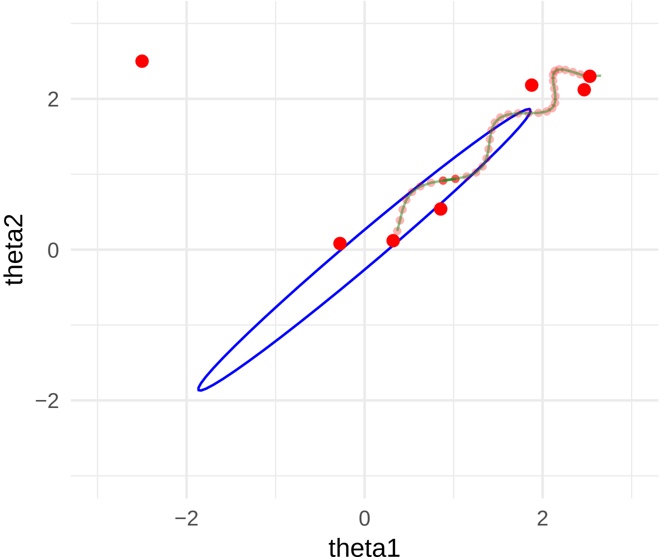
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



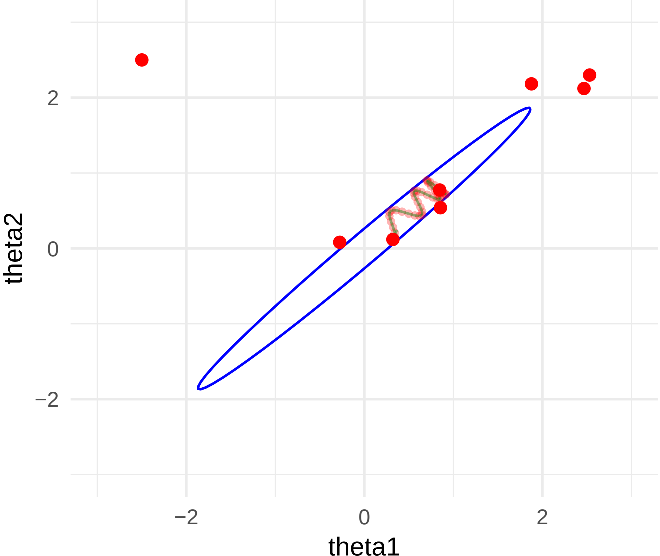
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



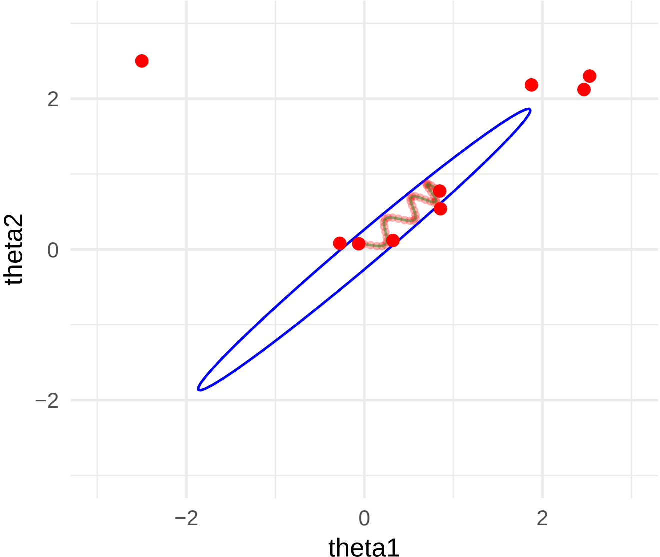
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



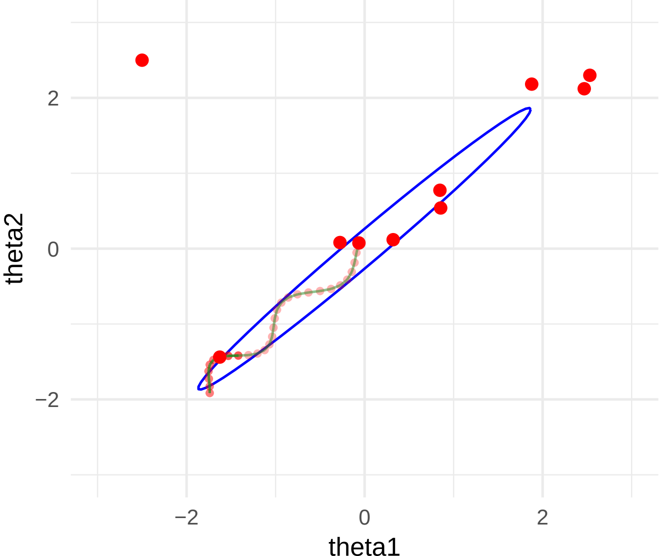
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



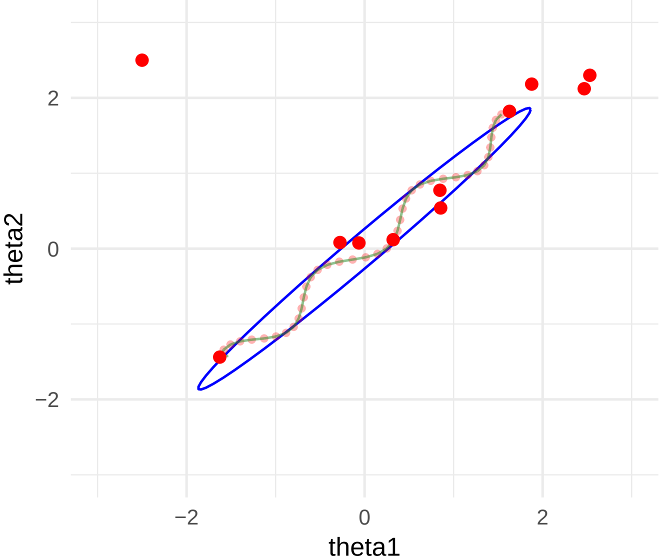
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



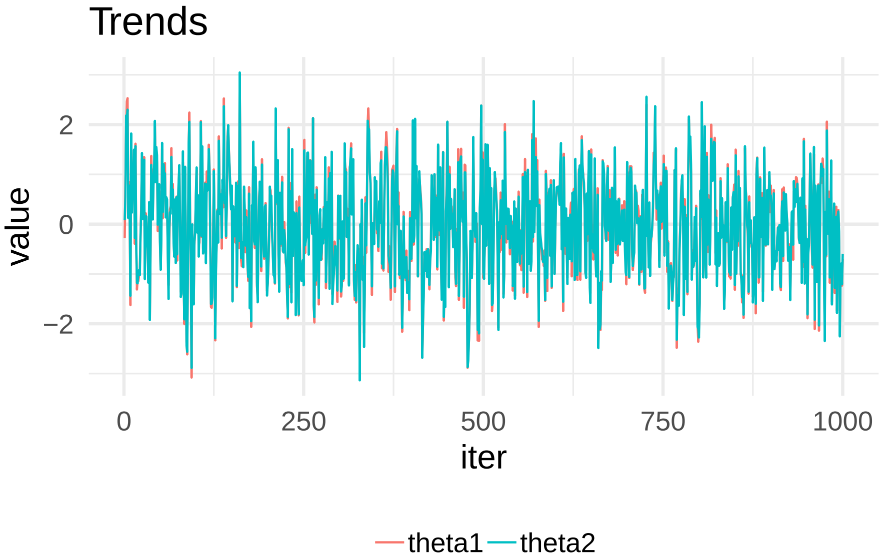
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



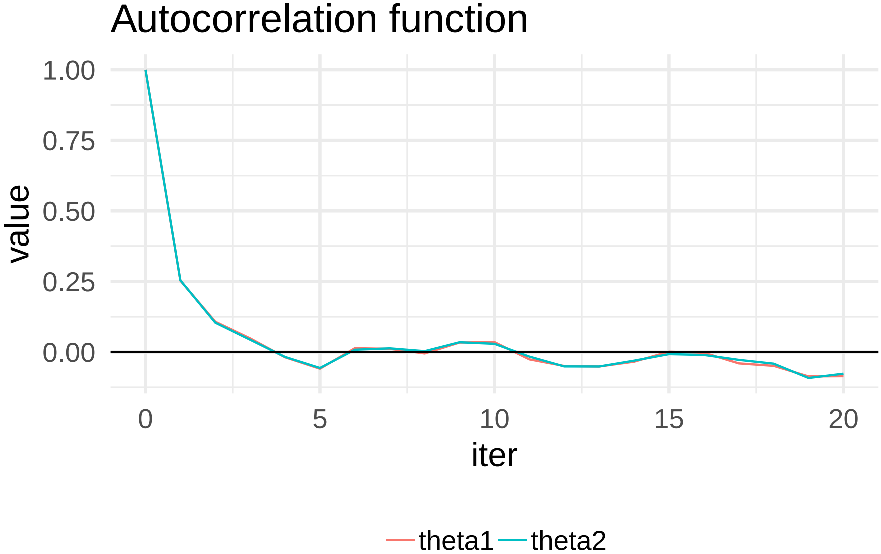
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



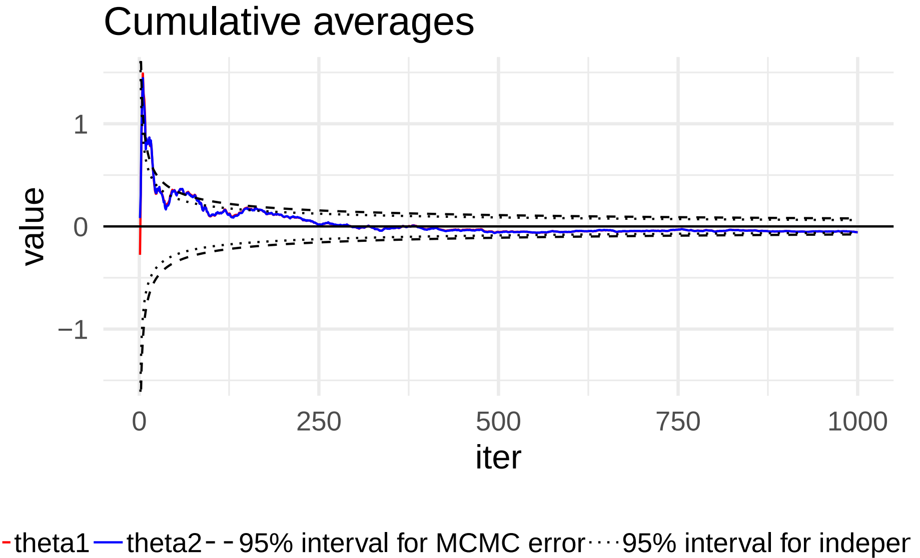
Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



Hamiltonian Monte Carlo

- Uses log density (negative log density is called energy)
- Uses gradient of log density for more efficient sampling



Hamiltonian Monte Carlo / No-U-Turn sampling

1. HMC basics (static HMC)

Hamiltonian Monte Carlo / No-U-Turn sampling

1. HMC basics (static HMC)
2. HMC + leapfrog discretization + Metropolis (static HMC)
    - Duane et al. (1987)

Hamiltonian Monte Carlo / No-U-Turn sampling

1. HMC basics (static HMC)
2. HMC + leapfrog discretization + Metropolis (static HMC)
   - Duane et al. (1987)
3. NUTS + slice sampling + Metropolis (dynamic HMC)
   - Hoffman & Gelman et al. (2014)

Hamiltonian Monte Carlo / No-U-Turn sampling

1. HMC basics (static HMC)
2. HMC + leapfrog discretization + Metropolis (static HMC)
   - Duane et al. (1987)
3. NUTS + slice sampling + Metropolis (dynamic HMC)
   - Hoffman & Gelman et al. (2014)
4. NUTS + multinomial (dynamic HMC)
   - Betancourt (2018)

Hamiltonian Monte Carlo

- Related methods
  - Factorizing $p(\theta_1, \theta_2) = p(\theta_1 \mid \theta_2)p(\theta_2)$: sample from
    1) $p(\theta_2)$,
    2) $p(\theta_1 \mid \theta_2)$

Hamiltonian Monte Carlo

- Related methods
  - Factorizing $p(\theta_1, \theta_2) = p(\theta_1 \mid \theta_2)p(\theta_2)$: sample from
    1) $p(\theta_2)$,
    2) $p(\theta_1 \mid \theta_2)$
  - Metropolis: jointly $p(\theta_1, \theta_2)$
    jump distribution is a combination of proposal distribution and point mass at the previous value

Hamiltonian Monte Carlo

- Related methods
  - Factorizing $p(\theta_1, \theta_2) = p(\theta_1 \mid \theta_2)p(\theta_2)$: sample from
    1) $p(\theta_2)$,
    2) $p(\theta_1 \mid \theta_2)$
  - Metropolis: jointly $p(\theta_1, \theta_2)$
    jump distribution is a combination of proposal distribution and point mass at the previous value
- HMC
  - Augment with ϕ (the same dimensionality as θ)
  - 1) sample directly from $p(\phi)$,
    2) make a special joint Metropolis step for $p(\theta, \phi) = p(\theta)p(\phi)$

Hamiltonian Monte Carlo

1) Sample from $p(\phi)$
   - define $p(\phi) = \mathrm{normal}(0, 1)$
2) Metropolis update for $p(\theta, \phi) = p(\theta)p(\phi)$
   - proposal from Hamiltonian dynamic simulation

Hamiltonian dynamic simulation

- Statistical mechanics and canonical distribution

$$
\begin{aligned}
p(\theta, \phi) &= p(\theta)p(\phi) \\
&= \frac{1}{Z}\exp(-(U(\theta) + K(\phi))) \\
&= \frac{1}{Z}\exp(-H(\theta, \phi))
\end{aligned}
$$

where

- U is potential energy function
- K is kinetic energy function
- H is Hamiltonian energy function
- ϕ is called a momentum variable

Hamiltonian dynamic simulation

- Statistical mechanics and canonical distribution

$$\begin{aligned} p(\theta, \phi) &= p(\theta)p(\phi) \\ &= \frac{1}{Z} \exp(-(U(\theta) + K(\phi))) \\ &= \frac{1}{Z} \exp(-H(\theta, \phi)) \end{aligned}$$

  where

  - U is potential energy function
  - K is kinetic energy function
  - H is Hamiltonian energy function
  - ϕ is called a momentum variable

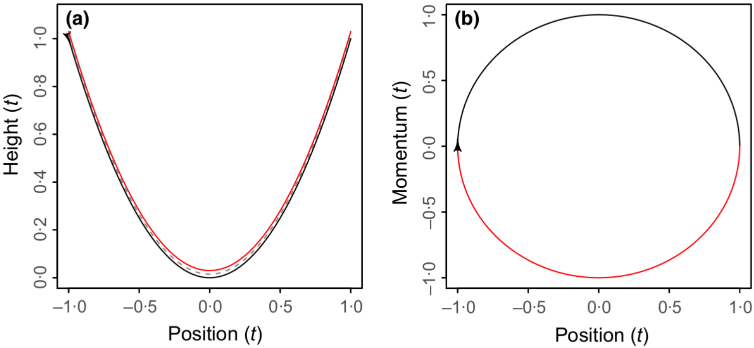
- The potential energy is the negative log density
  $U(\theta) = -\log(p(\theta)) + C$

Hamiltonian dynamic simulation

Equations of motion, use also the gradient

$$\frac{d\theta_i}{dt} = \frac{\partial H}{\partial \phi_i}$$

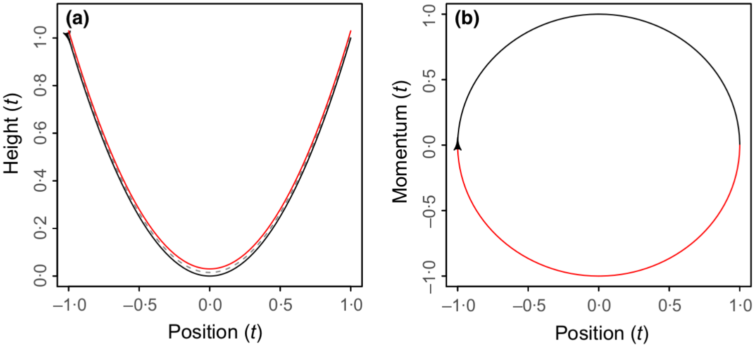
$$\frac{d\phi_i}{dt} = -\frac{\partial H}{\partial \theta_i}$$



From Monnahan et al (2017)

Hamiltonian Monte Carlo

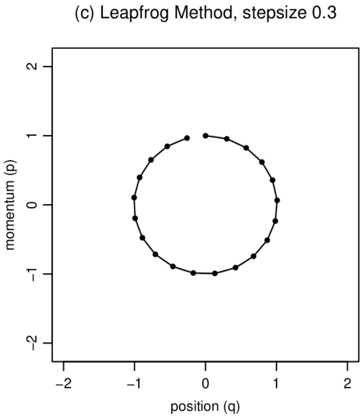
1) Sample from $p(\phi)$
    - define $p(\phi) = \mathrm{normal}(0, 1)$
2) Metropolis update for $p(\theta, \phi) = p(\theta)p(\phi)$
    - proposal from Hamiltonian dynamic simulation
      $p(\theta, \phi) \propto \exp(-H(\theta, \phi))$



From Monnahan et al (2017)

Leapfrog discretization

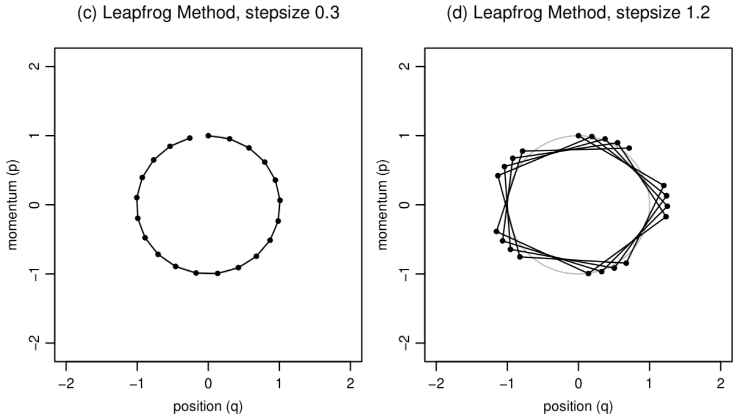
- Leapfrog discretization
  - preserves volume
  - reversible
  - discretization error does not usually grow in time



From Neal (2012)

Leapfrog discretization

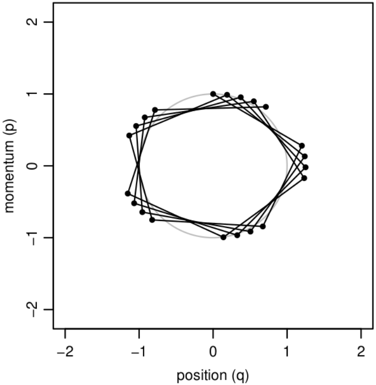
- Leapfrog discretization
  - preserves volume
  - reversible
  - discretization error does not usually grow in time



From Neal (2012)

Leapfrog discretization + Metropolis

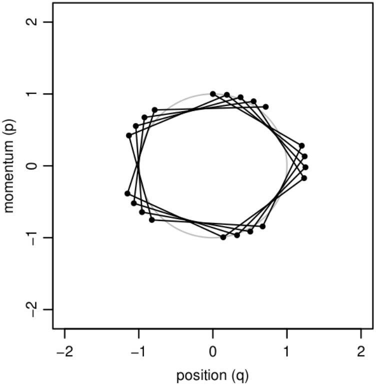
- Leapfrog discretization
  - due to the discretization error the simulation steps away from the constant contour



From Neal (2012)

Leapfrog discretization + Metropolis

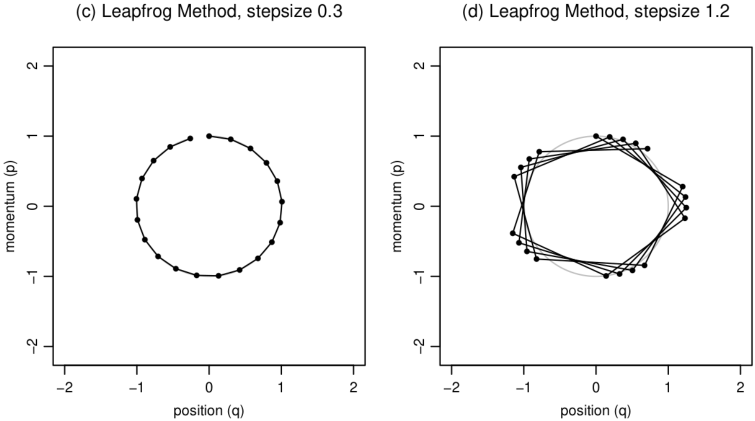
- Leapfrog discretization
  - due to the discretization error the simulation steps away from the constant contour
- Metropolis step with $r = \exp\left(-H(\theta^*, \phi^*) + H(\theta^{(t-1)}, \phi^{(t-1)})\right)$
  - accept if the Hamiltonian energy in the end is higher
  - accept with some probability if the Hamiltonian energy in the end is lower



From Neal (2012)

Two steps of Hamiltonian Monte Carlo

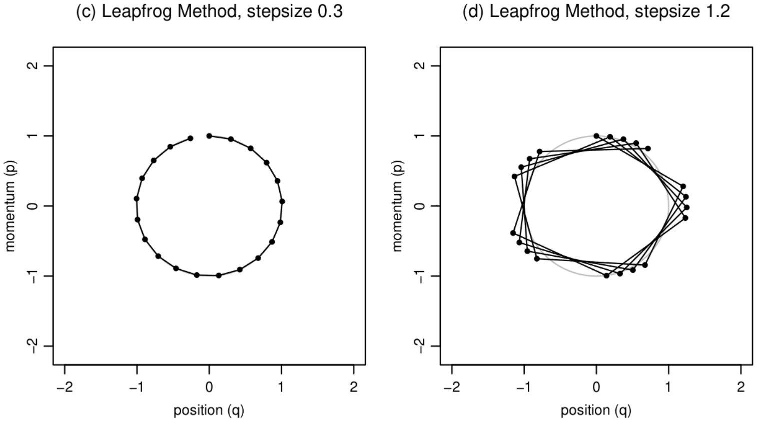
- Perfect simulation keeps $p(\theta, \phi)$ constant



From Neal (2012)

Two steps of Hamiltonian Monte Carlo

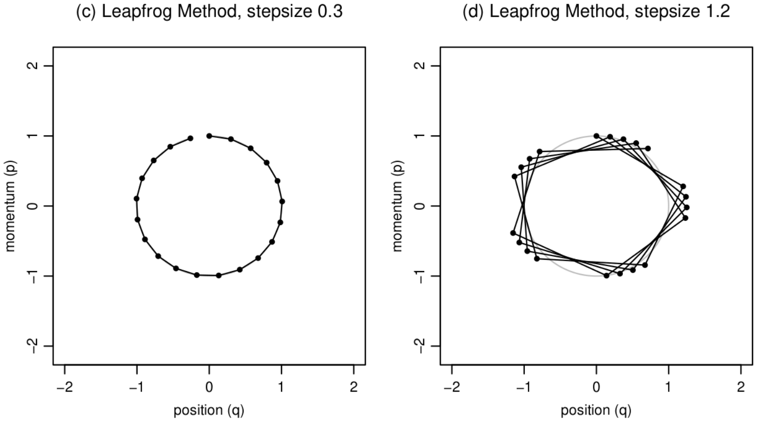
- Perfect simulation keeps $p(\theta, \phi)$ constant
- Discretized simulation keeps changes in $p(\theta, \phi)$ small



From Neal (2012)

Two steps of Hamiltonian Monte Carlo

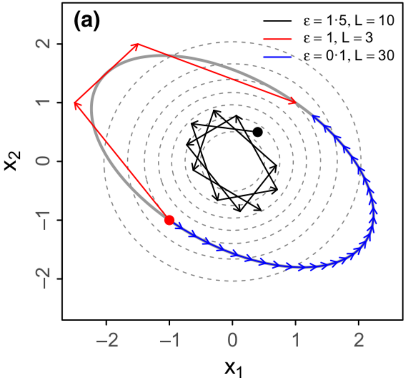
- Perfect simulation keeps $p(\theta, \phi)$ constant
- Discretized simulation keeps changes in $p(\theta, \phi)$ small
- Alternating sampling from $p(\phi)$ is crucial for moving to (θ, ϕ) points with different joint density



From Neal (2012)

Leapfrog discretization, step size

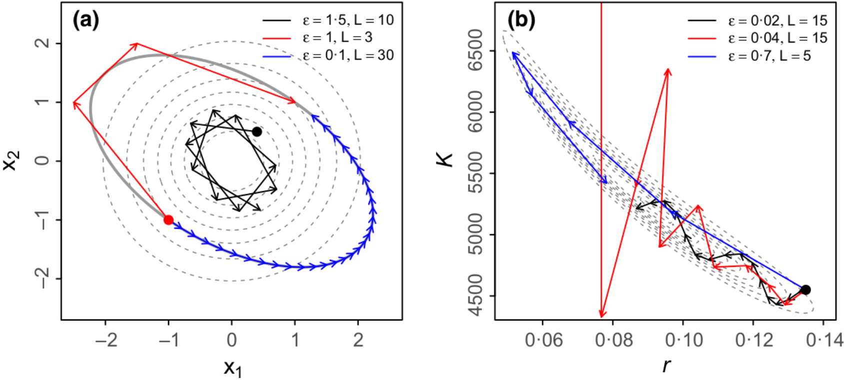
- Small step size → high acceptance rate, but many log density and gradient evaluations
- Big step size → less log density and gradient evaluations, but lower acceptance rate



From Monnahan et al (2017)

Leapfrog discretization, step size

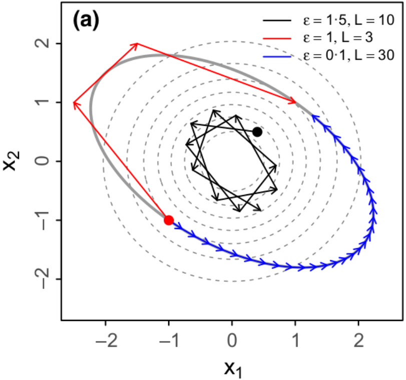
- Small step size → high acceptance rate, but many log density and gradient evaluations
- Big step size → less log density and gradient evaluations, but lower acceptance rate and the simulation may diverge



From Monnahan et al (2017)

Leapfrog discretization, the number of steps

- Many steps can reduce random walk
- Many steps require many log density and gradient evaluations



From Monnahan et al (2017)

Static Hamiltonian Monte Carlo

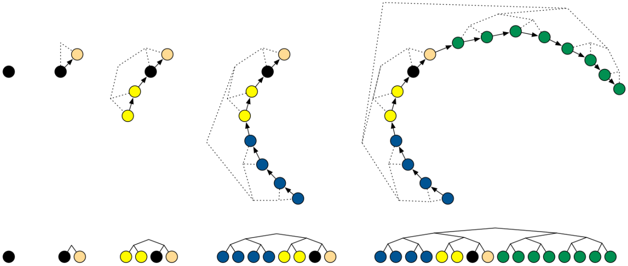
- Fixed number of steps
- Demo https://chi-feng.github.io/mcmc-demo/

No-U-Turn sampler

- Adaptively selects number of steps
    - NUTS is a dynamic HMC algorithm, where dynamic refers to the dynamic trajectory length

No-U-Turn sampler

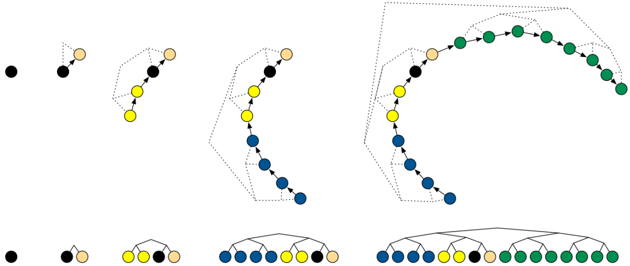
- Adaptively selects number of steps
  - NUTS is a dynamic HMC algorithm, where dynamic refers to the dynamic trajectory length
  - simulate until a U-turn is detected
  - the number of simulation steps doubled if no U-turn yet



from Hoffman & Gelman (2014)

No-U-Turn sampler

- Adaptively selects number of steps
  - NUTS is a dynamic HMC algorithm, where dynamic refers to the dynamic trajectory length
  - simulate until a U-turn is detected
  - the number of simulation steps doubled if no U-turn yet



from Hoffman & Gelman (2014)

- To keep reversibility of Markov chain
  - need to simulate in two directions
  - choose a point along the simulation path with slice sampling
  - Metropolis acceptance step for the selected point

No-U-Turn sampler

- Adaptively selects number of steps
  - NUTS is a dynamic HMC algorithm, where dynamic refers to the dynamic trajectory length
  - simulate until a U-turn is detected
  - the number of simulation steps doubled if no U-turn yet
- To keep reversibility of Markov chain
  - need to simulate in two directions
  - choose a point along the simulation path with slice sampling
  - Metropolis acceptance step for the selected point
- For further efficiency
  - simulation path parts further away from the starting point can have higher probability
  - max treedepth to keep computation in control

No-U-Turn sampler

- Adaptively selects number of steps
    - NUTS is a dynamic HMC algorithm, where dynamic refers to the dynamic trajectory length
    - simulate until a U-turn is detected
    - the number of simulation steps doubled if no U-turn yet
- To keep reversibility of Markov chain
    - need to simulate in two directions
    - choose a point along the simulation path with slice sampling
    - Metropolis acceptance step for the selected point
- For further efficiency
    - simulation path parts further away from the starting point can have higher probability
    - max treedepth to keep computation in control
- Demo https://chi-feng.github.io/mcmc-demo/

No-U-Turn sampler with multinomial sampling

- Original NUTS
  - choose a point along the simulation path with slice sampling
  - possibly with bigger weighting for further points
  - Metropolis acceptance step for the selected point
  - if the proposal is rejected the previous state is also the new state

No-U-Turn sampler with multinomial sampling

- Original NUTS
  - choose a point along the simulation path with slice sampling
  - possibly with bigger weighting for further points
  - Metropolis acceptance step for the selected point
  - if the proposal is rejected the previous state is also the new state
- NUTS with multinomial sampling
  - compute the probability of selecting a point and accepting it for all points
  - select the point with multinomial sampling
  - more likely to accept a point that is not the previous one

No-U-Turn sampler with multinomial sampling

- Original NUTS
  - choose a point along the simulation path with slice sampling
  - possibly with bigger weighting for further points
  - Metropolis acceptance step for the selected point
  - if the proposal is rejected the previous state is also the new state
- NUTS with multinomial sampling
  - compute the probability of selecting a point and accepting it for all points
  - select the point with multinomial sampling
  - more likely to accept a point that is not the previous one
- Demo https://chi-feng.github.io/mcmc-demo/

Mass matrix and the step size adaptation

- Mass matrix refers to having different scaling for different parameters and optionally also rotation to reduce correlations
    - mass matrix is estimated during the adaptation phase of the warm-up
    - mass matrix is estimated using the draws so far

Mass matrix and the step size adaptation

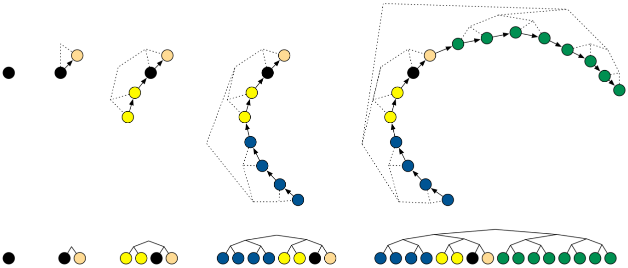
- Mass matrix refers to having different scaling for different parameters and optionally also rotation to reduce correlations
  - mass matrix is estimated during the adaptation phase of the warm-up
  - mass matrix is estimated using the draws so far
- Step size
  - adjusted to be as big as possible while keeping discretization error in control (adapt_delta)
  - “Dual averaging” demo https://chi-feng.github.io/mcmc-demo/

Mass matrix and the step size adaptation

- Mass matrix refers to having different scaling for different parameters and optionally also rotation to reduce correlations
  - mass matrix is estimated during the adaptation phase of the warm-up
  - mass matrix is estimated using the draws so far
- Step size
  - adjusted to be as big as possible while keeping discretization error in control (adapt_delta)
  - “Dual averaging” demo https://chi-feng.github.io/mcmc-demo/
- After adaptation the algorithm parameters are fixed and some more iterations run to finish the warmup

Max tree depth diagnostic

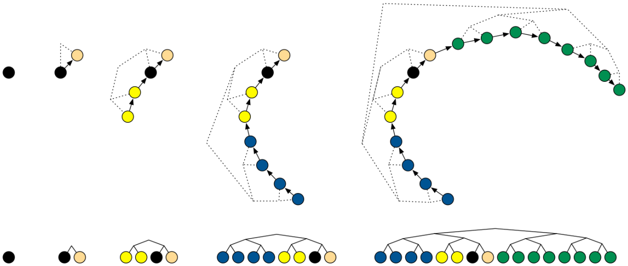
- NUTS specific diagnostic
  - the dynamic simulation is build as a binary tree



from Hoffman & Gelman (2014)

Max tree depth diagnostic

- NUTS specific diagnostic
  - the dynamic simulation is build as a binary tree

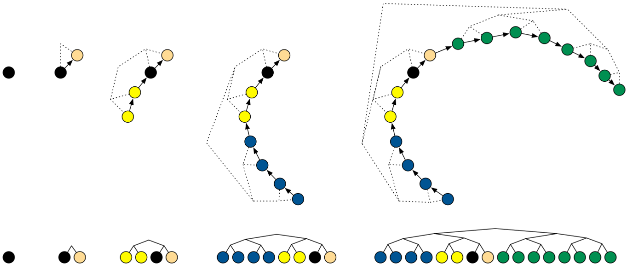


from Hoffman & Gelman (2014)

  - maximum simulation length is capped to avoid very long waiting times in case of bad behavior

Max tree depth diagnostic

- NUTS specific diagnostic
  - the dynamic simulation is build as a binary tree

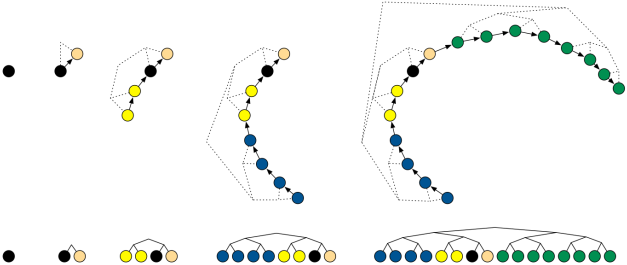


from Hoffman & Gelman (2014)

  - maximum simulation length is capped to avoid very long waiting times in case of bad behavior
- Indicates inefficiency in sampling leading to higher autocorrelations and lower ESS (S_{eff})
  - very low inefficiency can indicate problems that need to be inverse-distance
  - moderate inefficiency doesn't invalidate the result

Max tree depth diagnostic

- NUTS specific diagnostic
  - the dynamic simulation is build as a binary tree

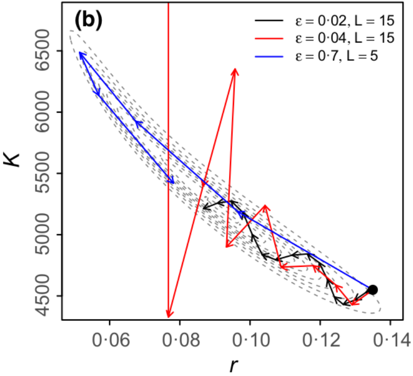


from Hoffman & Gelman (2014)

  - maximum simulation length is capped to avoid very long waiting times in case of bad behavior
- Indicates inefficiency in sampling leading to higher autocorrelations and lower ESS (S_{eff})
  - very low inefficiency can indicate problems that need to be inverse-distance
  - moderate inefficiency doesn't invalidate the result
- Different parameterizations matter

Divergences

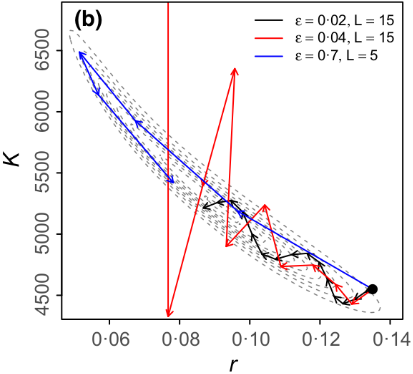
- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density (compared to the used step size)
    - indicates possibility of biased estimates



From Monnahan et al (2017)

Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density (compared to the used step size)
  - indicates possibility of biased estimates



From Monnahan et al (2017)

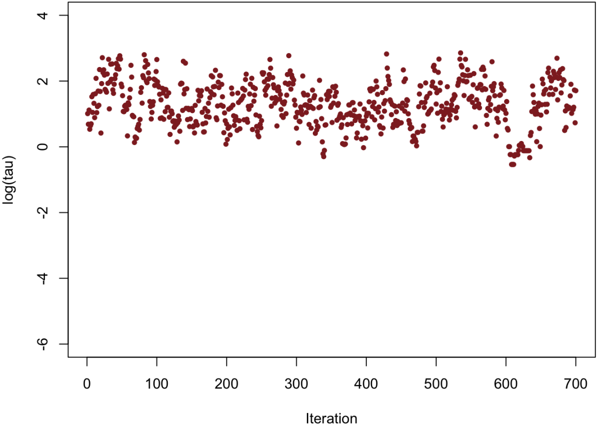
- Demo https://chi-feng.github.io/mcmc-demo/

Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
  - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html

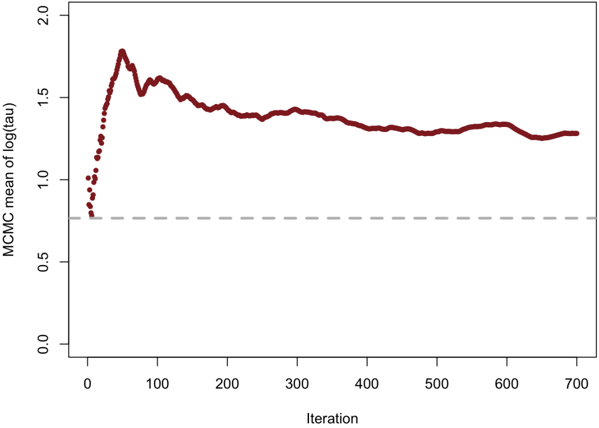
Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
    - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html



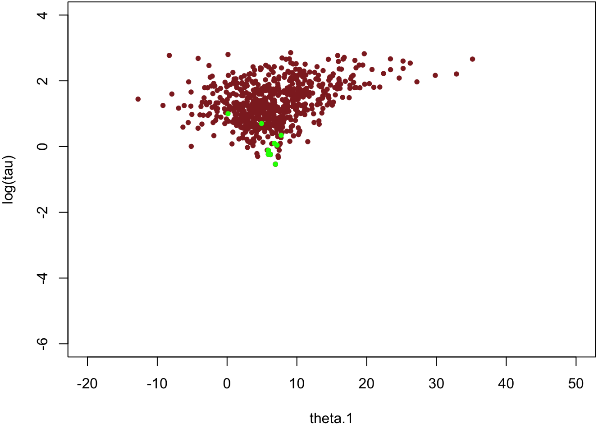
Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
  - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html



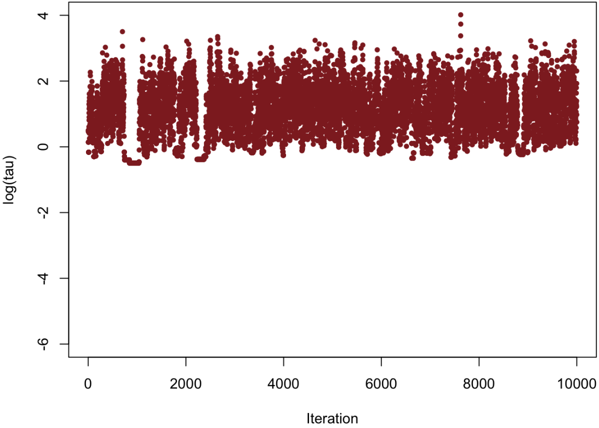
Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
    - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html



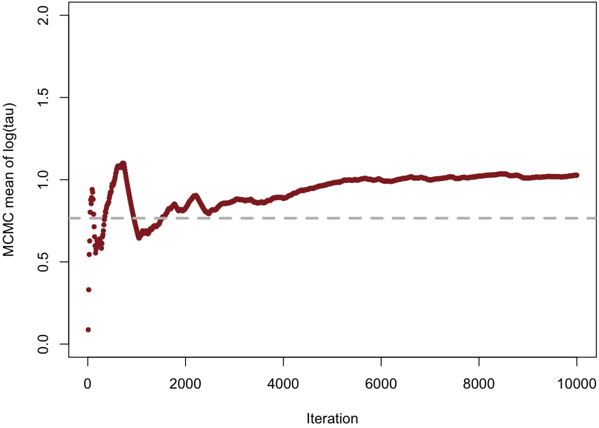
Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
  - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html



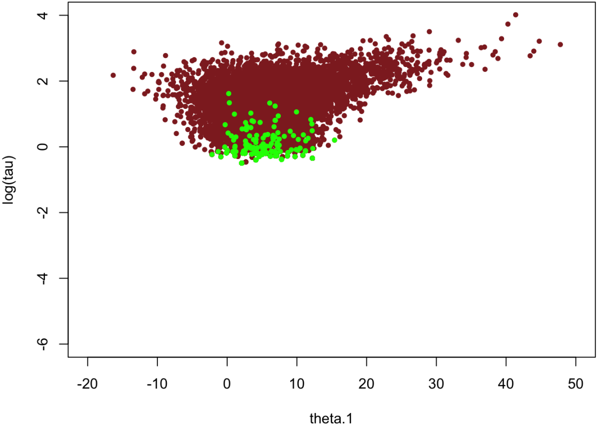
Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
    - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html



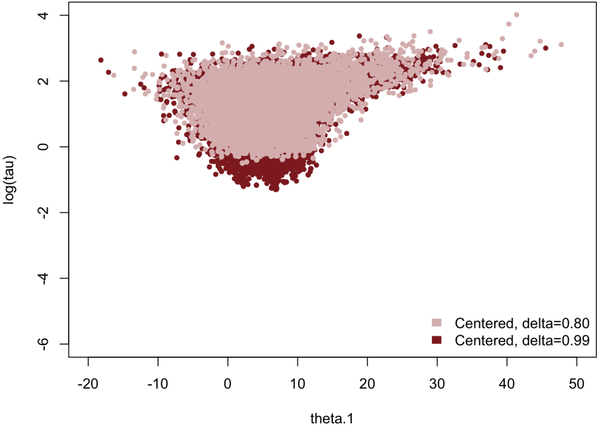
Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
  - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html



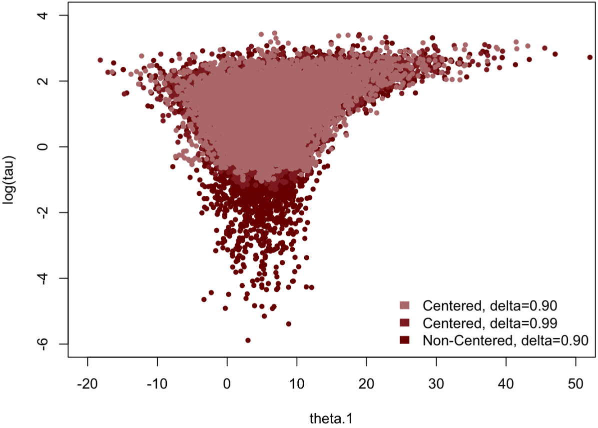
Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
  - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html



Divergences

- HMC specific: indicates that Hamiltonian dynamic simulation has problems with unexpected fast changes in log-density
  - indicates possibility of biased estimates
- http://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html



Problematic distributions

- Nonlinear dependencies
  - simple mass matrix scaling doesn’t help

Problematic distributions

- Nonlinear dependencies
  - simple mass matrix scaling doesn’t help
- Funnels
  - optimal step size depends on location

Problematic distributions

- Nonlinear dependencies
    - simple mass matrix scaling doesn’t help
- Funnels
    - optimal step size depends on location
- Multimodal
    - difficult to move from one mode to another

Problematic distributions

- Nonlinear dependencies
    - simple mass matrix scaling doesn't help
- Funnels
    - optimal step size depends on location
- Multimodal
    - difficult to move from one mode to another
- Long-tailed with non-finite variance and mean
    - efficiency of exploration is reduced
    - central limit theorem doesn't hold for mean and variance

Some other recent HMC and gradient based variants

- ChEES-HMC (Hoffman et al., 2021)
  - a GPU friendly adapted but fixed simulation length
  - static after adaptation
- MEADS (Hoffman & Sountsov, 2022)
  - a GPU friendly multi-chain adaptation for generalized HMC (Horowitz, 1991) in which the momentum is partially updated frequently
  - instead of simulation length, need to choose the partial update rate
- MALT (Riou-Durand and Vogrinc, 2022; Riou-Durand et al., 2022)
  - a GPU friendly method related to GHMC
  - but avoids momentum flips after rejection

Probabilistic programming language

- Wikipedia “A probabilistic programming language (PPL) is a programming language designed to describe probabilistic models and then perform inference in those models”

Probabilistic programming language

- Wikipedia “A probabilistic programming language (PPL) is a programming language designed to describe probabilistic models and then perform inference in those models”
- To make probabilistic programming useful
  - inference has to be as automatic as possible
  - diagnostics for telling if the automatic inference doesn’t work
  - easy workflow (to reduce manual work)
  - fast enough (manual work replaced with automation)

Probabilistic programming

- Enables agile workflow for developing probabilistic models
  - language
  - automated inference
  - diagnostics
- Many frameworks Stan, PyMC, Pyro (Uber), TFP (Google), Turing.jl, JAGS, ELFI, ...
  - Short review of the landscape:
    Štrumbelj et al. (2023). Past, Present, and Future of Software for Bayesian Inference. *Statistical Science*, accepted for publication. Preprint http://www.stat.columbia.edu/~gelman/research/published/Bayesian_software_review-8.pdf.

Stan - probabilistic programming framework

- Language, inference engine, user interfaces, documentation, case studies, diagnostics, packages, ...
    - autodiff to compute gradients of the log density

mc-stan.org

Stan - probabilistic programming framework

- Language, inference engine, user interfaces, documentation, case studies, diagnostics, packages, ...
  - autodiff to compute gradients of the log density
- Most popular, with more than 200K users in social, biological, and physical sciences, medicine, engineering, and business

mc-stan.org

Stan - probabilistic programming framework

- Language, inference engine, user interfaces, documentation, case studies, diagnostics, packages, ...
    - autodiff to compute gradients of the log density
- Most popular, with more than 200K users in social, biological, and physical sciences, medicine, engineering, and business
- Several full time developers, 40+ developers, more than 100 contributors

mc-stan.org

Stan - probabilistic programming framework

- Language, inference engine, user interfaces, documentation, case studies, diagnostics, packages, ...
    - autodiff to compute gradients of the log density
- Most popular, with more than 200K users in social, biological, and physical sciences, medicine, engineering, and business
- Several full time developers, 40+ developers, more than 100 contributors
- R, Python, Julia, Scala, Stata, command line interfaces
- More than 200 R packages using Stan

mc-stan.org

Stan

- Stanislaw Ulam (1909-1984)
    - Monte Carlo method
    - H-Bomb

Binomial model - Stan code

Domain-specific language for constructing models

```
data {
  int<lower=0> N;              // number of experiments
  int<lower=0,upper=N> y;  // number of successes
}

parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}

model {
  theta ~ beta(1,1);          // prior
  y ~ binomial(N,theta);  // observation model / likelihood
}
```

Binomial model - Stan code

Domain-specific language for constructing models

```
data {
  int<lower=0> N;          // number of experiments
  int<lower=0,upper=N> y;  // number of successes
}

parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}

model {
  theta ~ beta(1,1);       // prior
  y ~ binomial(N,theta);   // observation model / likelihood
}
```

Binomial model - Stan code

Domain-specific language for constructing models

```
data {
  int<lower=0> N;              // number of experiments
  int<lower=0,upper=N> y;      // number of successes
}

parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}

model {
  theta ~ beta(1,1);           // prior
  y ~ binomial(N,theta);       // observation model / likelihood
}
```

Binomial model - Stan code

```
data {
  int<lower=0> N;              // number of experiments
  int<lower=0,upper=N> y;  // number of successes
}
```

- Data type and size are declared
- Stan checks that given data matches type and constraints

Binomial model - Stan code

```
data {
  int<lower=0> N;              // number of experiments
  int<lower=0,upper=N> y;   // number of successes
}
```

- Data type and size are declared
- Stan checks that given data matches type and constraints
  - If you are not used to strong typing, this may feel annoying, but it will reduce the probability of coding errors, which will reduce probability of data analysis errors

Binomial model - Stan code

```
parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}
```

- Parameters may have constraints
- Stan makes transformation to unconstrained space and samples in unconstrained space
  - e.g. log transformation for <lower=a>
  - e.g. logit transformation for <lower=a,upper=b>

Binomial model - Stan code

```
parameters {
  real<lower=0,upper=1> theta; // parameter of the binomial
}
```

- Parameters may have constraints
- Stan makes transformation to unconstrained space and samples in unconstrained space
  - e.g. log transformation for <lower=a>
  - e.g. logit transformation for <lower=a,upper=b>
- For these declared transformation Stan automatically takes into account the Jacobian of the transformation (see BDA3 p. 21)

Binomial model - Stan code

```
model {
  theta ~ beta(1,1);        // prior
  y ~ binomial(N,theta);  // observation model / likelihood
}
```

Binomial model - Stan code

```
model {
  theta ~ beta(1,1);        // prior
  y ~ binomial(N,theta);  // observation model / likelihood
}
```

$\sim$ is syntactic sugar and this is equivalent to

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

Binomial model - Stan code

```
model {
  theta ~ beta(1,1);      // prior
  y ~ binomial(N,theta); // observation model / likelihood
}
```

$\sim$ is syntactic sugar and this is equivalent to

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

- `target` is the log posterior density (Lecture 4 discussed log)

Binomial model - Stan code

```
model {
  theta ~ beta(1,1);      // prior
  y ~ binomial(N,theta); // observation model / likelihood
}
```

~ is syntactic sugar and this is equivalent to

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

- `target` is the log posterior density (Lecture 4 discussed log)
- `_lpdf` for continuous, `_lpmf` for discrete distributions (discrete for the left hand side of `|`)

Binomial model - Stan code

```
model {
  theta ~ beta(1,1);      // prior
  y ~ binomial(N,theta); // observation model / likelihood
}
```

∼ is syntactic sugar and this is equivalent to

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

- `target` is the log posterior density (Lecture 4 discussed log)
- `_lpdf` for continuous, `_lpmf` for discrete distributions (discrete for the left hand side of `|`)
- for Stan sampler there is no difference between prior and likelihood, all that matters is the final `target`

Binomial model - Stan code

```
model {
  theta ~ beta(1,1);        // prior
  y ~ binomial(N,theta); // observation model / likelihood
}
```

$\sim$ is syntactic sugar and this is equivalent to

```
model {
  target += beta_lpdf(theta | 1, 1);
  target += binomial_lpmf(y | N, theta);
}
```

- `target` is the log posterior density (Lecture 4 discussed log)
- `_lpdf` for continuous, `_lpmf` for discrete distributions (discrete for the left hand side of `|`)
- for Stan sampler there is no difference between prior and likelihood, all that matters is the final `target`
- you can write in Stan language any program to compute the log density (Stan language is Turing complete)

Stan

- Stan compiles (transplies) the model written in Stan language to C++
    - this makes the sampling for complex models and bigger data faster
    - also makes Stan models easily portable, you can use your own favorite interface and scripting language for manipulating data and inference results (e.g. R, Python, Julia, Stata, ...)

CmdStanR

RStan

```
library(cmdstanr)
options(mc.cores = 1)

d_bin <- list(N = 10, y = 7)
mod_bin <- cmdstan_model(stan_file = 'binom.stan')
fit_bin <- mod_bin$sample(data = d_bin)
```

CmdStanR

RStan

```
library(cmdstanr)
options(mc.cores = 1)

d_bin <- list(N = 10, y = 7)
mod_bin <- cmdstan_model(stan_file = 'binom.stan')
fit_bin <- mod_bin$sample(data = d_bin)
```

PyStan

PyStan

```
import pystan
import stan_utility

data = dict(N=10, y=8)
model = stan_utility.compile_model('binom.stan')
fit = model.sampling(data=data)
```

PyStan

PyStan

```
import pystan
import stan_utility

data = dict(N=10, y=8)
model = stan_utility.compile_model('binom.stan')
fit = model.sampling(data=data)
```

Stan

- Compilation (unless previously compiled model available)
- Warm-up including adaptation
- Sampling
- Generated quantities
- Save posterior draws
- Report divergences, n_{eff}, $\widehat{R}$

Difference between proportions

- An experiment was performed to estimate the effect of beta-blockers on mortality of cardiac patients
- A group of patients were randomly assigned to treatment and control groups:
  - out of 674 patients receiving the control, 39 died
  - out of 680 receiving the treatment, 22 died

Difference between proportions

```
data {
  int<lower=0> N1;
  int<lower=0> y1;
  int<lower=0> N2;
  int<lower=0> y2;
}
parameters {
  real<lower=0,upper=1> theta1;
  real<lower=0,upper=1> theta2;
}
model {
  theta1 ~ beta(1,1);
  theta2 ~ beta(1,1);
  y1 ~ binomial(N1,theta1);
  y2 ~ binomial(N2,theta2);
}

generated quantities {
  real oddsratio;
  oddsratio = (theta2/(1-theta2))/(theta1/(1-theta1));
}
```

Difference between proportions

```
data {
  int<lower=0> N1;
  int<lower=0> y1;
  int<lower=0> N2;
  int<lower=0> y2;
}
parameters {
  real<lower=0,upper=1> theta1;
  real<lower=0,upper=1> theta2;
}
model {
  theta1 ~ beta(1,1);
  theta2 ~ beta(1,1);
  y1 ~ binomial(N1,theta1);
  y2 ~ binomial(N2,theta2);
}

generated quantities {
  real oddsratio;
  oddsratio = (theta2/(1-theta2))/(theta1/(1-theta1));
}
```

Difference between proportions

```
generated quantities {
  real oddsratio;
  oddsratio = (theta2/(1-theta2))/(theta1/(1-theta1));
}
```

- generated quantities is run after the sampling

Difference between proportions

```
d_bin2 <- list(N1 = 674, y1 = 39, N2 = 680, y2 = 22)
mod_bin2 <- cmdstan_model(stan_file = 'binom2.stan')
fit_bin2 <- mod_bin2$sample(data = d_bin2, refresh=1000)
```

```
> Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
...
All 4 chains finished successfully.
Mean chain execution time: 0.0 seconds.
Total execution time: 0.2 seconds.
```

Difference between proportions

```
options(posterior.num_args=list(sigfig=2))
fit_bin2$summary()
```

```
  variable      mean   median     sd      mad        q5      q95  rhat ess_bulk ess_tail
1 lp__       -2.5e+2 -2.5e+2 1.0     0.74    -2.6e+2 -2.5e+2   1.0    1751.    2231.
2 theta1      5.9e-2  5.9e-2 0.0093  0.0093   4.5e-2  7.5e-2   1.0    3189.    2657.
3 theta2      3.4e-2  3.3e-2 0.0069  0.0067   2.3e-2  4.6e-2   1.0    3229.    2163.
4 oddsratio   5.7e-1  5.5e-1 0.16    0.15     3.5e-1  8.7e-1   1.0    2998.    2685.
```

Difference between proportions

```
options(posterior.num_args=list(sigfig=2))
fit_bin2$summary()
```

```
  variable      mean   median      sd     mad        q5      q95  rhat ess_bulk ess_tail
1 lp__       -2.5e+2  -2.5e+2 1.0     0.74    -2.6e+2  -2.5e+2   1.0    1751.    2231.
2 theta1      5.9e-2   5.9e-2 0.0093  0.0093   4.5e-2   7.5e-2   1.0    3189.    2657.
3 theta2      3.4e-2   3.3e-2 0.0069  0.0067   2.3e-2   4.6e-2   1.0    3229.    2163.
4 oddsratio   5.7e-1   5.5e-1 0.16    0.15     3.5e-1   8.7e-1   1.0    2998.    2685.
```

- `lp__` is the log density, ie, same as `target`

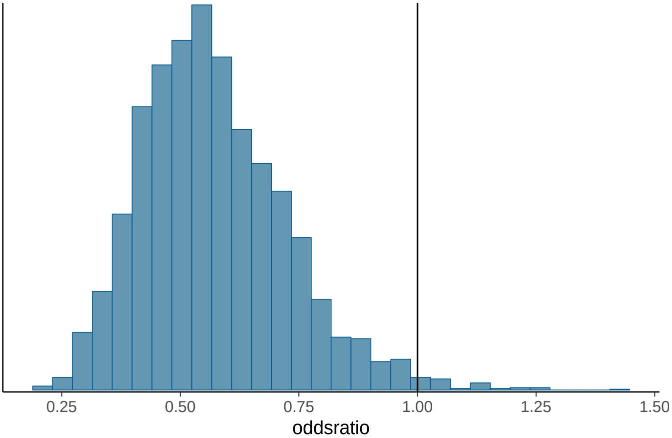
HMC specific diagnostics

```
fit_bin2$diagnostic_summary(diagnostics = c("divergences",
                                             "treedepth"))
```

```
$num_divergent
[1] 0 0 0 0

$num_max_treedepth
[1] 0 0 0 0
```

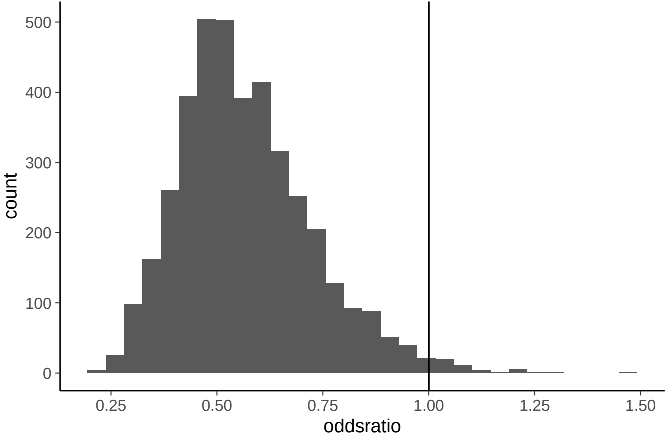
Difference between proportions (bayesplot)

```
draws <- fit_bin2$draws(format = "df")
mcmc_hist(draws, pars = 'oddsratio') +
  geom_vline(xintercept = 1) +
  scale_x_continuous(breaks = c(seq(0.25,1.5,by=0.25)))
```



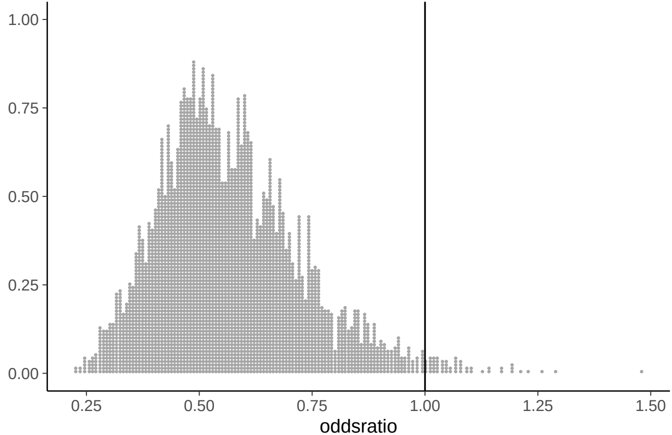
Difference between proportions (ggplot2)

```
draws <- fit_bin2$draws(format = "df")
draws |> ggplot(aes(x=oddsratio)) +
  geom_histogram() +
  geom_vline(xintercept = 1) +
  scale_x_continuous(breaks = c(seq(0.25,1.5,by=0.25)))
```



Difference between proportions (ggdist dot plot)

```
draws <- fit_bin2$draws(format = "df")
draws |> ggplot(aes(x=oddsratio)) +
  geom_dotsinterval() +
  geom_vline(xintercept = 1) +
  scale_x_continuous(breaks = c(seq(0.25,1.5,by=0.25)))
```



Difference between proportions (probability and MCSE)

Probability (and corresponding MCSE) that oddsratio<1

```
> draws |>
    mutate_variables(p_oddsratio_lt_1 =
                       as.numeric(oddsratio<1)) |>
    subset_draws("p_oddsratio_lt_1") |>
    summarise_draws(prob=mean, MCSE=mcse_mean)

  variable            prob    MCSE
  p_oddsratio_lt_1    0.99  0.0023
```

posterior object formats

Default is `draws_array`

```
> fit_bin2$draws()
# A draws_array: 1000 iterations, 4 chains, and 4 variables
, , variable = lp__

         chain
iteration    1    2    3    4
        1 -253 -253 -254 -253
        2 -253 -253 -255 -252
        3 -254 -252 -254 -253
        4 -255 -253 -254 -254
        5 -253 -253 -253 -253

, , variable = theta1

         chain
iteration     1     2     3     4
        1 0.054 0.052 0.045 0.049
        2 0.062 0.060 0.070 0.058
...
```

posterior object formats

`draws_df` looks prettier and works with `ggplot()`

```
> fit_bin2$draws(format ="df")
# A draws_df: 1000 iterations, 4 chains, and 4 variables
   lp__  theta1  theta2  oddsratio
1  -253   0.054   0.033       0.59
2  -253   0.062   0.035       0.55
3  -254   0.047   0.026       0.54
4  -255   0.049   0.049       0.99
5  -253   0.068   0.035       0.50
6  -253   0.056   0.027       0.47
7  -253   0.071   0.031       0.43
8  -253   0.049   0.036       0.72
9  -253   0.049   0.036       0.72
10 -253   0.063   0.026       0.39
# ... with 3990 more draws
# ... hidden reserved variables {'.chain', '.iteration', '.draw'}
```

posterior object formats

`draws_rvar` makes it easy to compute derived quantities

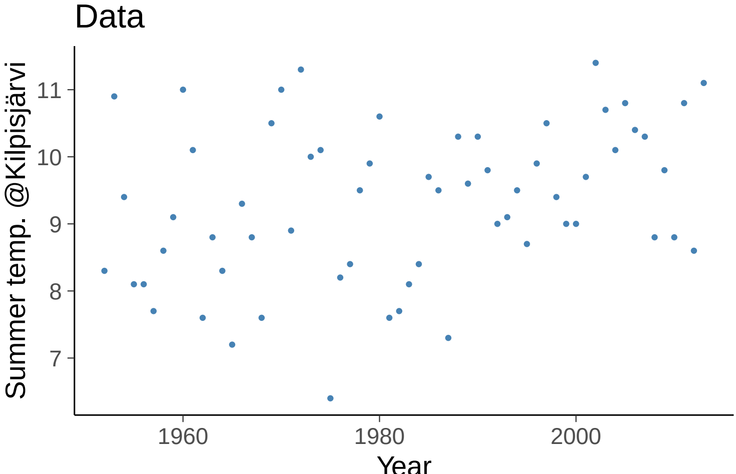
```
> as_draws_rvars(fit_bin2$draws())
# A draws_rvars: 1000 iterations, 4 chains, and 4 variables
$lp__: rvar<1000,4>[1] mean ± sd:
[1] -253 ± 1

$theta1: rvar<1000,4>[1] mean ± sd:
[1] 0.059 ± 0.0093

$theta2: rvar<1000,4>[1] mean ± sd:
[1] 0.034 ± 0.0069

$oddsratio: rvar<1000,4>[1] mean ± sd:
[1] 0.57 ± 0.16

> with(draws, (theta2/(1-theta2))/(theta1/(1-theta1)))
rvar<1000,4>[1] mean ± sd:
[1] 0.5689 ± 0.1577

> draws$oddsratio<1
rvar<1000,4>[1] mean ± sd:
[1] 0.9865 ± 0.1154
```

Shinystan

- Graphical user interface for analysing MCMC results

Kilpisjärvi summer temperature

- Temperature at Kilpisjärvi in June, July and August from 1952 to 2013
- Is there change in the temperature?



Gaussian linear model

```
data {
    int<lower=0> N; // number of data points
    vector[N] x; //
    vector[N] y; //
}
parameters {
    real alpha;
    real beta;
    real<lower=0> sigma;
}
transformed parameters {
    vector[N] mu;
    mu <- alpha + beta*x;
}
model {
    y ~ normal(mu, sigma);
}
```

Gaussian linear model

```
data {
    int<lower=0> N; // number of data points
    vector[N] x; //
    vector[N] y; //
}
```

- difference between `vector[N] x` and `array[N] real x`

Gaussian linear model

```
data {
    int<lower=0> N; // number of data points
    vector[N] x; //
    vector[N] y; //
}
```

- difference between `vector[N] x` and `array[N] real x`
- only integer arrays: `array[N] int x`

Gaussian linear model

```
parameters {
    real alpha;
    real beta;
    real<lower=0> sigma;
}
transformed parameters {
    vector[N] mu;
    mu <- alpha + beta*x;
}
```

- transformed parameters are deterministic transformations of parameters and data

Priors for Gaussian linear model

```
data {
    int<lower=0> N; // number of data points
    vector[N] x; //
    vector[N] y; //
    real pmualpha; // prior mean for alpha
    real psalpha;  // prior std for alpha
    real pmubeta;  // prior mean for beta
    real psbeta;   // prior std for beta
}
...
transformed parameters {
    vector[N] mu;
    mu <- alpha + beta*x;
}
model {
    alpha ~ normal(pmualpha,psalpha);
    beta ~ normal(pmubeta,psbeta);
    y ~ normal(mu, sigma);
}
```

Student-*t* linear model

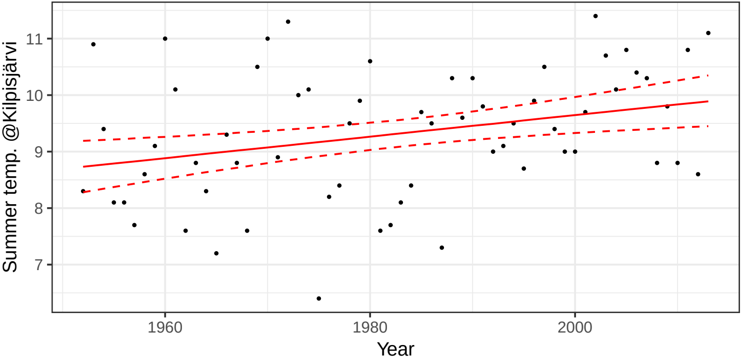
```
...
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
  real<lower=1,upper=80> nu;
}
transformed parameters {
  vector[N] mu;
  mu <- alpha + beta*x;
}
model {
  nu ~ gamma(2,0.1);
  y ~ student_t(nu, mu, sigma);
}
```

Priors

- Prior for temperature increase?

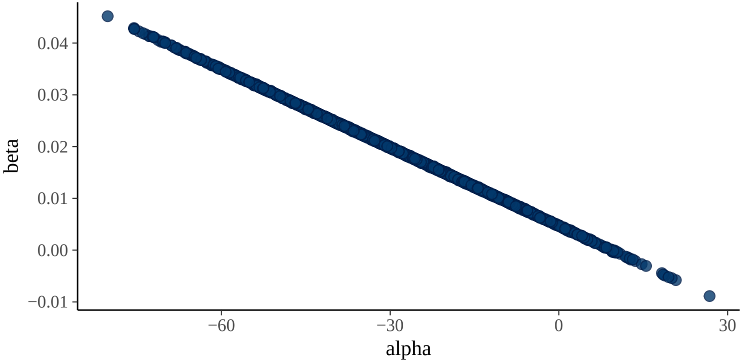
Kilpisjärvi summer temperature

Posterior fit



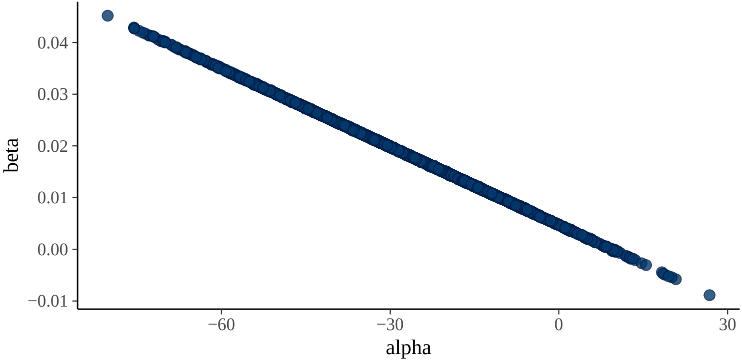
Kilpisjärvi summer temperature

Posterior draws of alpha and beta



Kilpisjärvi summer temperature

Posterior draws of alpha and beta



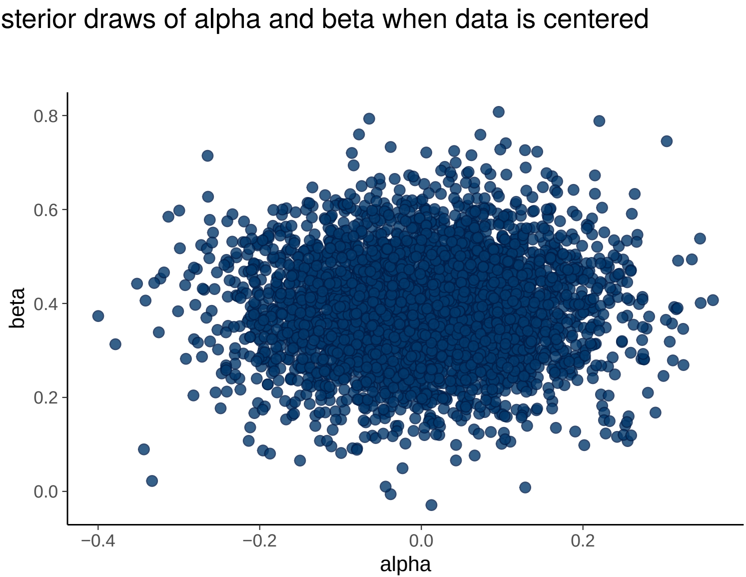
```
Warning: 1 of 4000 (0.0%) transitions hit the maximum treedepth limit of 1
See https://mc-stan.org/misc/warnings for details.
```

Linear regression model in Stan

Center the data inside the model code

```
data {
  int<lower=0> N; // number of data points
  vector[N] x; //
  vector[N] y; //
  real xpred; // input location for prediction
}
transformed data {
  vector[N] x_std;
  vector[N] y_std;
  real xpred_std;
  x_std = (x - mean(x)) / sd(x);
  y_std = (y - mean(y)) / sd(y);
  xpred_std = (xpred - mean(x)) / sd(x);
}
```

Kilpisjärvi summer temperature



Kilpisjärvi summer temperature

Without centering

```
> fit_lin$summary(variables=c("alpha","beta"),
                  default_convergence_measures())
  variable      rhat  ess_bulk  ess_tail
  alpha          1.0      919.      897.
  beta           1.0      919.      895.
```

With centering

```
> fit_lin_std$summary(variables=c("alpha","beta"),
                      default_convergence_measures())
  variable      rhat  ess_bulk  ess_tail
  alpha          1.0     3872.     2616.
  beta           1.0     3770.     2396.
```

RStanARM

- `RStanARM` provides simplified model description with pre-compiled models
  - no need to wait for compilation
  - a restricted set of models

Two group Binomial model:

```
d_bin2 <- data.frame(N = c(674, 680), y = c(39,22), grp2 = c(0,1))
fit_bin2 <- stan_glm(y/N ~ grp2, family = binomial(), data = d_bin2,
                     weights = N)
```

RStanARM

- `RStanARM` provides simplified model description with pre-compiled models
  - no need to wait for compilation
  - a restricted set of models

Two group Binomial model:

```
d_bin2 <- data.frame(N = c(674, 680), y = c(39,22), grp2 = c(0,1))
fit_bin2 <- stan_glm(y/N ~ grp2, family = binomial(), data = d_bin2,
                     weights = N)
```

Gaussian linear model

```
fit_lin <- stan_glm(temp ~ year, data = d_lin)
```

brms

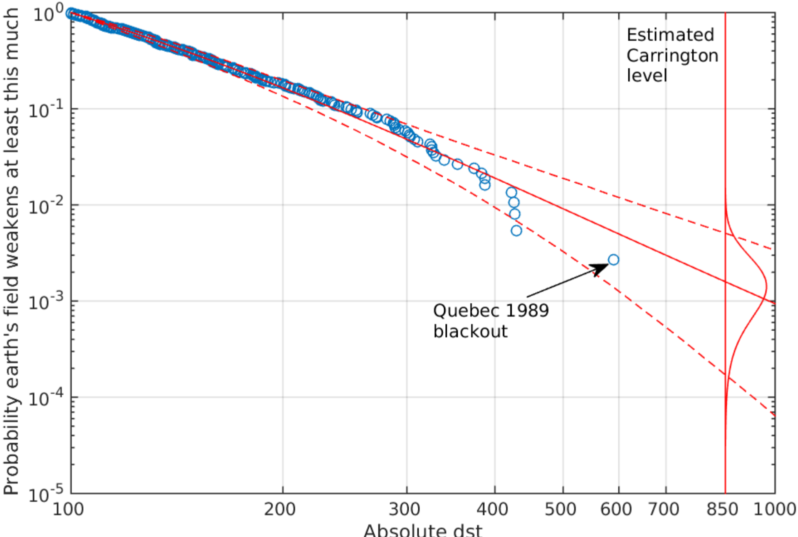
- `brms` provides simplified model description
  - \+ a larger set of models than `RStanARM`, but still restricted
  - \- need to wait for the compilation

```
fit_bin2 <- brm(y | trials(N) ~ grp2, family = binomial(), data = d_bin2)
```

```
fit_lin_t <- brm(temp ~ year, data = d_lin, family = student())
```

Extreme value analysis

Geomagnetic storms



Extreme value analysis

```
data {
  int<lower=0> N;
  vector<lower=0>[N] y;
  int<lower=0> Nt;
  vector<lower=0>[Nt] yt;
}
transformed data {
  real ymax;
  ymax <- max(y);
}
parameters {
  real<lower=0> sigma;
  real<lower=-sigma/ymax> k;
}
model {
  y ~ gpareto(k, sigma);
}
generated quantities {
  vector[Nt] predccdf;
  predccdf<-gpareto_ccdf(yt,k,sigma);
}
```

User defined functions

```
functions {
  real gpareto_lpdf(vector y, real k, real sigma) {
    // generalised Pareto log pdf with mu=0
    // should check and give error if k<0
    // and max(y)/sigma > -1/k
    int N;
    N <- dims(y)[1];
    if (abs(k) > 1e-15)
      return -(1+1/k)*sum(log1pv(y*k/sigma)) -N*log(sigma);
    else
      return -sum(y/sigma) -N*log(sigma); // limit k->0
  }
  vector gpareto_ccdf(vector y, real k, real sigma) {
    // generalised Pareto log ccdf with mu=0
    // should check and give error if k<0
    // and max(y)/sigma < -1/k
    if (abs(k) > 1e-15)
      return exp((-1/k)*log1pv(y/sigma*k));
    else
      return exp(-y/sigma); // limit k->0
  }
}
```

Different interfaces

- `CmdStanR` / `CmdStanPy`
  - Interface on top of command-line program CmdStan
- `RStan` / `PyStan`
  - C++ functions of Stan are called directly from R / Python
  - Higher integration between R/Python and Stan, but maybe more difficult to install due to more requirements of compatible C++ compilers and libraries

Other packages

- R
    - `posterior` — posterior handling and diagnostics (Lecture 5 and 6)
    - `shinystan` — interactive diagnostics
    - `bayesplot` — visualization and model checking (Lectures 5, 6, and 8)
    - `tidybayes` and `ggdist` – more posterior and prediction visualization (Lecture 6)
    - `loo` — cross-validation model assessment and comparison (Lecture 9)
    - `projpred` — projection predictive variable selection (Lecture 12)
    - `priorsense` — prior and likelihood sensitivity diagnostics (Lecture 12)

- Python
    - `ArviZ` — visualization, and model checking and assessment