Lecture 6: Hamiltonian Monte Carlo and how to speak Stan

Ben Lambert[1]

ben.lambert@some.ox.ac.uk

[1]Somerville College
University of Oxford

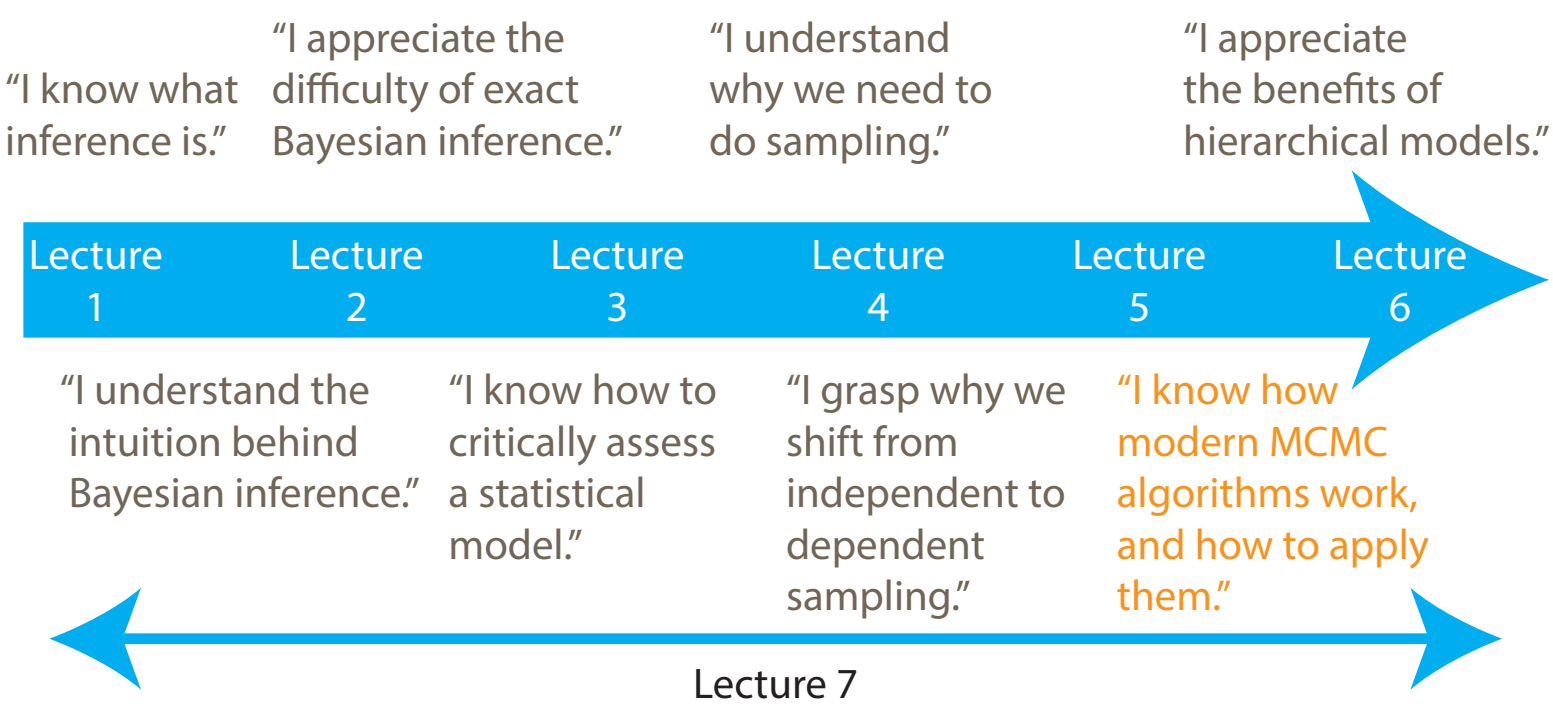
Wednesday 30th November, 2016

Lecture outcomes

By the end of this lecture you should:

1. Grasp the intuition behind Hamiltonian Monte Carlo, and the benefits of this algorithm.
2. Know how to start coding up a model in Stan.
3. Appreciate how easy Stan makes things for us compared to coding up the algorithm ourselves.
4. Know what to do when coding goes wrong.
5. Know what to do when sampling goes wrong.

Overall course outline



Effective sample size

Question: How do we compare the performance of two sampling algorithms?

Answer: Estimate the number of effective samples per X iterations,
"The **effective sample size** for X iterations is the equivalent number of samples from an independent sampler."

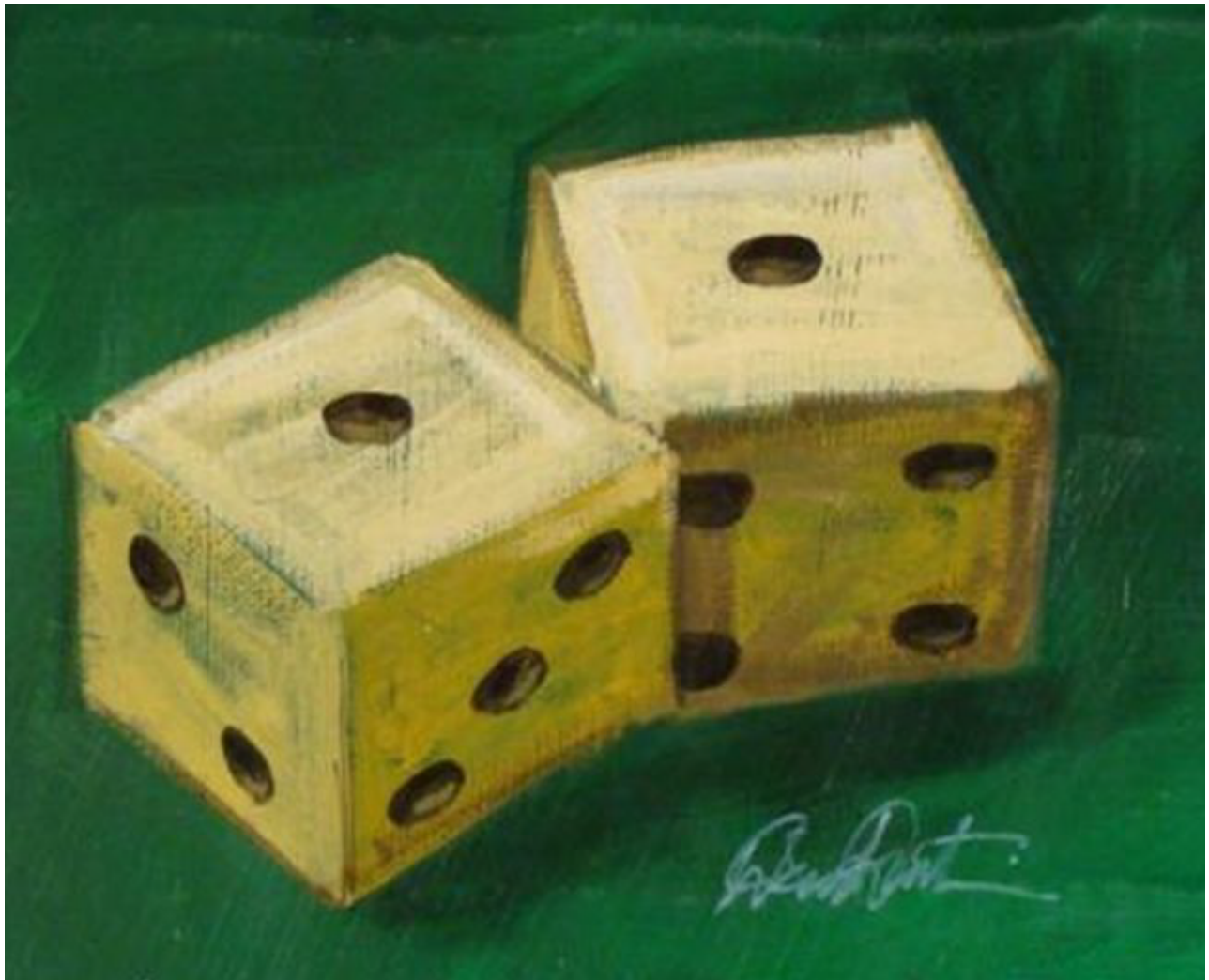
Effective sample size

Question: But how do we estimate the effective sample size?

Answer: Design a metric so that as dependence $\to 0 \implies$ effective sample size $\to$ actual sample size. But why does dependence matter?

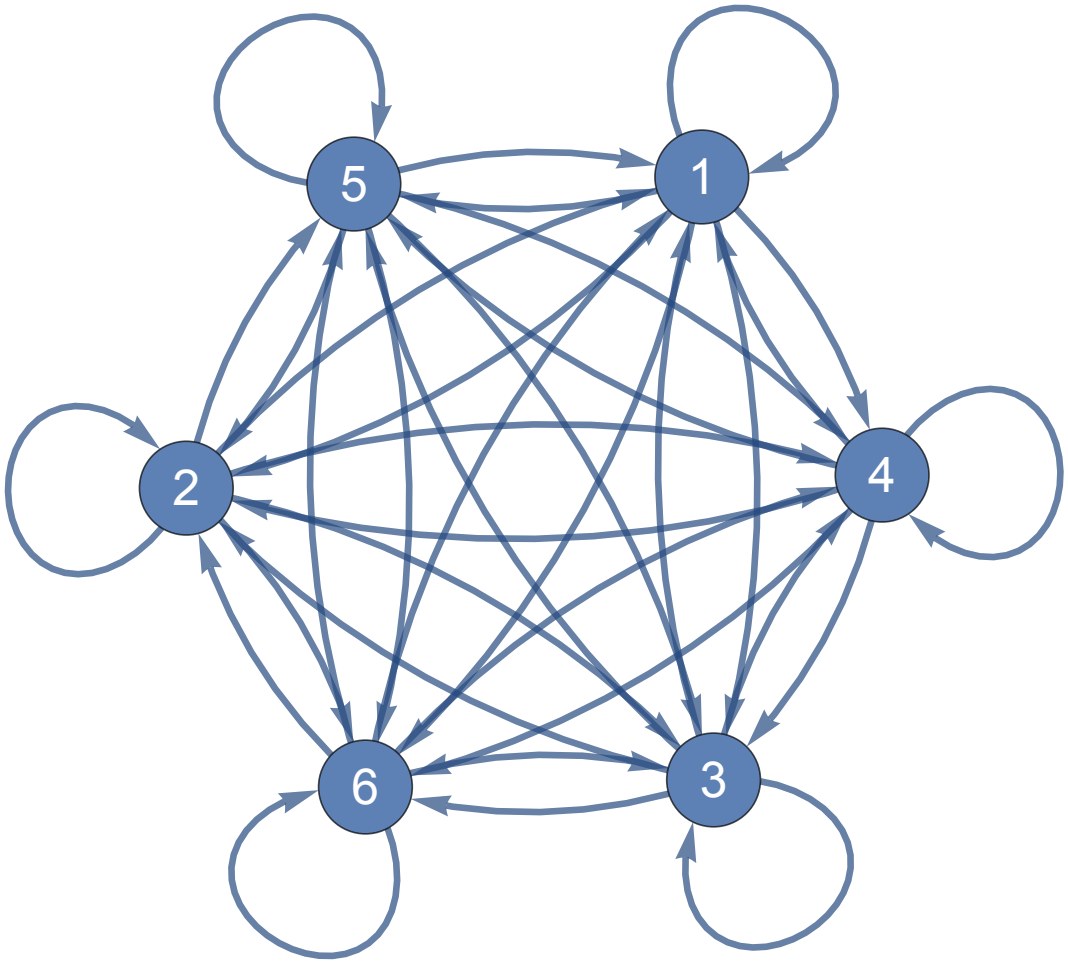
Introducing a Markovian die

A standard die has six faces, all of which are equally likely to be obtained on a given throw.



Introducing a Markovian die

This can be represented in graphical form,



Introducing a Markovian die

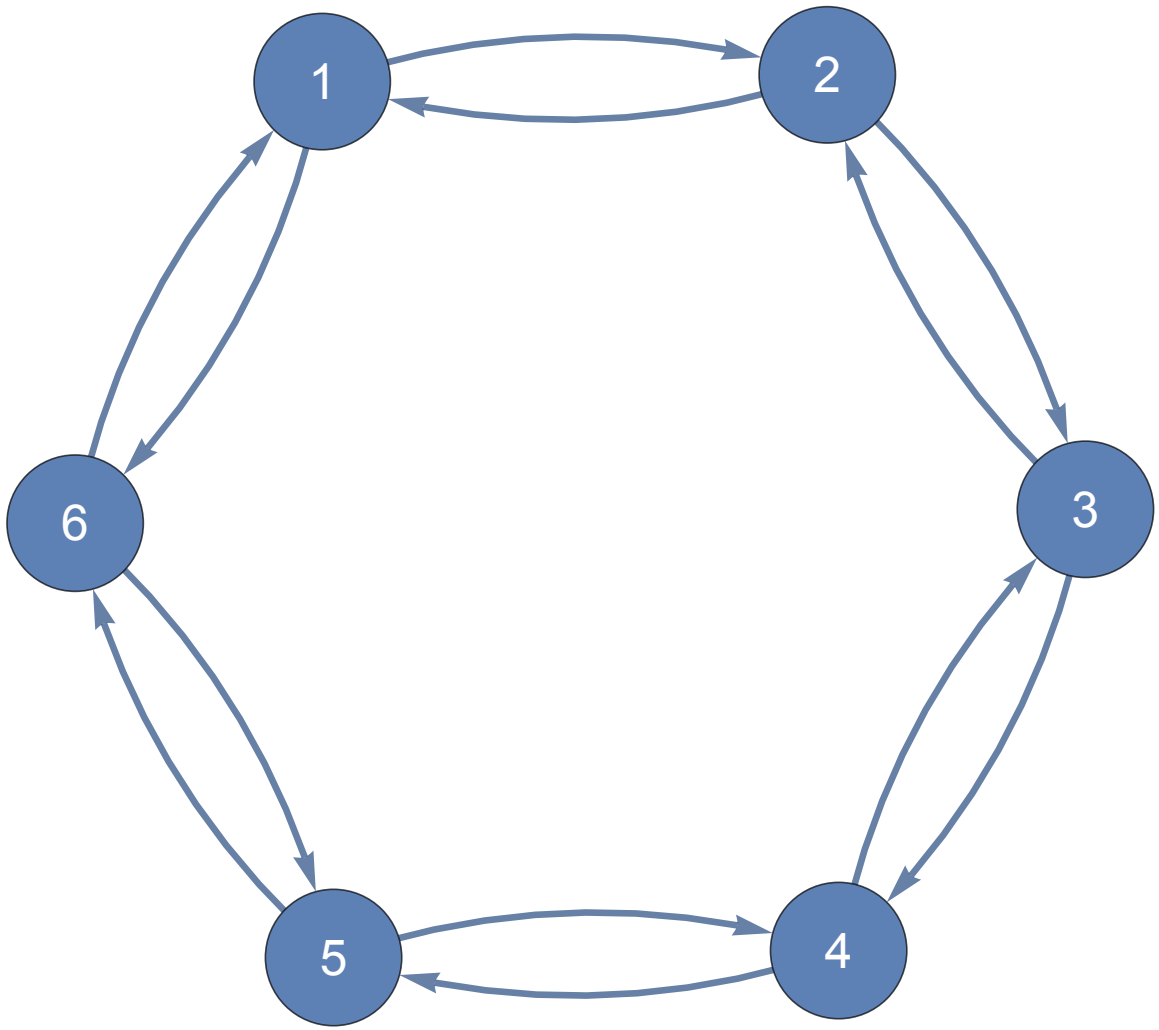
Now suppose we have a die where from each number only consecutive numbers can be obtained,

- $1 \to 2$ or $1 \to 6$; each with probability $1/2$.
- $2 \to 3$ or $2 \to 1$; each with probability $1/2$.
- ...
- $6 \to 1$ or $6 \to 5$; each with probability $1/2$.

This die has **dependence** – the next value we obtain depends on the current value! **However** has same unconditional distribution (i.e. across all throws) as independent die $\to$ same mean.

Introducing a Markovian die

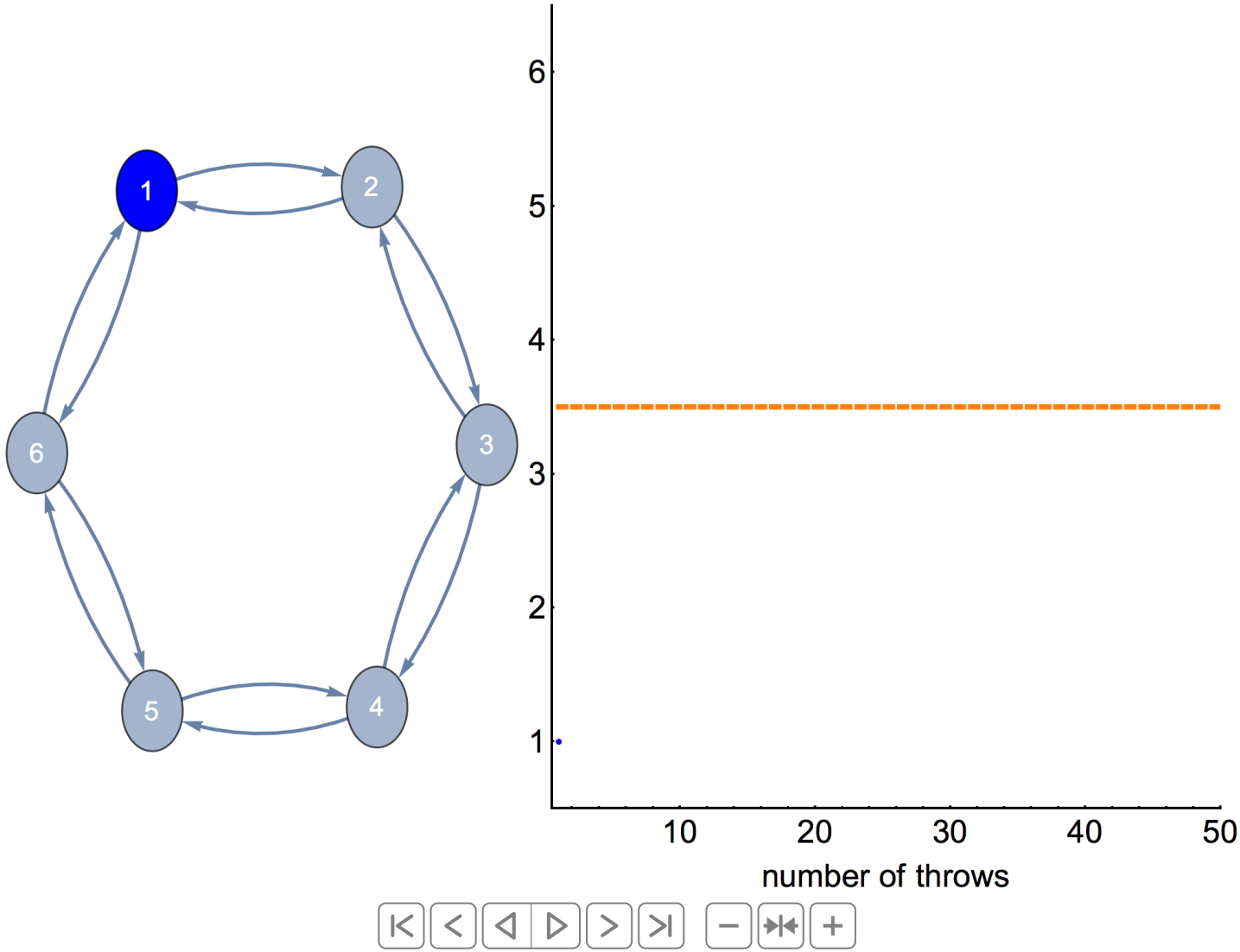
This **Markovian** die can be represented in graphical form,



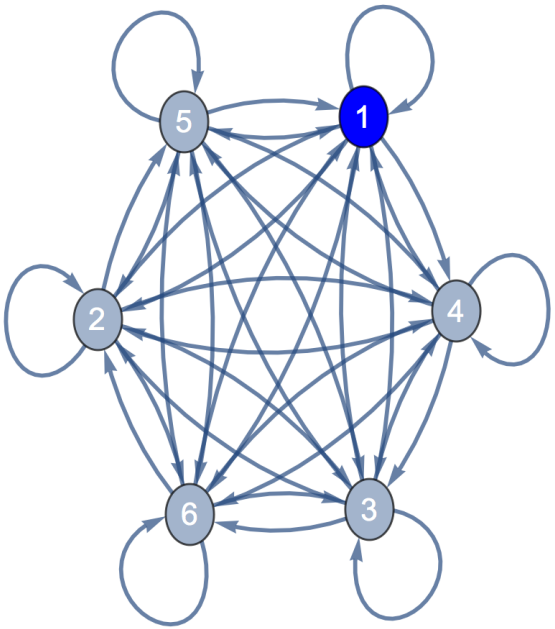
Introducing a Markovian die

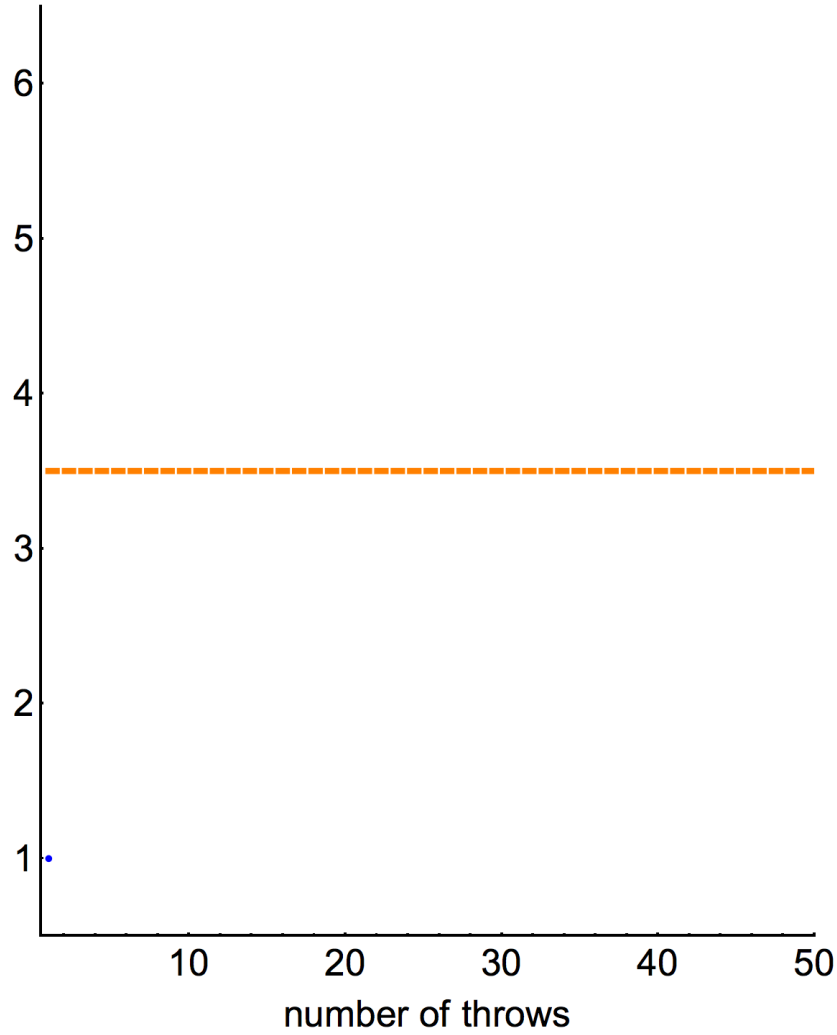
Question: Which of these two dies – the independent and Markovian one – is better able to estimate the mean of the die?
Answer: shake it off (again)!

Effective sample size: Markovian die



Effective sample size: independent die





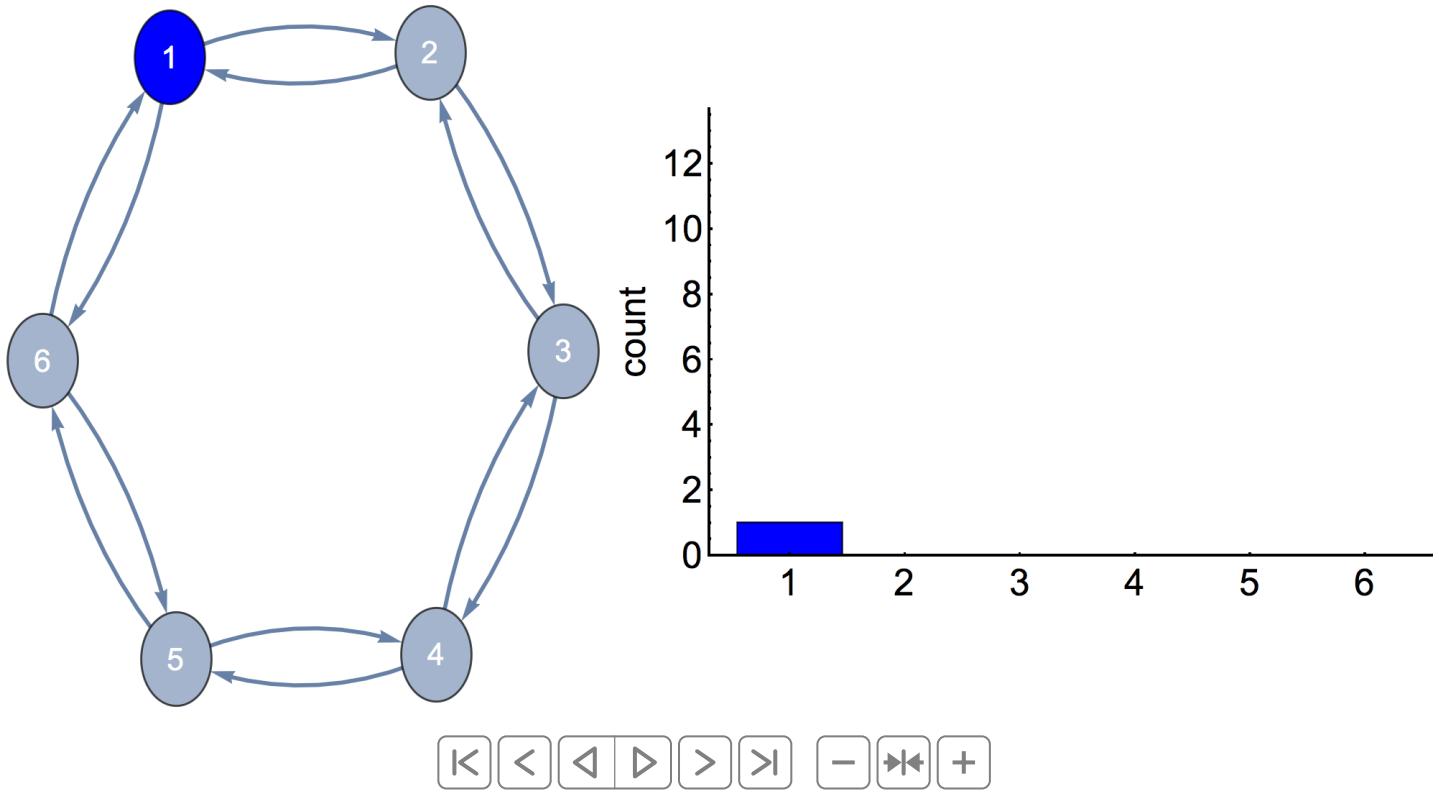
Effective sample size

Conclude: independent die converges more rapidly to true mean!

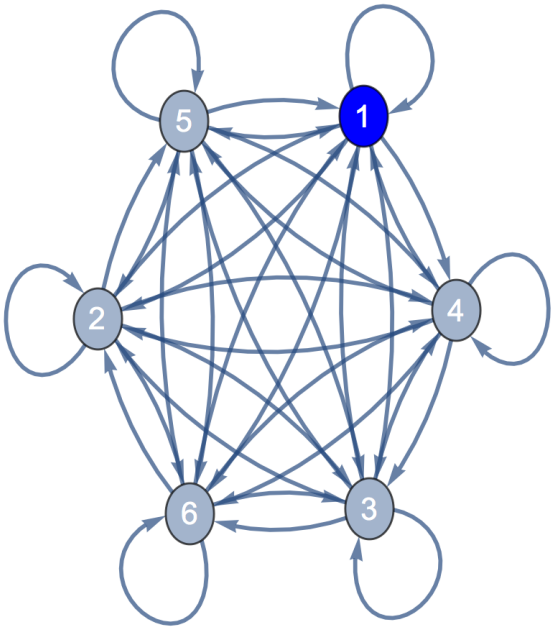
Question: How does each die fare in estimating any other quantity?

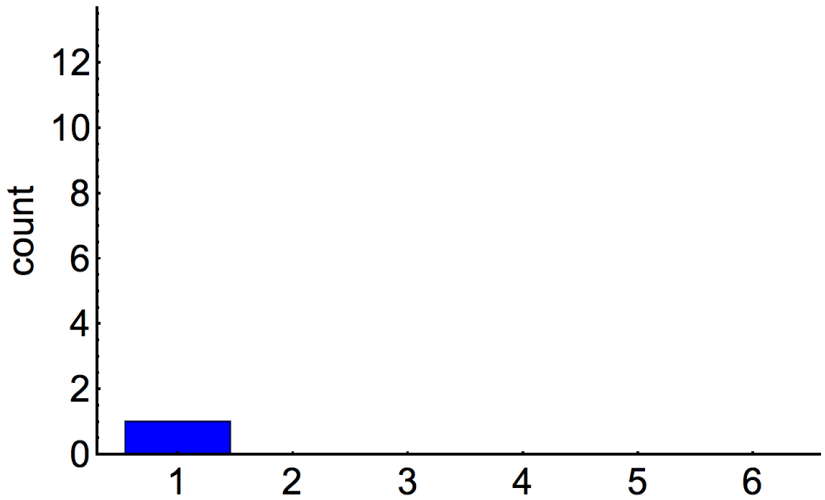
Answer: Depends on how well each method approximates the probability distribution!

Effective sample size: Markovian die



Effective sample size: independent die





Effective sample size: depends on dependence

- The Markovian die performs worse than the independent sampler.
- This is due to the **dependence** of throwing the Markovian die $\implies$ takes longer for sampler to explore parameter space!
- As dependence$\uparrow$ the gap between the independent sampler and the Markovian one increases.
- Therefore conclude that as dependence$\uparrow$ the effective sample size $\downarrow$.

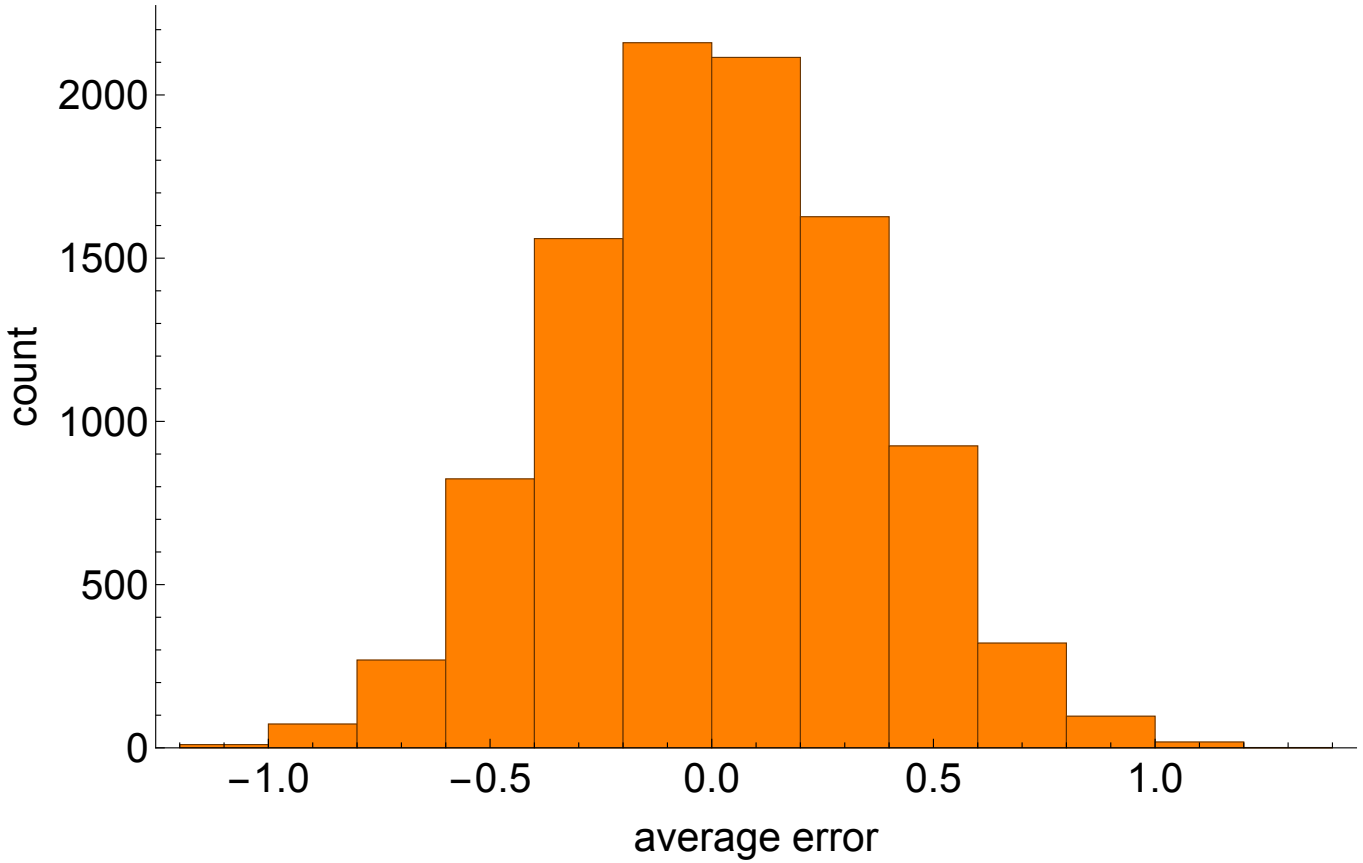
Effective sample size: estimating for the Markovian die

Question: But how do we estimate effective sample size for a sample size of 50 from the Markovian die?

Answer: Estimate the error in estimating the mean (or another quantity if appropriate) for such a sample size, then find a sample size for the independent sampler that equilibrates the error.

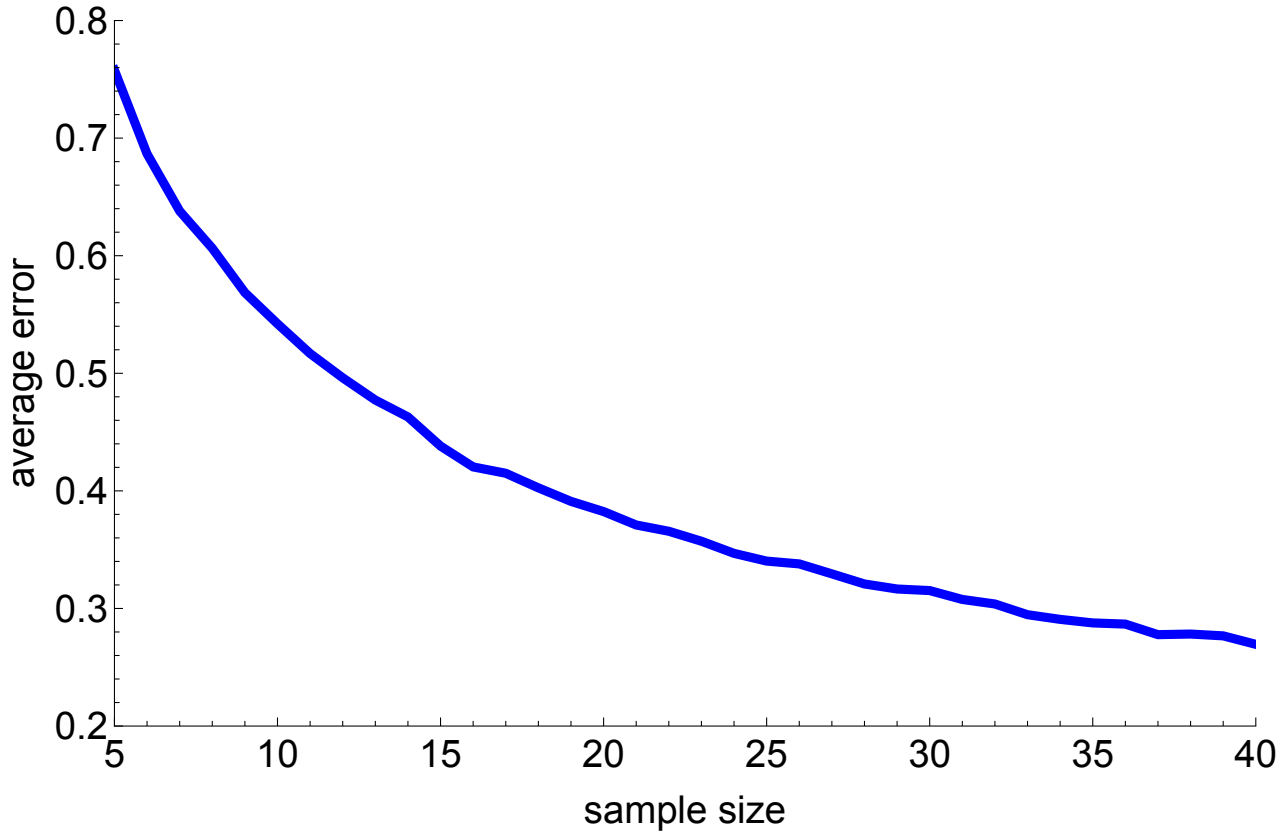
Effective sample size: estimating for the Markovian die

Across 10,000 samples, each of size 50, determine the error in estimating the mean, Root mean squared error ("average error") is ≈ 0.34.



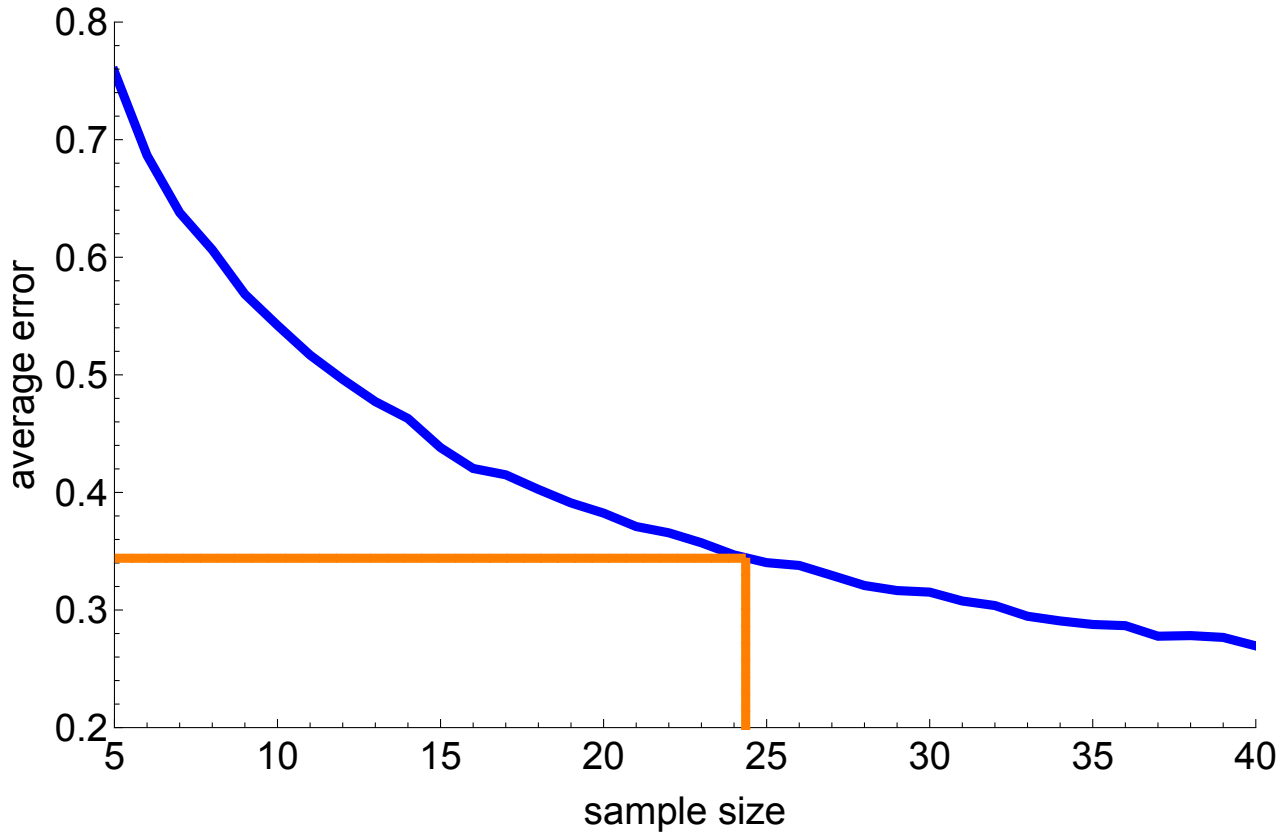
Effective sample size: estimating for the Markovian die

Plot error of independent die as a function of sample size (again using 10,000 replicates of each sample size),



Effective sample size: estimating for the Markovian die

Find sample size that produces equivalent error for independent sampler $\rightarrow$ roughly 24, so independent sampler is about twice as efficient as Markovian die!



Effective sample size: summary

- The worth of a sample is **not** dictated by its number of samples per second.
- More important is the net information gained per second.
- As dependence of sampler increases there is less incremental information gained per sample.
- Quantify this using concept of effective sample size – the equivalent number of samples from an independent sampler.
- Stan calculates this **automatically** for all parameters! No need to know formula.

Defining the Gibbs sampler

For a parameter vector: $\boldsymbol{\theta} = (\theta_1, \theta_2, \theta_3)$:

- Select a random starting location: $(\theta_1^0, \theta_2^0, \theta_3^0)$, along the same lines as for Random Walk Metropolis.
- For each iteration $t = 1, ..., T$ do:
  1. Select a random parameter update ordering, for example $(\theta_3, \theta_2, \theta_1)$.
  2. **Independently** sample from the conditional posterior for each parameter in order using the most up-to-date parameters.

Defining the Gibbs sampler

Start with $(\theta_1^0, \theta_2^0, \theta_3^0)$, then we sample:

$$\theta_3^1 \sim p(\theta_3 | \theta_2^0, \theta_1^0) \tag{1}$$

Then conditional on freshly-sampled θ_3^1:

$$\theta_2^1 \sim p(\theta_2 | \theta_1^0, \theta_3^1) \tag{2}$$

Then conditional on freshly-sampled θ_3^1 and θ_2^1:

$$\theta_1^1 \sim p(\theta_1 | \theta_2^1, \theta_3^1) \tag{3}$$

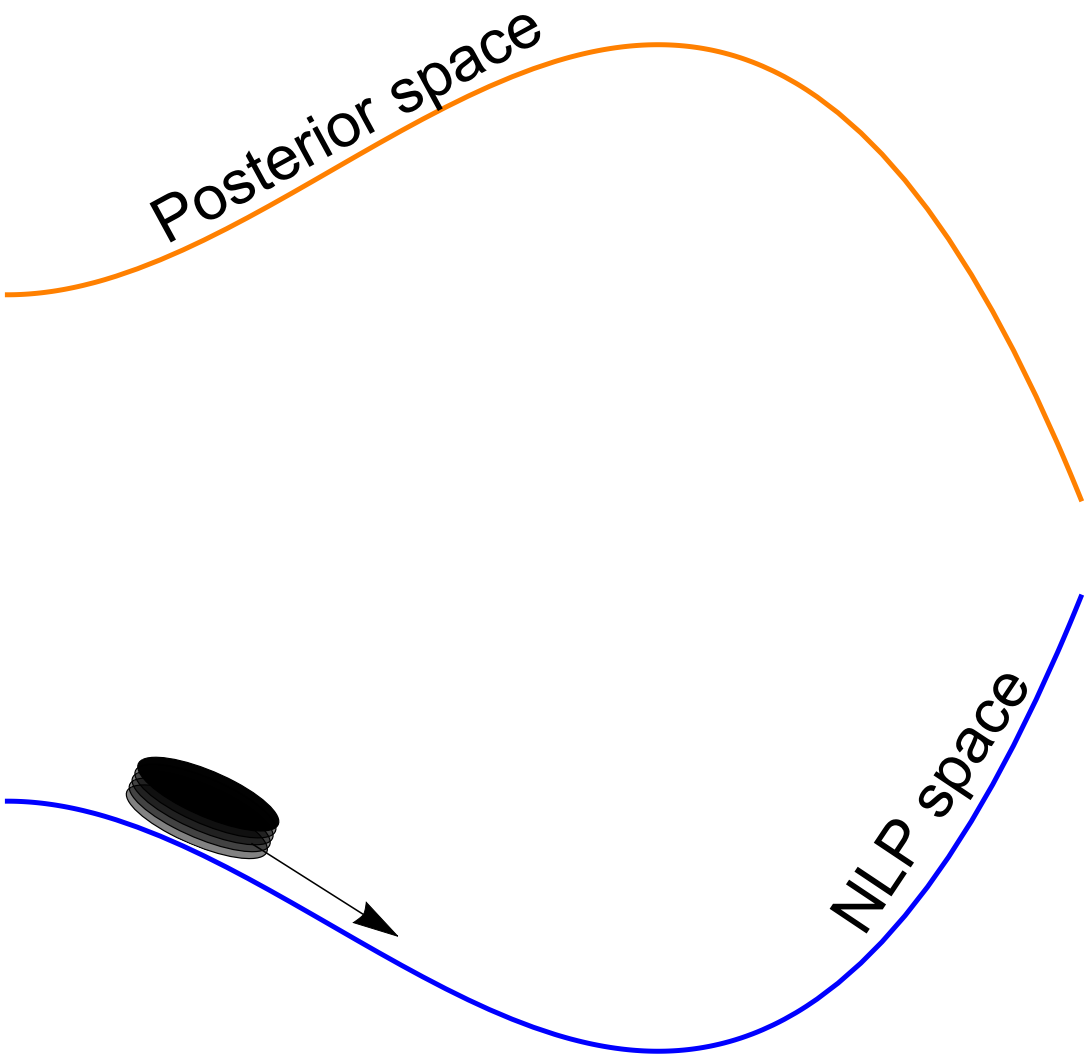
How to make a Gibbs sampler?

- Typically better efficiency (i.e. higher effective sample size per iteration) than Random Walk Metropolis.
- Gibbs sampling requires that we derive **conditional** densities, for example $p(\theta_1|\theta_2, \theta_3)$.
- But how can we do find these densities in practice? (See problem set!)
- $\Longrightarrow$ write down full posterior distribution, and take only those parts which involve the variable(s) that is being sampled.
- Then use knowledge of distributions to determine densities.
- Would prefer a method that didn't require us to do any maths!

Introduction to Hamiltonian Monte Carlo

- Assume a space related to posterior space can be thought of as a landscape.
- Imagine an ice puck moving over the frictionless surface of this terrain.
- At defined time points we measure the location of the puck, and instantaneously give the puck a shove in a random direction.
- The locations traced out by the puck represent proposed steps from our sampler.
- Based on the height of the posterior and momentum of the puck we accept/reject steps.

Why does this physical analogy help us?



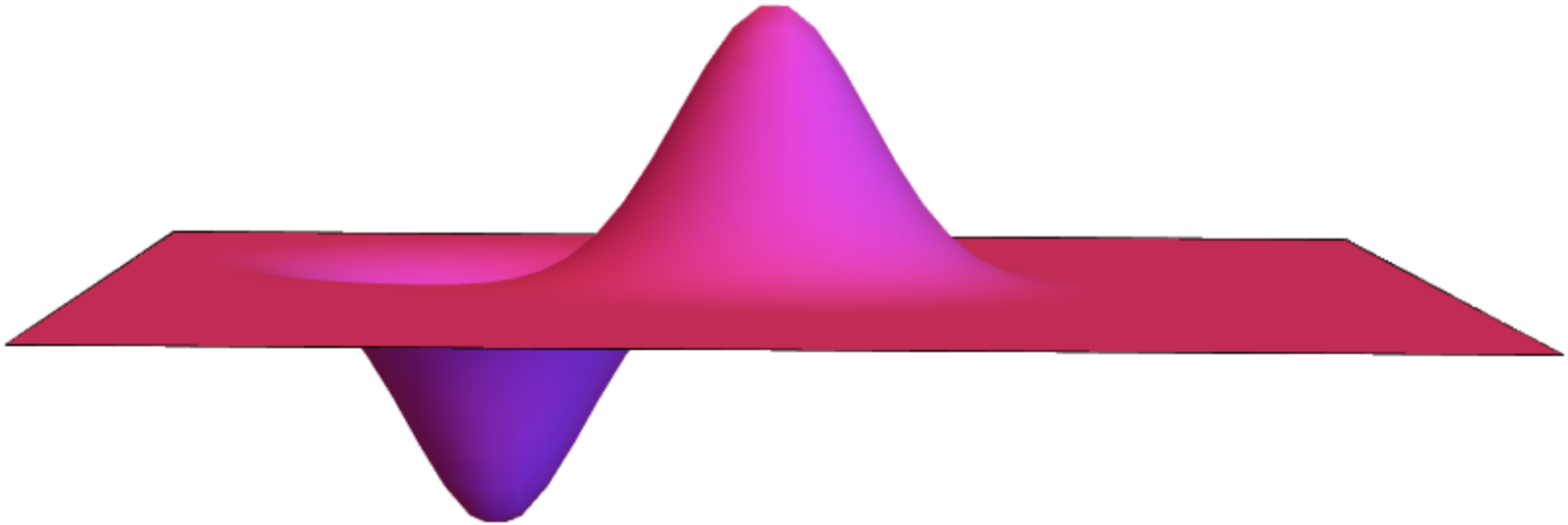
Why does this physical analogy help us?

- Allow the potential energy of the puck to be determined partly by the posterior density.
- $\Longrightarrow$ puck will move in the "natural" directions dictated by the posterior geometry.
- And will visit areas of low NLP $\Longrightarrow$ high posterior density.
- **NLP** stands for the **negative log (un-normalised) posterior**,

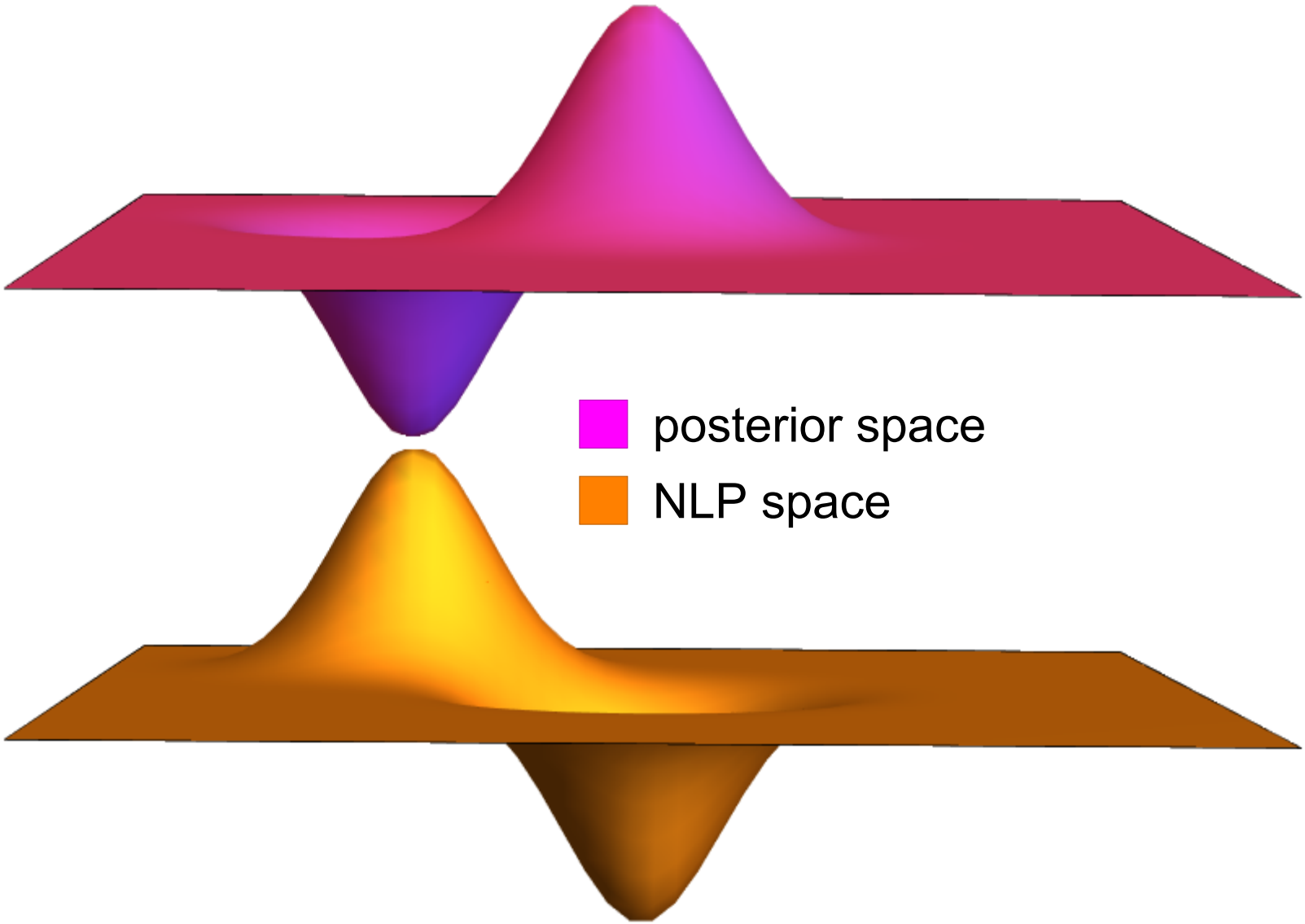
$$NLP = -\log\ [p(X|\theta) \times p(\theta)] \tag{4}$$

Important to remember that HMC uses the **log** of the posterior. (When coding up model in Stan "sampling" statements amount to incrementing the log probability.)

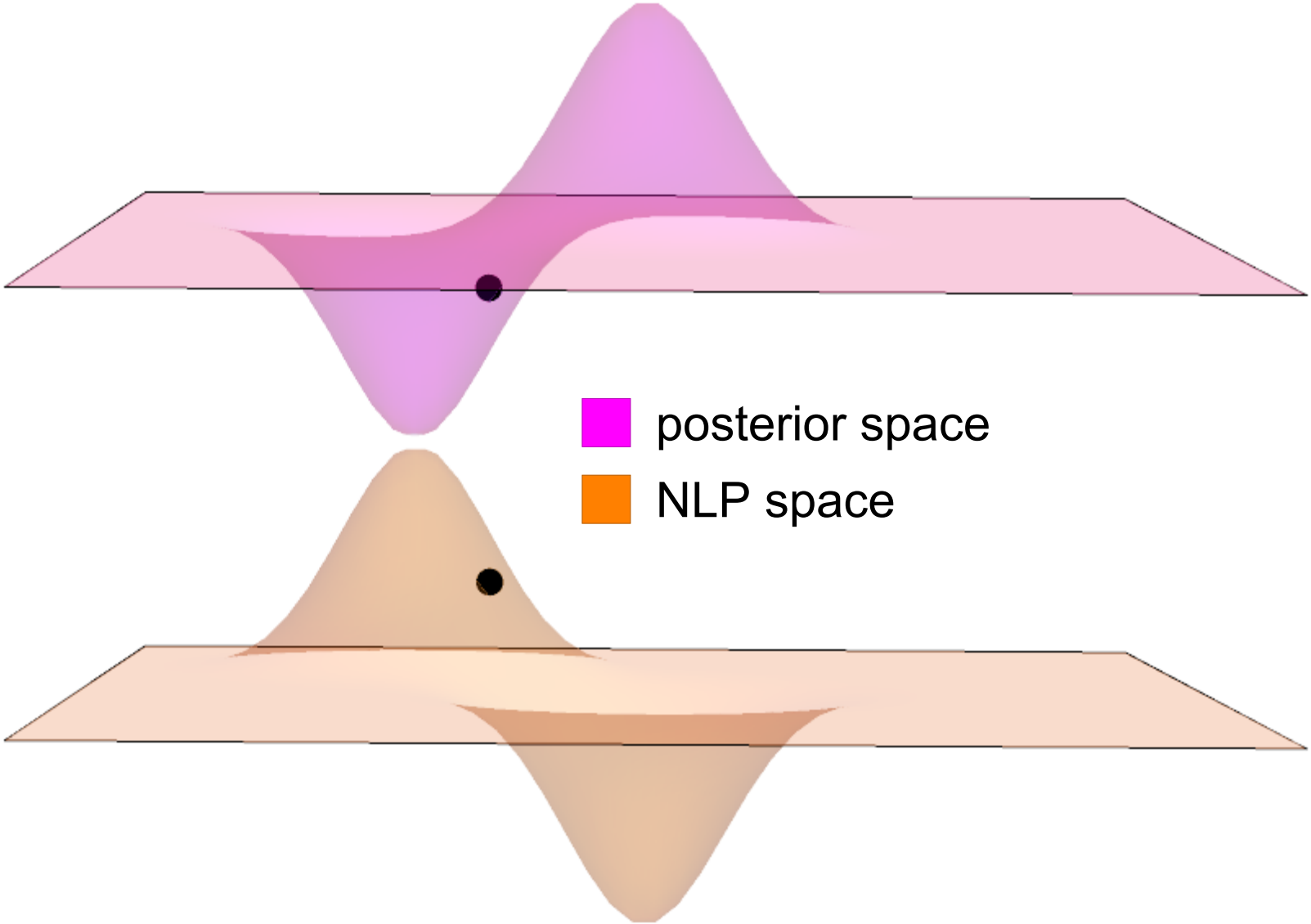
Simulating the puck's motion in NLP space: start with a posterior



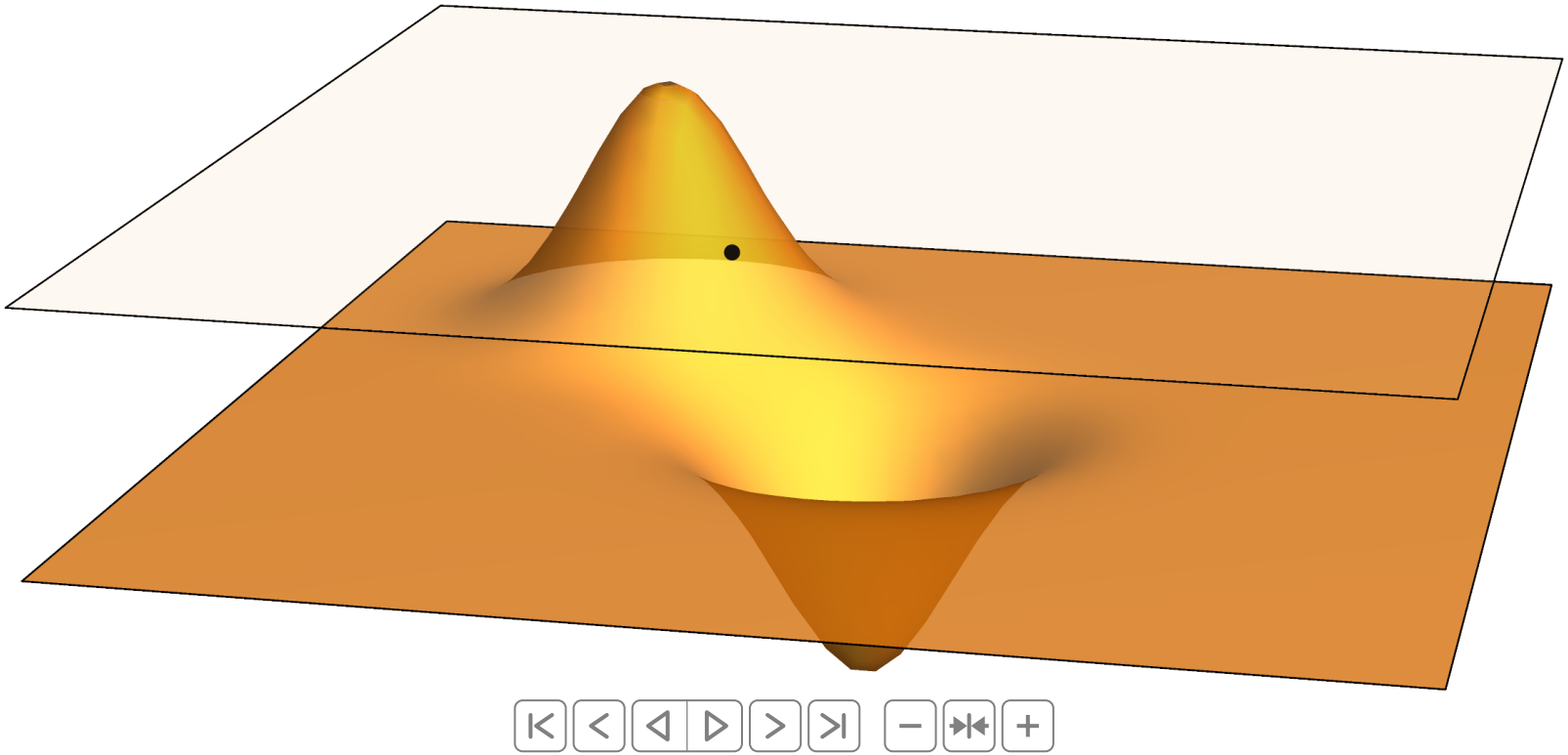
Simulating the puck's motion in NLP space: find NLP space



Simulating the puck's motion in NLP space: consider a point in posterior space



The path traced out for 100 different shoves from same distribution



The HMC algorithm

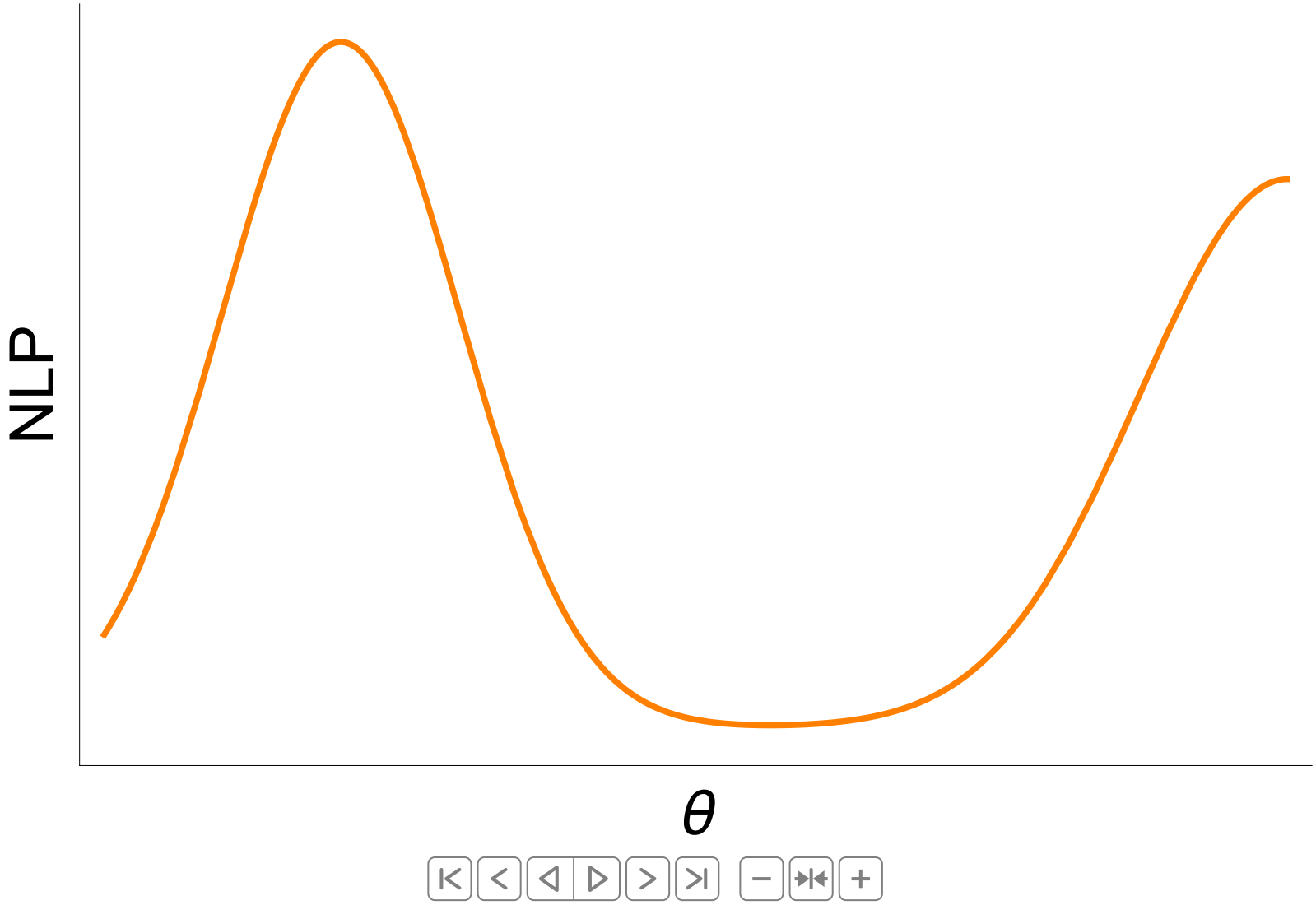
The HMC algorithm works as follows,

- Start at random location θ_0.
- For $i = 1, ..., N$ do:
  1. Give puck random initial momentum, $k \sim N(0, \Sigma)$.
  2. Simulate puck's movement across NLP surface for a fixed time T (number of discrete steps of numerical integration algo).
  3. Compute a ratio:

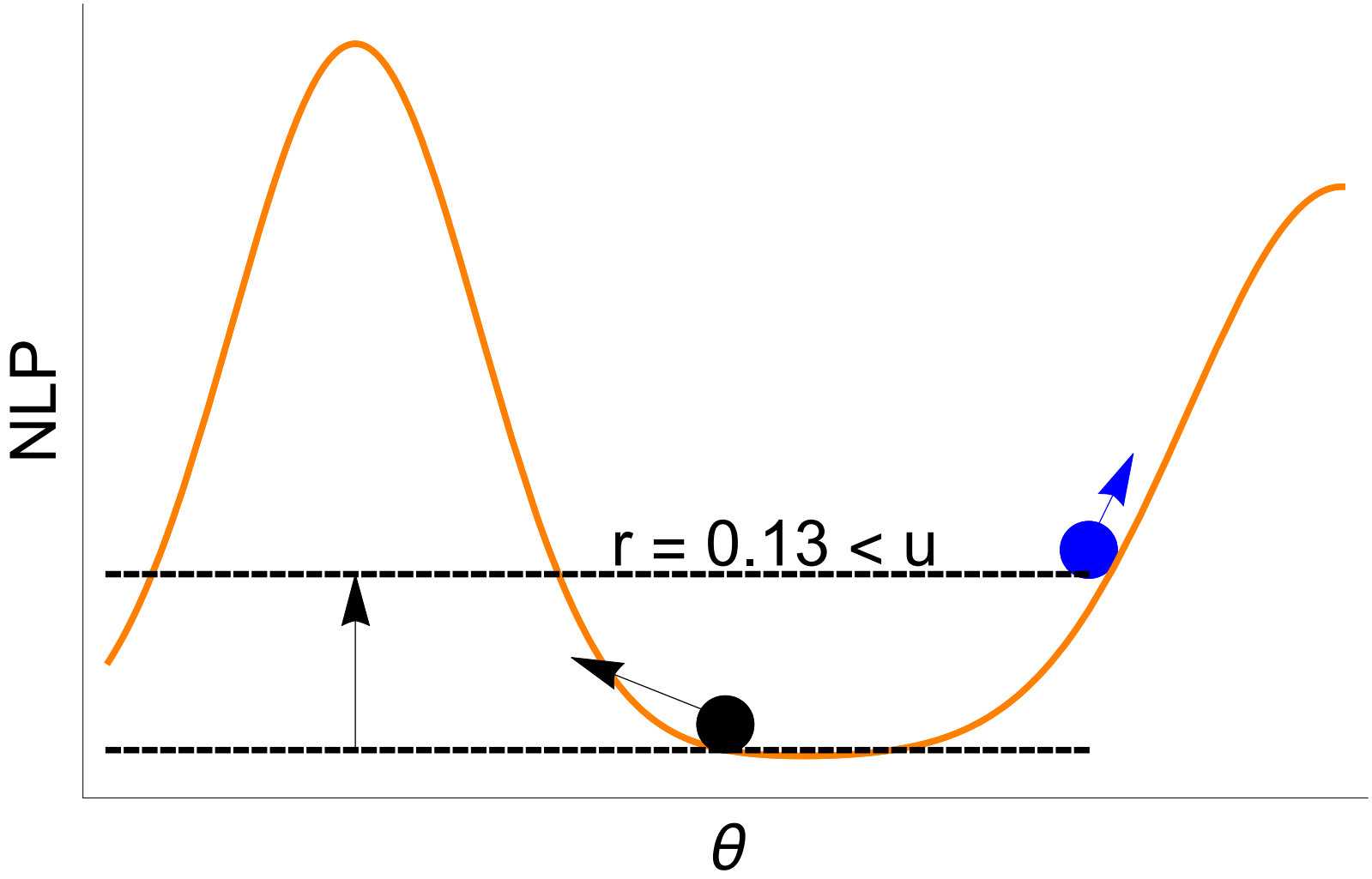
$$r = \frac{p(\theta_t|X)}{p(\theta_{t-1}|X)} \times \frac{p(k')}{p(k)} \tag{5}$$

     where k is the final momentum.
  4. If $r > u \implies$ move to θ_t, otherwise return to θ_{t-1}.

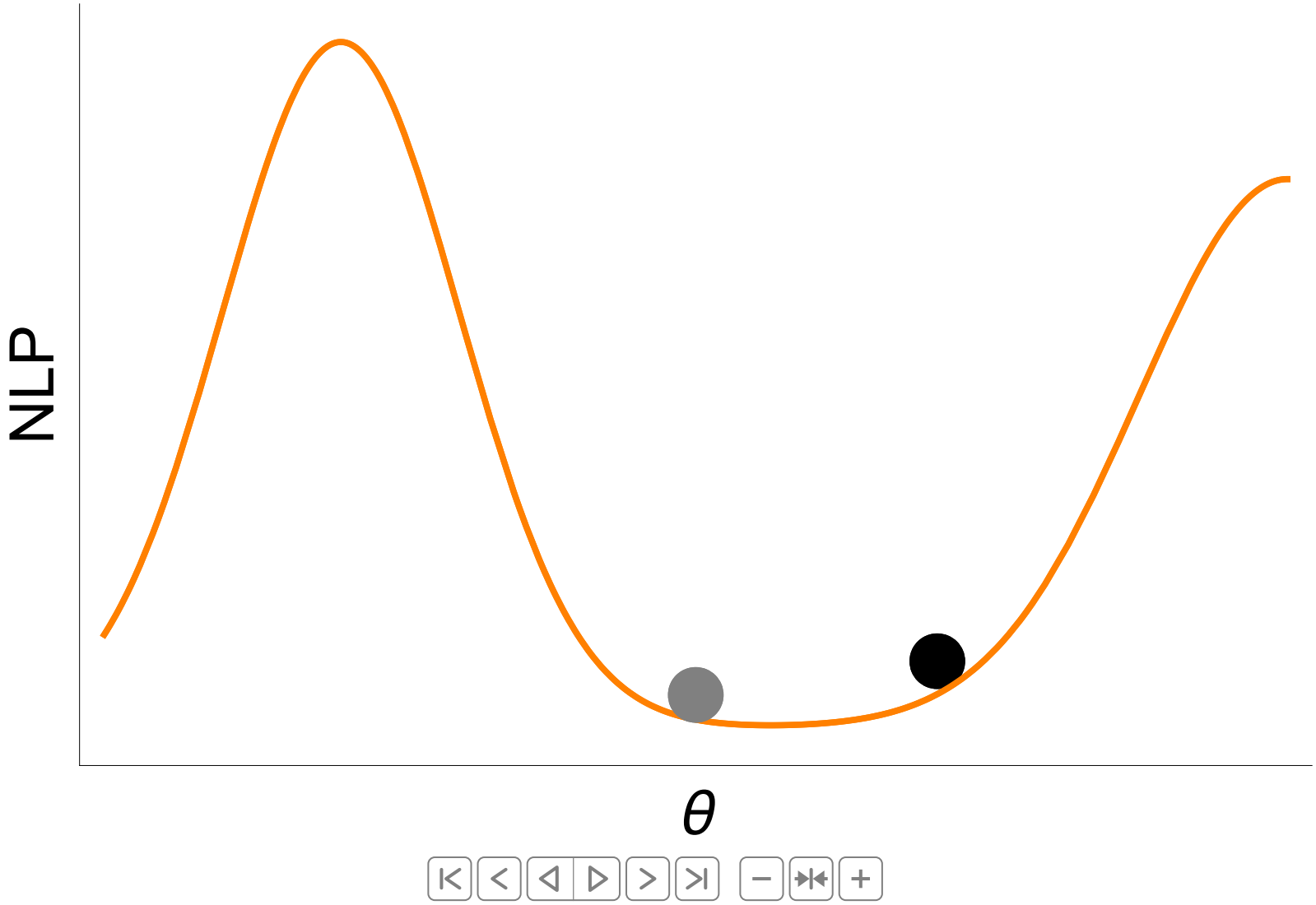
The HMC algorithm



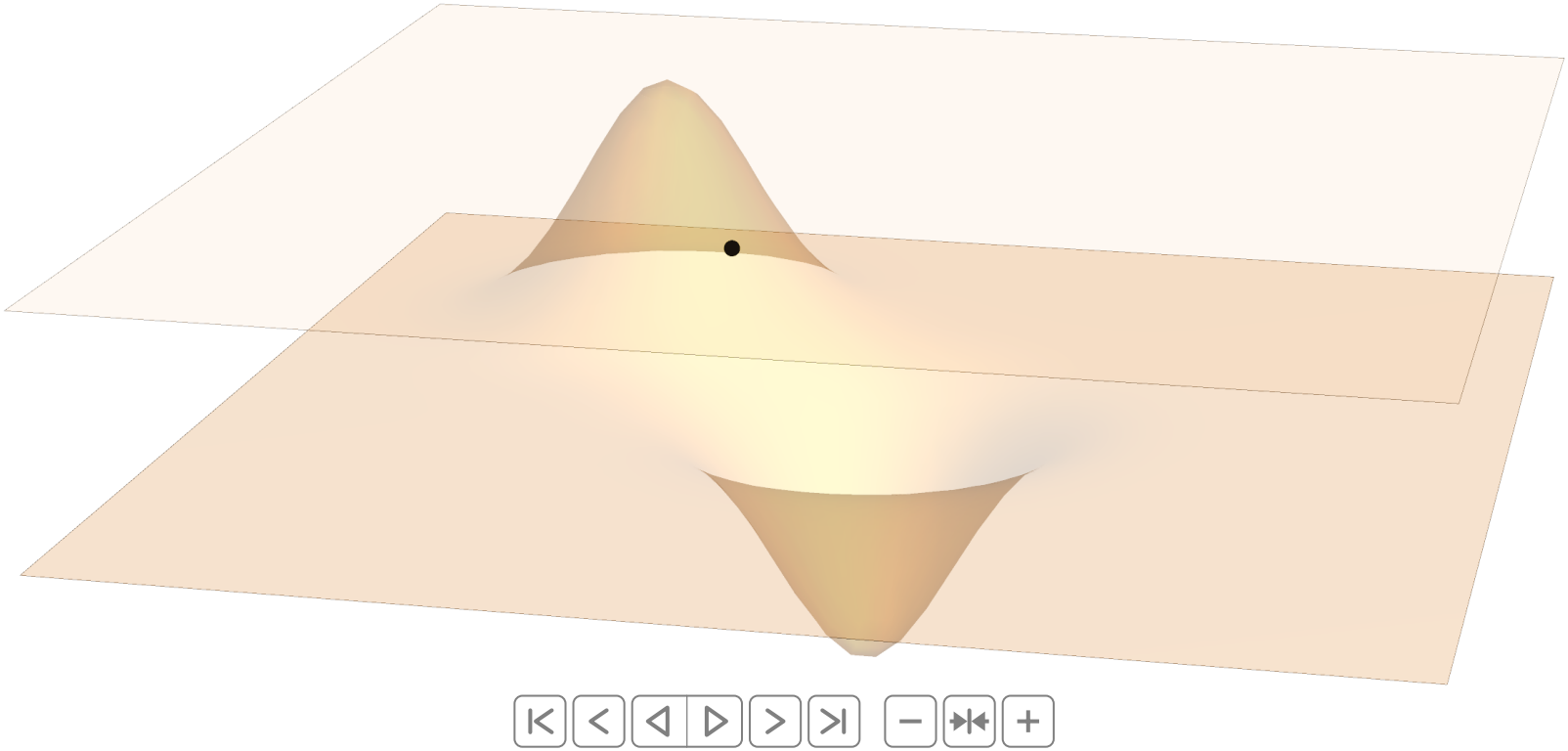
The HMC algorithm



The HMC algorithm

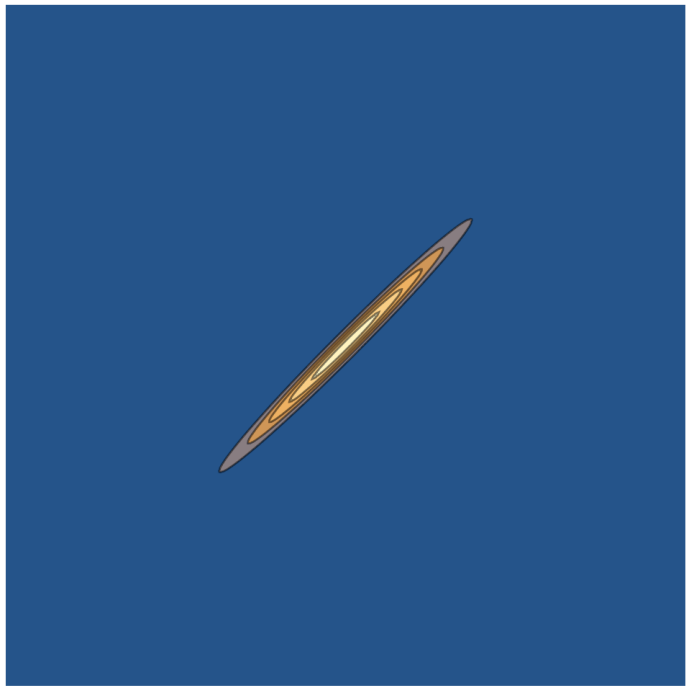


Simulating the puck's motion in NLP space: randomly shoving the puck at intervals of 50 steps

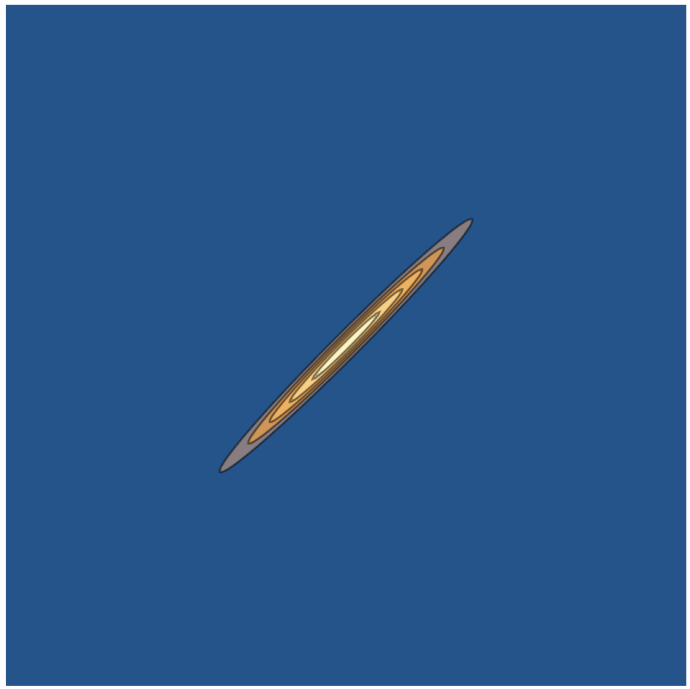


RWM and Gibbs performance

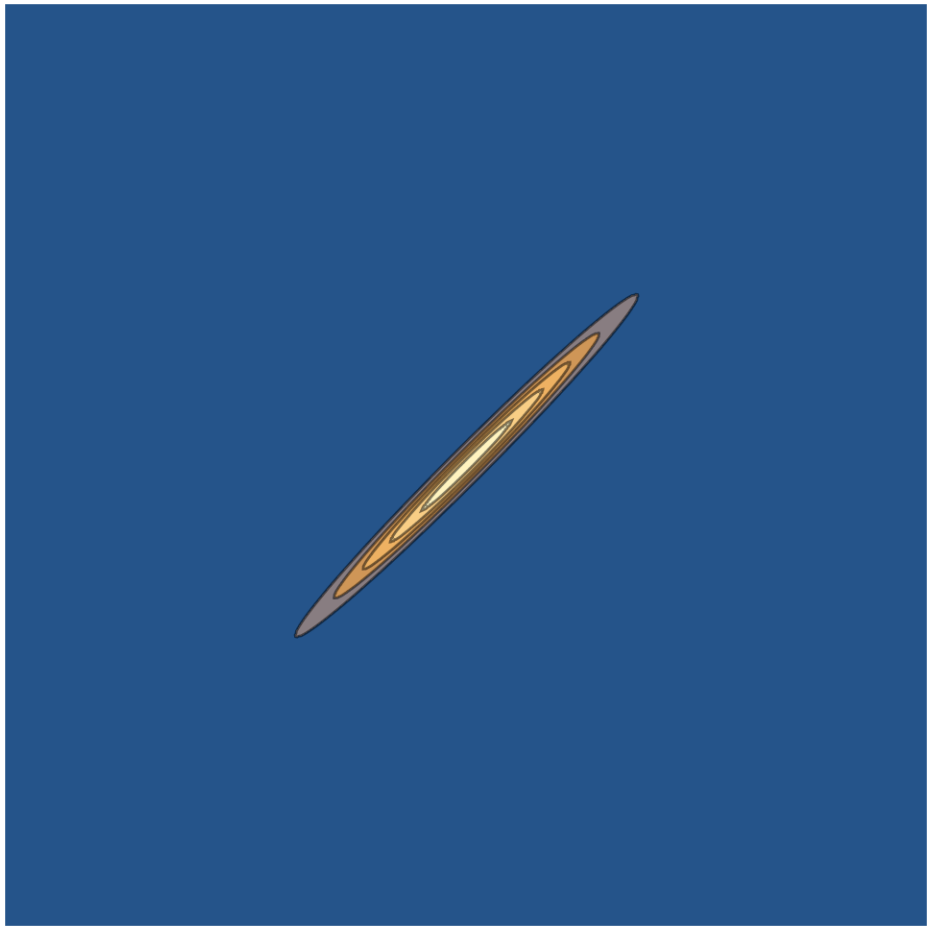
RWM



Gibbs



Hamiltonian Monte Carlo performance



Hamiltonian Monte Carlo: summary

- Imagine a puck sliding across a frictionless surface of the negative log posterior (NLP).
- Give puck random shoves at predefined time intervals, and simulate path of puck for predefined time interval.
- Puck will tend to visit areas of low NLP which correspond to areas of high posterior density!
- Algorithm often performs considerably better than Random Walk Metropolis or Gibbs.
- Stan uses a fancy variant of HMC known as NUTS (No U-Turn Sampler) that determines the optimal time intervals to simulate puck for.

Coding up MCMC algorithms

- Up until now we have coded up our own MCMC algorithms $\Longrightarrow$ (relatively) straightforward for simple models.
- However for more complex models this is a time-consuming (and frustrating) process.
- Flavours of MCMC algorithms:
  - **Random Walk Metropolis:** fairly basic but ineffectual for many real-life models.
  - **Gibbs:** a bit faster than RWM but also suffers from same issues. However can be significantly harder to code up!
  - **Hamiltonian Monte Carlo:** significantly more efficient than the aforementioned but also significantly harder to code and tune.

Introducing Stan: avoiding manual labour

Stan is an intuitive yet sophisticated programming language that does the hard work for us.
What is Stan and how do we use it?

- **Imperative** programming language like R, Python, Matlab, C++ etc. (BUGS/JAGS are a weird type of language called **declarative**.)
- Turing-complete language (unlike BUGS/JAGS) and works like most other languages: can use loops, conditional statements, and functions.
- Code up a model in Stan and then it implements Hamiltonian Monte Carlo (actually something called NUTS but similar) for us.

Why should we use Stan?

- Stan is the brainchild of Andrew Gelman at Colombia; the World's foremost Bayesian statistician.
- Stan's uses an extension of HMC called NUTS that automatically tunes. It is **fast**. Very fast $\Longrightarrow$ typically generates multiples times as many effective samples per second than BUGS/JAGS.
- Stan is **simple** to learn.
- Stan has excellent documentation (a manual full of extensive examples).
- The Stan team have translated **all** the example models from popular books; Gelman, Kruschke etc.
- **Most important:** Stan has a very active and helpful user forum and development team; for example, typical question answered in less than a couple of hours.

Why should we use Stan?

Overall: Stan makes our life easier.

- It is easy to learn; even if you are used to BUGS/JAGS or something else.
- The language is here to stay $\Longrightarrow$ all recent books use examples written in Stan rather than BUGS/JAGS.
- **Important:** it is popular $\Longrightarrow$ if you have a problem with your model you can get help, **fast**.
- The best minds in the business are working on making it even better.
- Finally, "Shiny Stan" makes it really quite fun!

How do we use it?

- Code up model in Stan code in a text editor and save as ".stan" file.
- Call Stan to run the model from:
  - R.
  - Python.
  - The command line.
  - Matlab.
  - Stata.
  - Julia.
- Use one of the above to analyse the data (of course you can export to another one.)

A straightforward example

Suppose:

- We record the height, Y_i, of 10 people.
- We want a model to explain the variation, and choose a normal likelihood:

$$Y_i \sim N(\mu, \sigma) \tag{6}$$

- We choose the following (independent) priors on each parameter:
  - $\mu \sim N(0, 1)$.
  - $\sigma \sim \text{gamma}(1, 1)$.

  Question: how do we code this up in Stan?

An example Stan program: heights

```
data {
    real Y[10]; ## Heights for 10 people
}

parameters {
    real mu;
    real<lower=0> sigma;
}

model {
    Y ~ normal(mu,sigma); ## Likelihood
    mu ~ normal(0,1); ## Prior for mu
    sigma ~ gamma(1,1); ## Prior for sigma
}
```

An example Stan program: data block

```
data {
    real Y[10]; ## Heights for 10 people
}
```

- Declare all data that you will pass to Stan to estimate your model.
- Terminate all statements with a semi-colon ";".
- Use "##" or "//" for comments.

An example Stan program: data block

```
data {
    real Y[10]; ## Heights for 10 people
}
```

Strongly, statically-typed language: need to tell Stan the type of data variable. For example:

- `real` for continuous data.
- `int` for discrete data.
- Arrays: above we specified `Y` as an array of continuous data of length 10.

An example Stan program: data block

```
data {
    real Y[10]; ## Heights for 10 people
}
```

- Can place limits on data, for example:
  - `real<lower=0,upper=1> X`
  - `real<lower=0> Z`
- Vectors and matrices; only contain reals and can be used for matrix operations.

An example Stan program: parameter block

```
parameters {
    real mu;
    real<lower=0> sigma;
}
```

- Declare all parameters that you use in your model.
- Place limits on variables, for example `real<lower=0> sigma` above.
- A multitude of parameter types including some of the aforementioned:
  - `real` for continuous parameters.
  - Arrays of types, for example `real epsilon[12]`.

An example Stan program: parameter block

```
parameters {
    real mu;
    real<lower=0> sigma;
}
```

- `vector` or `matrix`, specified by:
  - `vector[5] beta`
  - `matrix[5,3] gamma`
- `simplex` for a parameter vector that must sum to 1.
- More exotic types like `corr_matrix` , or `ordered`.

An example Stan program: parameter block

```
parameters {
    real mu;
    real<lower=0> sigma;
}
```

Important: HMC/NUTS not developed yet to work with **discrete** parameters $\implies$ following options in Stan:

- Marginalise out the parameter. For example, if $p(\beta, \theta)$ where β is continuous and θ is discrete:

$$p(\beta) = \sum_{i=1}^{K} p(\beta, \theta_i) \tag{7}$$

- Some models can be reformulated without discrete parameters.
- Failing either of the above $\implies$ use BUGS/JAGS.

An example Stan program: model block

```
model {
    Y ~ normal(mu,sigma); ## Likelihood
    mu ~ normal(0,1); ## Prior for mu
    sigma ~ gamma(1,1); ## Prior for sigma
}
```

- Used to define:
  - Likelihood.
  - Priors on parameters.
- If don't specify priors on parameters $\Longrightarrow$ Stan assumes you are using flat priors (which can be improper.)

An example Stan program: model block

```
model {
    Y ~ normal(mu,sigma); ## Likelihood
    mu ~ normal(0,1); ## Prior for mu
    sigma ~ gamma(1,1); ## Prior for sigma
}
```

- **Huge** range of probability distributions covered, across a range of parameterisations. For example:
  - **Discrete:** bernoulli, binomial, normal, poisson, beta-binomial, negative-binomial, categorical, multinomial.
  - **Continuous unbounded:** normal, skew-normal, student-t, cauchy, logistic.
  - **Continuous bounded:** uniform, beta, log-normal, exponential, gamma, chi-squared, inverse-chi-squared, weibull, wiener diffusion, pareto.

An example Stan program: model block

```
model {
    Y ~ normal(mu,sigma); ## Likelihood
    mu ~ normal(0,1); ## Prior for mu
    sigma ~ gamma(1,1); ## Prior for sigma
}
```

- **Multivariate continuous:** normal, student-t, gaussian process.
- **Exotics:** dirichlet, LKJ correlation distribution, wishart and its inverse, von-mises.

Running Stan

Write model in a text editing program. For example:

- **emacs** - Stan syntax actually supported.
- **notepad++**.
- **Word** or **notepad**.

$\Longrightarrow$ save as "exampleModel.stan" in same directory as you run statistical software.

Running Stan in R

```
## Load packages
library(rstan)

## Generate fake data
Y = rnorm(10,mean = 0, sd = 1)

## Compile and run model, and save in fit
fit = stan(file='exampleModel.stan',data=list(Y=Y),
           iter=1000,chains=4)
```

Running Stan on example model

```
## Compile and run model, and save in fit
fit = stan(file='exampleModel.stan',data=list(Y=Y),
           iter=1000,chains=4)
```

The above R code runs NUTS for our model with the following options:

- 1000 MCMC samples of which 500 are discarded as warm-up.
- Across 4 chains.
- Using a random number seed of 1 (good to ensure you can reproduce results.)

Example model: results

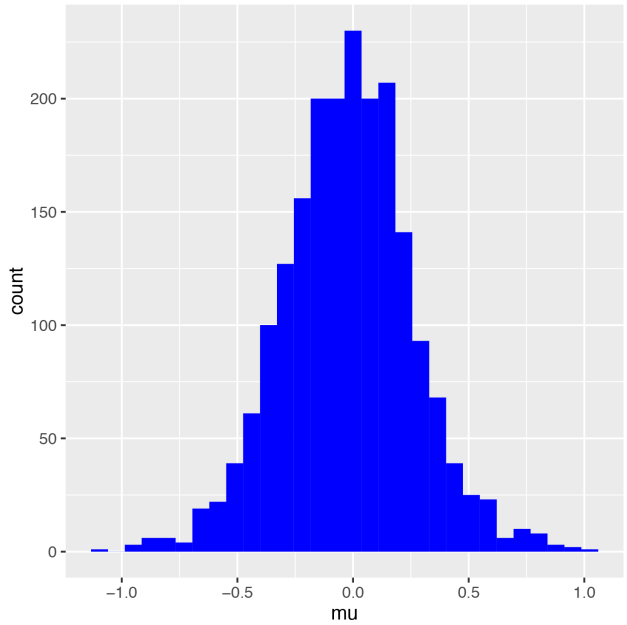
```
## Print summary statistics
print(fit,probs = c(0.25, 0.5, 0.75))

## Inference for Stan model: exampleModel.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500
##
##         mean se_mean   sd   25%   50%   75% n_eff Rhat
## mu     -0.02    0.01 0.28 -0.20 -0.02  0.16   689    1
## sigma  0.95    0.01 0.24  0.78  0.91  1.08   696    1
## lp__  -5.26    0.04 1.00 -5.69 -4.97 -4.50   639    1
```

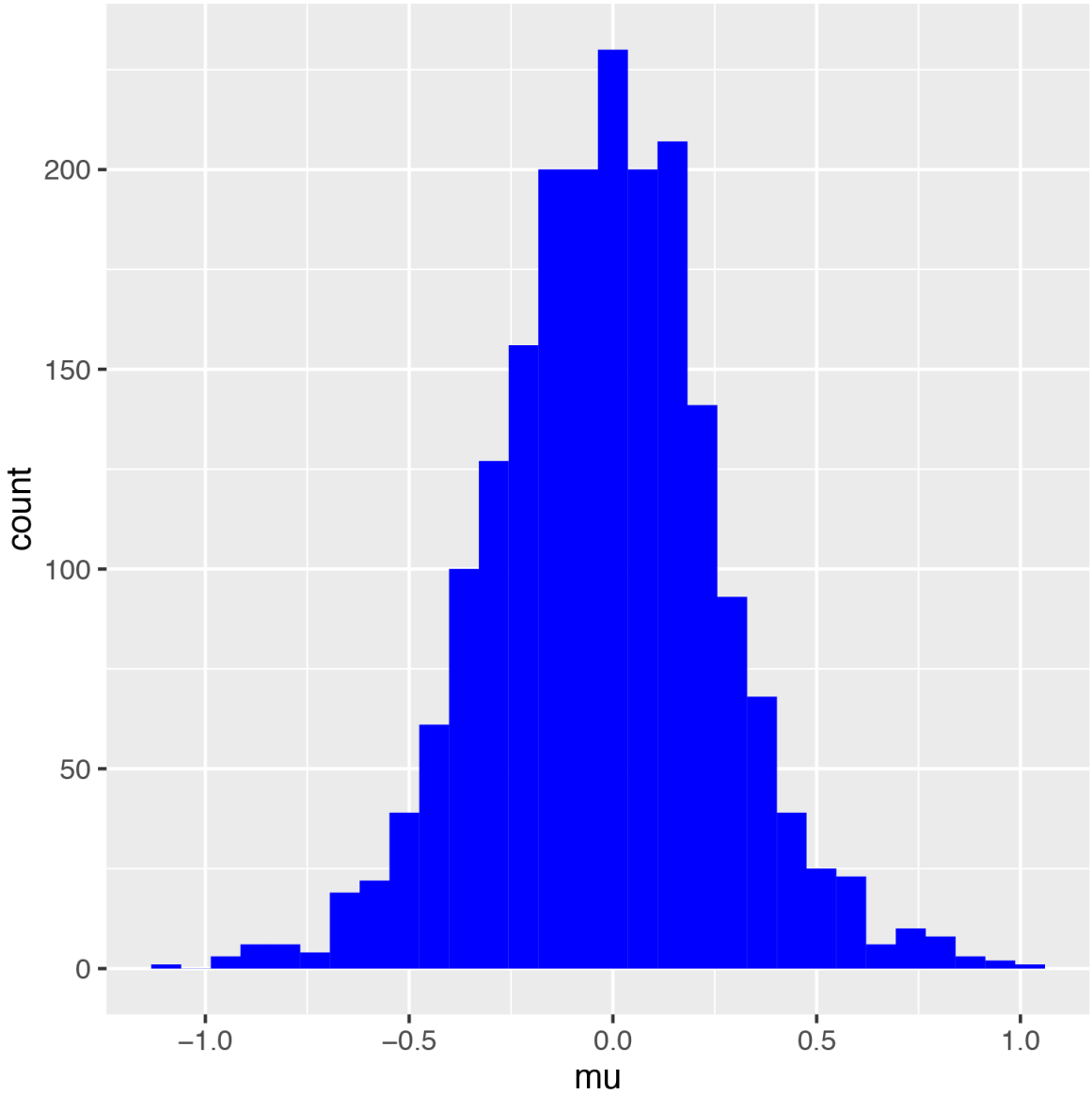
Example model: results

```
## Extract element and plot
mu = extract(fit,'mu')[[1]]

## Plot histogram
library(ggplot2)
qplot(mu, fill=I("blue"))
```



Example model: results



Quick note: what does $\sim$ actually mean?

```
model {
    Y ~ normal(mu,sigma); ## Likelihood
    mu ~ normal(0,1); ## Prior for mu
    sigma ~ gamma(1,1); ## Prior for sigma
}
```

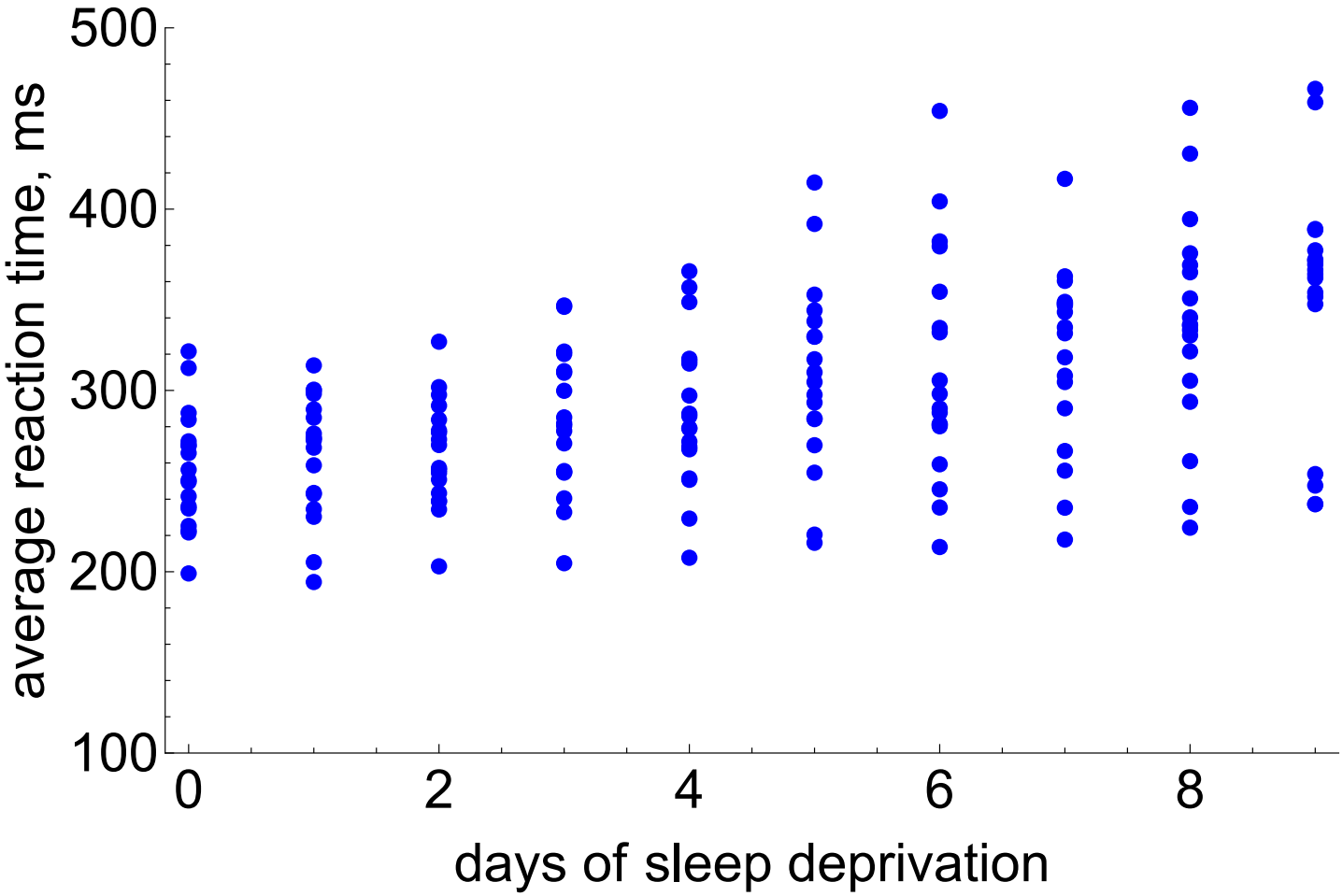
- $\sim$ doesn't mean "sampling" although for many circumstances it isn't terrible to think of it like that.
- Remember: HMC/NUTS uses the negative of the log-posterior to find potential energy of puck.
- $\implies$ logging the posterior converts it from a product into a sum.
- As such $\sim$ really means "increment log probability".

Sleep deprivation study: revisited

- Data from a laboratory experiment that measured the effect of sleep deprivation on cognitive performance[1].
- 18 subjects within a population of interest - long-distance lorry drivers - volunteered to participate in the 10 day experiment.
- Subjects were restricted to 3 hours of sleep per night.
- On each day the subjects' reaction time across a range of cognitive tasks were measured.

Sleep deprivation study: model aims

- Build a model to explain the effect of sleep deprivation on reaction times.



Sleep deprivation study: model

On the basis of the previous graph, assume:

$$R(t) \sim N(\alpha + \beta t, \sigma) \tag{8}$$

where $R(t)$ is the reaction time for a sleep deprivation of t days.

Question: how can we code this up in Stan?

Sleep deprivation study: 1st part of Stan code

```
data {
    int N; ## number of observations
    vector[N] t; ## days of sleep deprivation
    vector[N] R; ## reaction times
}
parameters {
    real alpha; ## reaction time at start
    real beta; ## daily increment to reaction time
    real<lower=0> sigma; ## variation about mean
}
```

Sleep deprivation study: 2nd part of Stan code

```
model {
    ## likelihood
    for (i in 1:N){
        R[i] ~ normal(alpha + beta * t[i], sigma);
    }

    ## priors
    alpha ~ normal(0,250);
    beta ~ normal(0,250);
    sigma ~ normal(0,50);
}
```

Sleep deprivation study: 2nd part of Stan code

However can write same model with faster and more efficient Stan code using vectorization.

```
data {
    int N; ## number of observations
    matrix[N,2] X; ## ones + days of sleep depriv.
    vector[N] R; ## reaction times
 }
 parameters {
    vector[2] gamma;
    real<lower=0> sigma;
  }
model {
     ## likelihood
     R ~ normal(X * gamma, sigma);
     gamma ~ normal(0,250);
}
```

Sleep deprivation study: Results

Sleep deprivation study: Shiny Stan

Demo: instead of printing use Shiny Stan.

Sleep deprivation study: posterior predictive distribution

- Want to carry out posterior predictive checks $\Longrightarrow$ need posterior predictive distribution.
- Use "generated quantities" block.

```
generated quantities {
    vector[N] R_sim; ## Store post-pred samples
    for (i in 1:N){
        R_sim[i] = normal_rng(X[i] * gamma, sigma);
    }
}
```

Sleep deprivation study: posterior predictive distribution

```
generated quantities {
        vector[N] R_sim; ## Store post-pred samples
        for (i in 1:N){
        R_sim[i] = normal_rng(X[i] * gamma, sigma);
    }
}
```

The function `normal_rng` generates a single **independent** sample from a normal distribution with parameters:

- mean $= X[i] * gamma$, where *gamma* is a sample from the estimated posterior.
- std. dev $=$ *sigma*, where *sigma* is a sample from the estimated posterior.

Sleep deprivation study: model changes

- Suppose based on posterior predictive checks we want to use a wider sampling distribution $\implies$ use a Student T.
- If we were coding this up ourselves this would involve a large structural change in the code and MCMC algorithm.

Question: how long does it take us to recode our model in Stan?

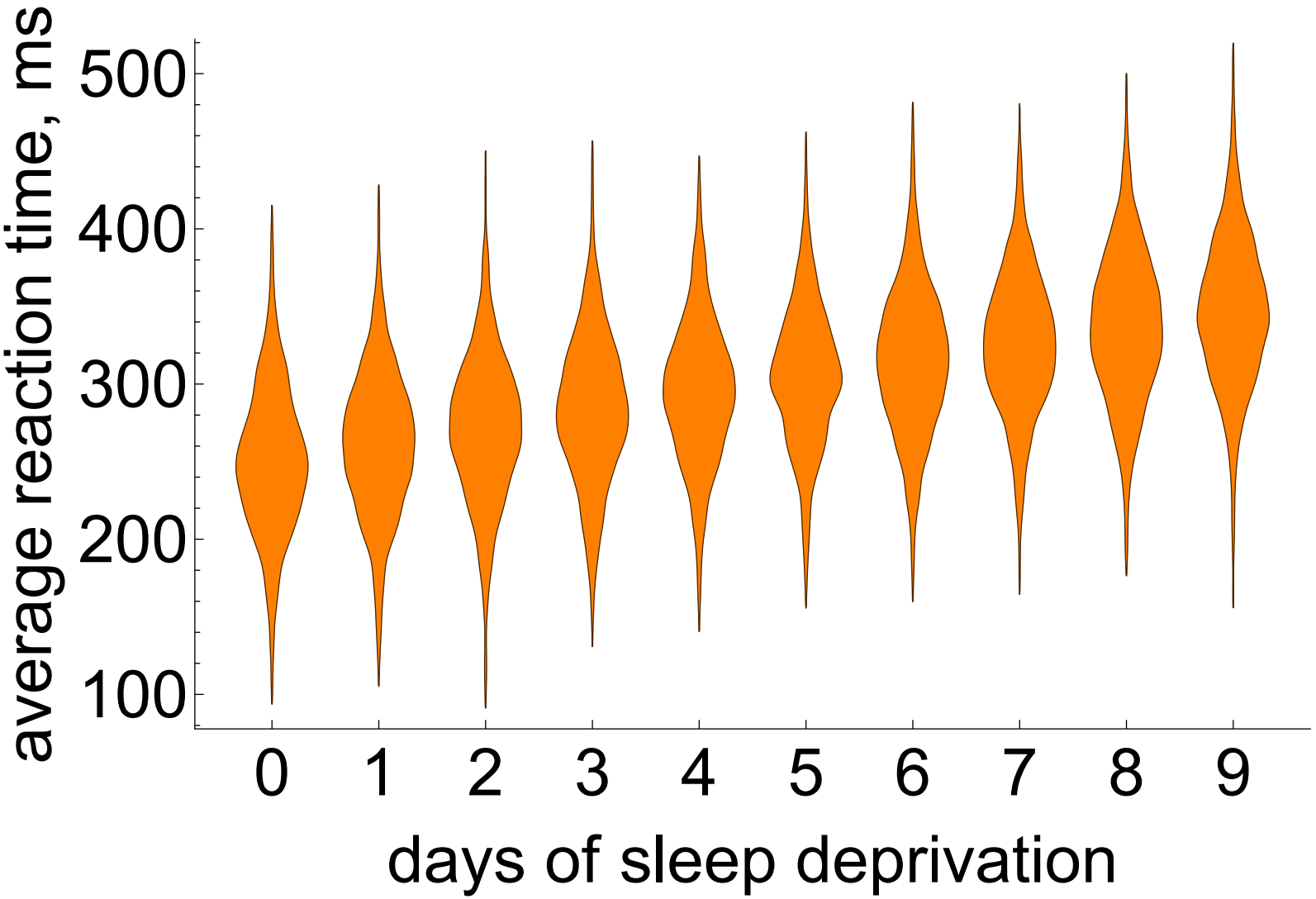
Sleep deprivation study: model changes

```
parameters {
    ...
    real<lower=0> nu;
}
model {
    ## likelihood
    R ~ student_t(nu, X * gamma, sigma);
    ...
    nu ~ gamma(1,1);
}
```

$\Longrightarrow$ three changes/additions necessary.

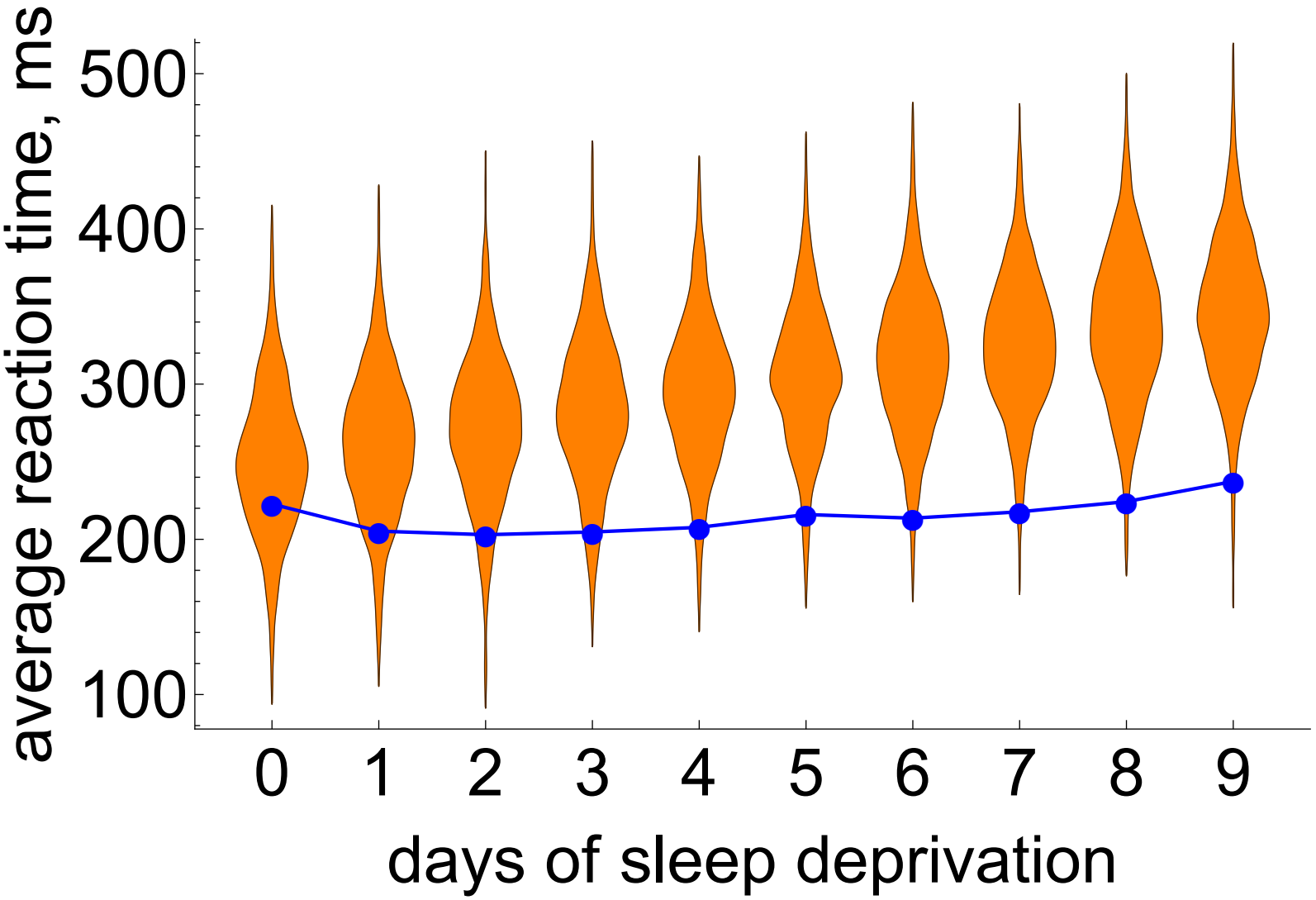
Sleep deprivation: posterior predictive checks

Posterior predictive distribution for Student T sampling.



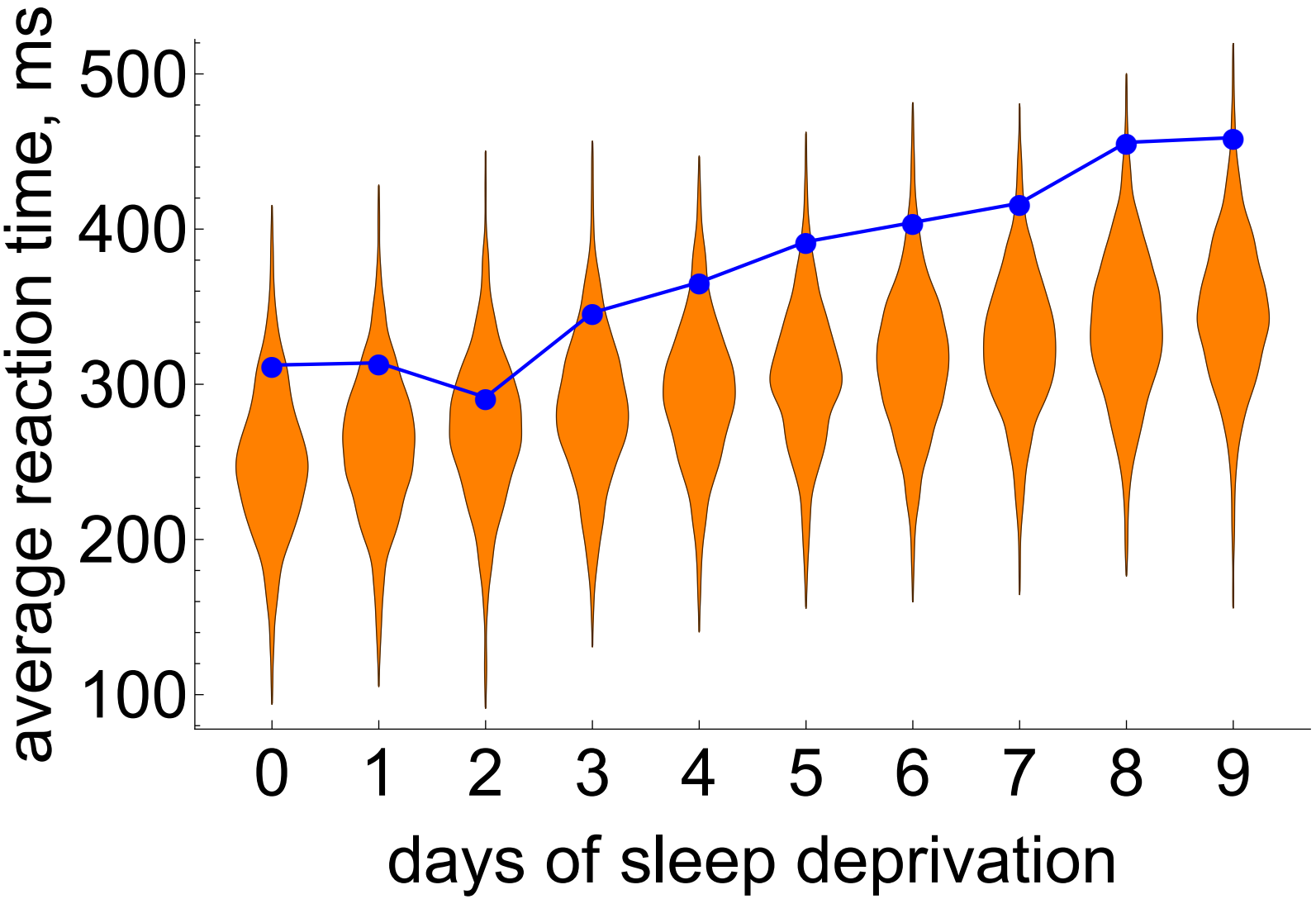
Sleep deprivation: posterior predictive checks

One participant.



Sleep deprivation: posterior predictive checks

And another.



Sleep deprivation study: model changes

Moving to a Student T distribution did not solve individual fit issues $\Longrightarrow$ try a more ambitious change:

- Go back to normal sampling model.
- Allow each subject their own response parameters.
- $\Longrightarrow$ parameters are now 18-dimensional; one per each of the 18 subjects.
- (A better way to do this is with hierarchical models; more on this later.)

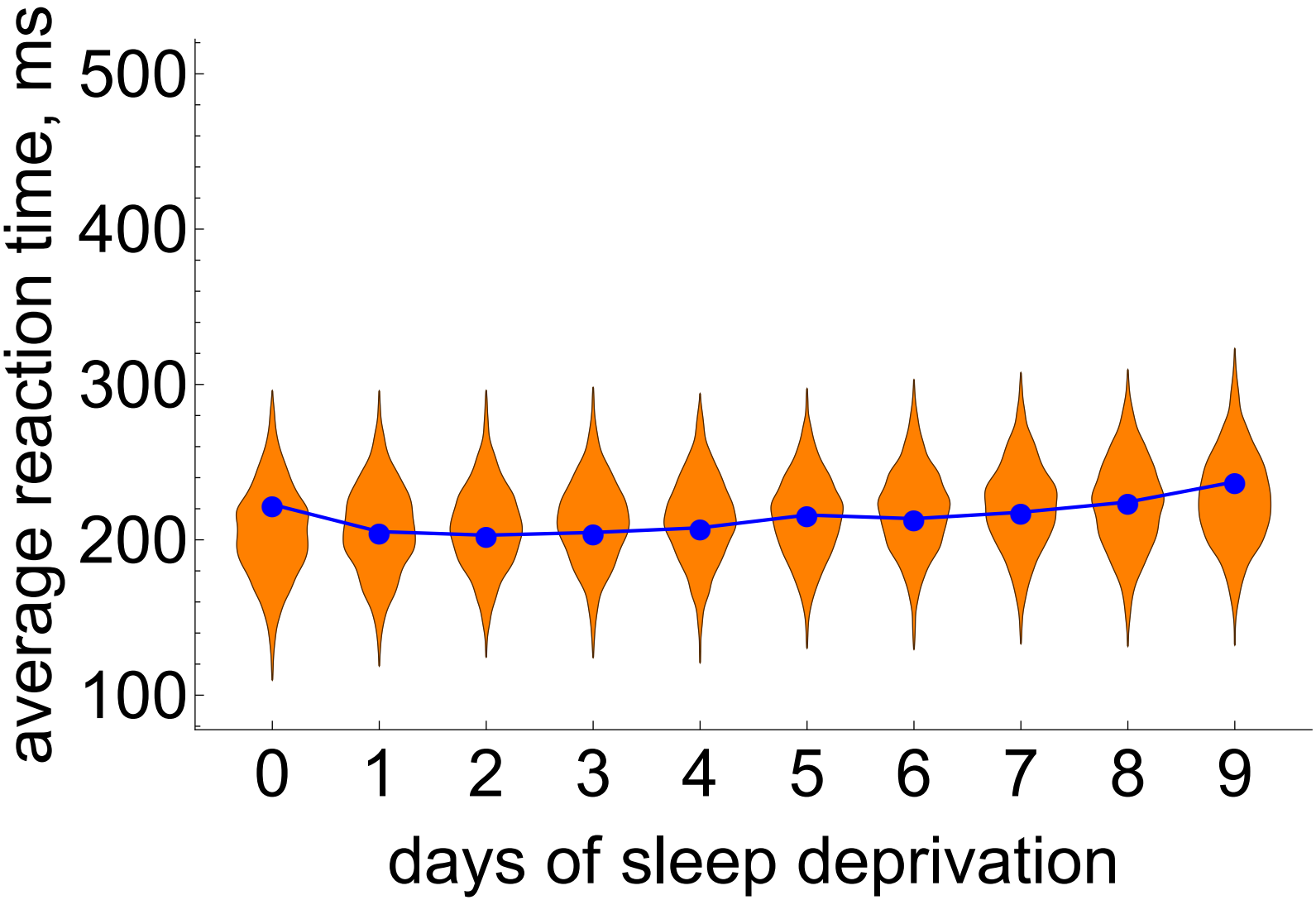
Question: how long does this modification take us?

Sleep deprivation study: model changes

```
data {
    ...
    ## array of subject ids: 1, 2,..,18
    int subject[N];
}
parameters {
    ...
    real alpha[18]; ## 18 elements of each param
    real beta[18]; ## one for each subject
}
model {
    for (i in 1:N){
        R[i] ~ normal(alpha[subject[i]] +
                      beta[subject[i]] * t[i], sigma);
    }
}
```

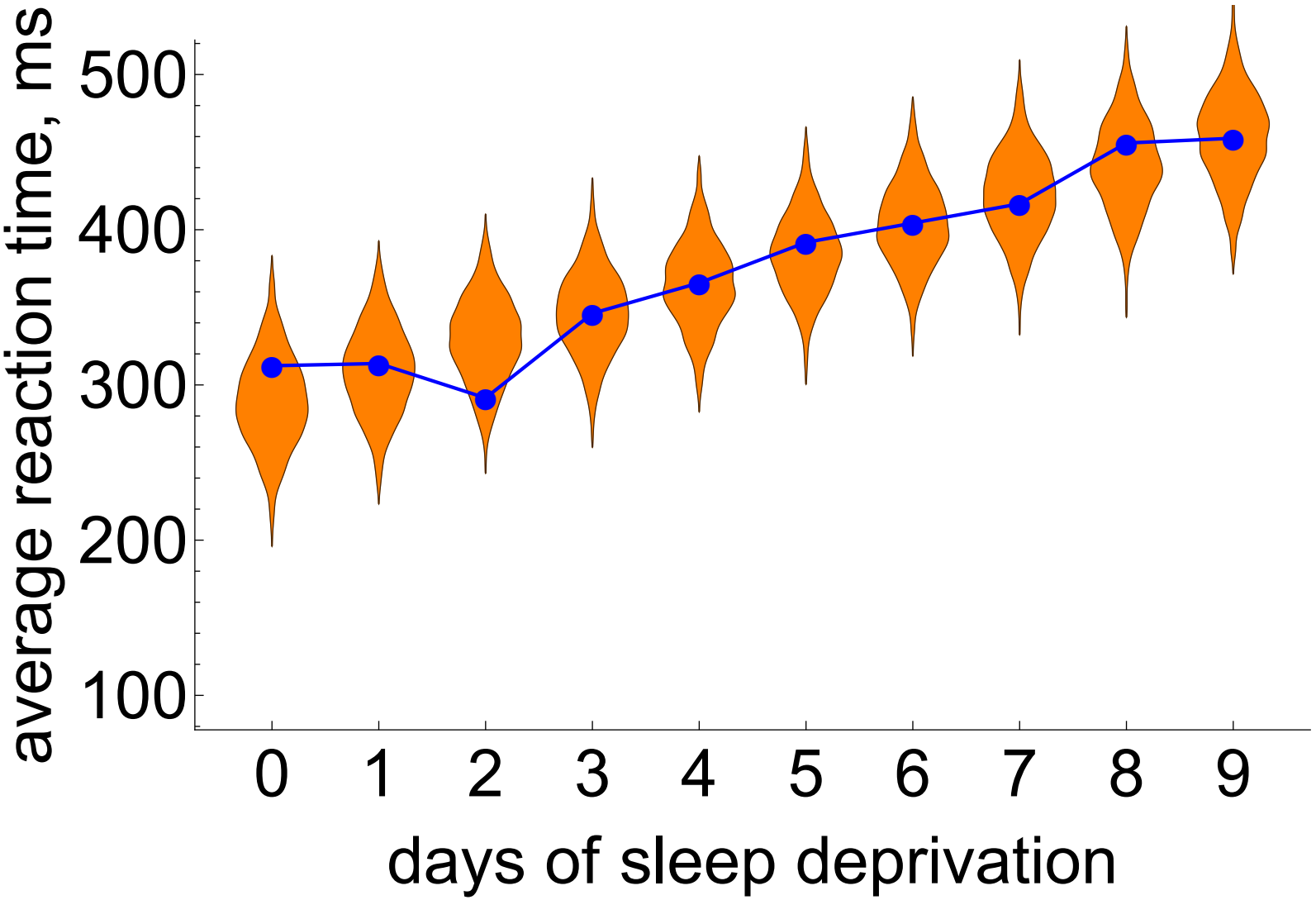
Sleep deprivation: posterior predictive checks

First problematic subject.



Sleep deprivation: posterior predictive checks

Other problematic subject.



Stan: a few of the loops and conditions

Stan has pretty much the full range of language constructs to allow pretty much any model to be coded.

```
for (i in 1:10) {something;}

while (i > 1) {something;}

if(i > 1) {something 1;}
else if (i==0) {something2;}
else {something 3;}
```

Note: this is not the case with JAGS/BUGS due to their **declarative** nature.

Stan: speed concerns

Whilst Stan is fast it pays to know the importance of each code block for efficiency.

- **data** - called once at beginning of execution.
- **parameters** - every log probability evaluation!
- **model** - every log probability evaluation!
- **generated quantities** - once per sample.

Remaining code blocks

Thus far focussed on the four main code blocks, but others exist:

- **transformed data** - carry out a data transformation in Stan; executed once after data.
- **transformed parameters** - often easier to work with transformations of original parameters; executed every log probability evaluation!
- **functions** - allows user-defined functions, must go at top of Stan program. How many times it is called depends on the function's nature. Make it possible to use any density whose log pdf can be written down (see problem set)!

Stan in parallel

In R can run chains in parallel easily using:

```
# smaller font size for chunks
library(rstan)
options(mc.cores=8)
```

Stan summary

- Stan works by default with a HMC-like algorithm called NUTS.
- The Stan language is similar in nature to other common languages with loops, conditional statements and user-definable functions (didn't cover here).
- Stan makes life easier for us than coding up the MCMC algorithms ourselves.
- Stan is typically multiple-times faster than BUGS/JAGS for generating effective samples.

How to debug a Stan model?

Two issue flavours, each with their own response.

- Coding errors.
- Sampling issues.

Coding errors

Stan error messages are generally quite informative however inevitably there are times when it is less clear why code fails

$\implies$ debug by print!

```
model {
    ...
    print(theta);
}
```

In R this prints (neatly) to the console output.

Coding errors

Important: failing a resolution via the above go to `http://mc-stan.org/` and do:

1. Look through manual for a solution.
2. Look through user forum for previous answers to similar problems.
3. Ask a question; be clear, and thorough - post as simple a model that replicates the issue.
4. Ask me!

Sampling issues

Different sort of issue to a coding error, and falls into two (often related) issues:

- **Slow convergence:** still have $\hat{R} > 1.1$ after many thousands of iterations.
- **Divergent iterations:** get a warning in output from Stan with the number of iterations where the NUTS sampler has terminated prematurely.

Sampling issues

Most important thing today: Gelman's "Folk Theorem":

"Issues with computational sampling are almost always due to problems with the underlying statistical model, **not** the algorithm."

Sampling issues: slow convergence

- Poor chain mixing is usually due to lack of **parameter identification**.
- A parameter is identified if it has some unique effect on the data generating process that can be separated from the effect of the other parameters.

Solution (very important): use **fake data** where you know the true parameter values.

$\Longrightarrow$ informative as to whether the data + priors are sufficient to estimate a parameter's value.
If possible use the most simple version of your model that replicates the error.

Sampling issues: slow convergence

A significant number of divergent transitions of NUTS are problematic:

- Indicates that the stepwise integrator used to approximate Hamiltonian dynamics has likely diverged from exact trajectory.
- Therefore these samples **cannot** be viewed as being from the posterior.
- Causes a bias away from problem area of parameter space.
- Almost always because the step size is too large relative to the curvature of the posterior.
- However can be due to placing limits on parameters that preclude an area of high probability mass.

Diagnosing problem: use Shiny Stan (or otherwise) to make pairwise plots of variables $\Longrightarrow$ look for parameters with high bivariate correlation; indicates high curvature.

Sampling issues: slow convergence

Solution: if significant number of divergent iterations do the following:

1. Lower step size and increase acceptance rate in the call to Stan from R or otherwise.
2. If above doesn't help change priors then likelihood.

Again failing all the above look at the Stan user forums, then ask a question.

What to do when things go wrong: summary

- To debug a model that fails read error messages carefully, then try "print" statements.
- Problems with sampling are almost invariably problems with the underlying model **not** the sampling algorithm per se.
- Use fake data with all models to test for parameter identification (and that you've coded up correctly.)
- Stan has an active developer and user forum, great documentation, and an extensive answer bank.
- If you ask a question on the forum include your model, or ideally a simplified version that replicates the issue.

Lecture summary

- Hamiltonian Monte Carlo lets posterior geometry determine proposals by simulating movement of a puck in NLP space $\Longrightarrow$ get high proportion of acceptances, and fast exploration of posterior space.
- Stan carries out inference by a variant of HMC called NUTS $\Longrightarrow$ very fast!
- Stan is (hopefully) a fairly intuitive language to learn.
- Makes MCMC sampling easier for a wide class of problems in Bayesian inference.
- If things go wrong $\Longrightarrow$ there is hope!

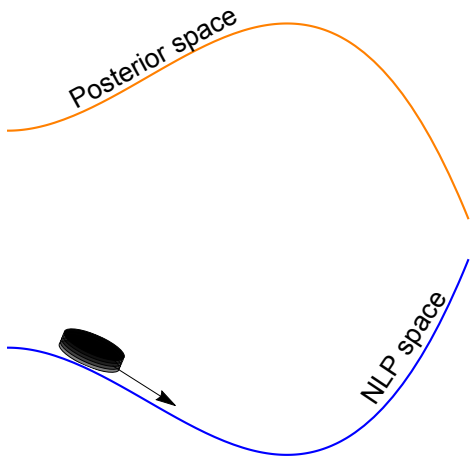
Reading list

Only big chunks this week.

- Chapters 11 (basic MCMC) and 12 (advanced MCMC) in "Bayesian data analysis", by Gelman et al. (2014), 3rd edition.
- Chapters 7 (MCMC) and 14 (HMC and Stan) in "Doing Bayesian data analysis", by Kruschke (2015), 2nd edition.
- Chapter 8 (MCMC) in "Statistical Rethinking", by McElreath (2016).
- Chapter 5 (HMC) by Neal, in "Handbook of Markov Chain Monte Carlo", edited by Brooks et al. (2011).

Not sure I understand?

Hamiltonian Monte Carlo.



Hamilton in Monte Carlo.