Lecture 6: speaking Stan and hierarchical models

Ben Lambert[1]
ben.lambert@some.ox.ac.uk

[1]Somerville College
University of Oxford

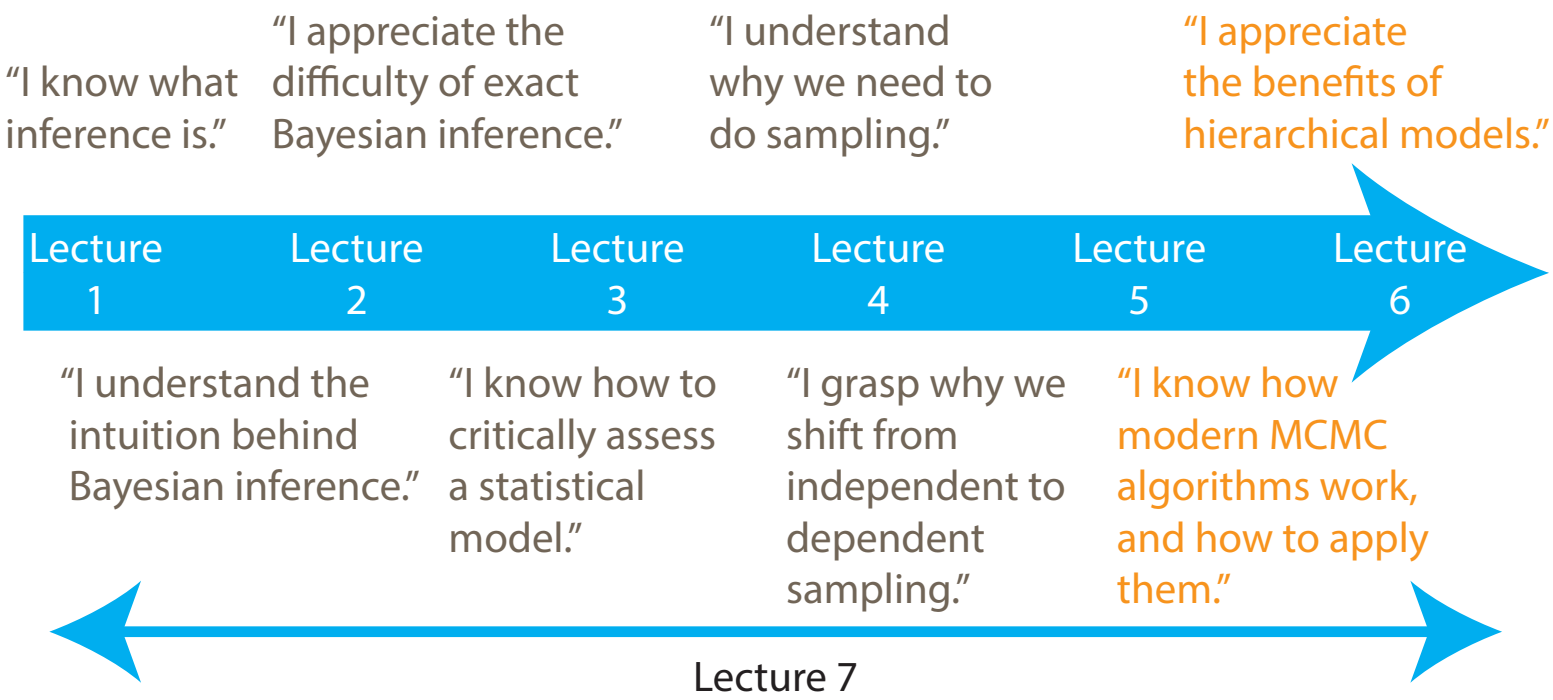
June 14, 2016

Lecture outcomes

By the end of this lecture you should:

1. Know how to start coding up a model in Stan.
2. Appreciate how easy Stan makes things for us compared to coding up the algorithm ourselves.
3. Know what to do when coding goes wrong.
4. Know what to do when sampling goes wrong.
5. Understand what is meant by a hierarchical model.
6. Appreciate the benefits of hierarchical models versus completely-pooled or completely-heterogeneous models.

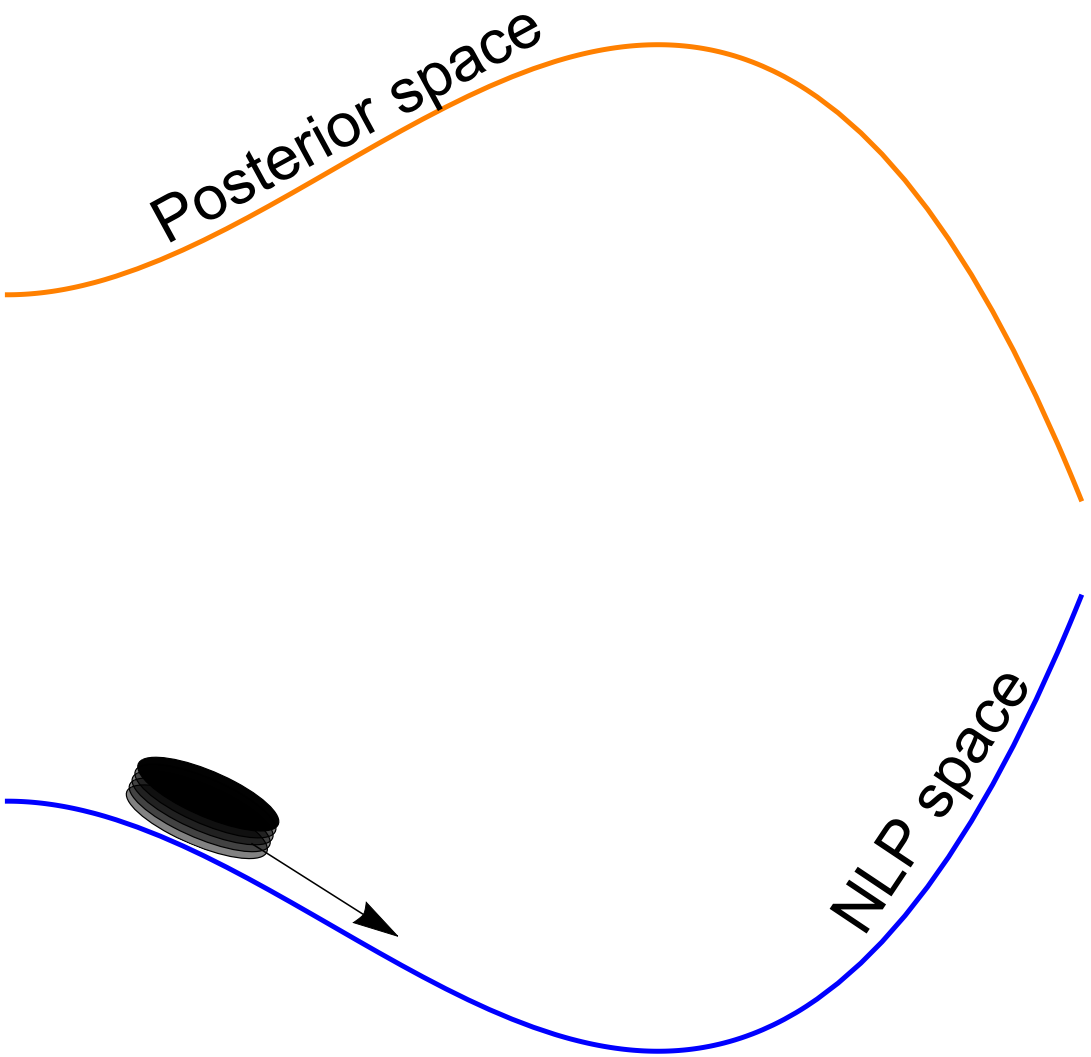
Overall course outline



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?

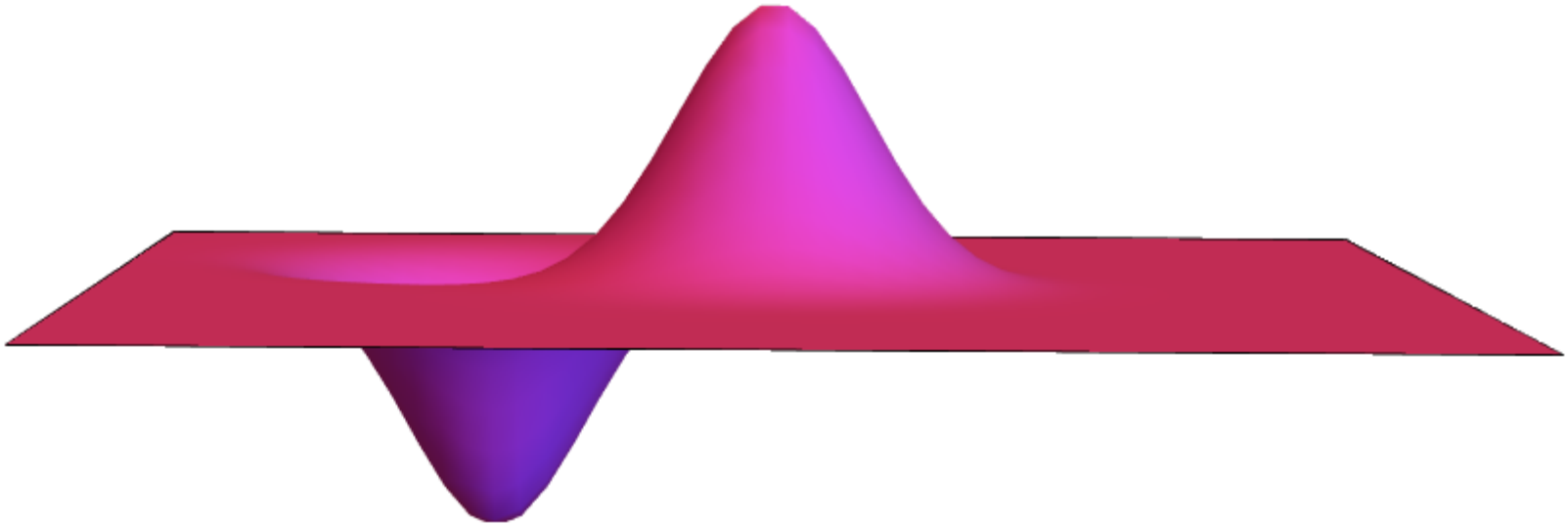


Why does this physical analogy help us?

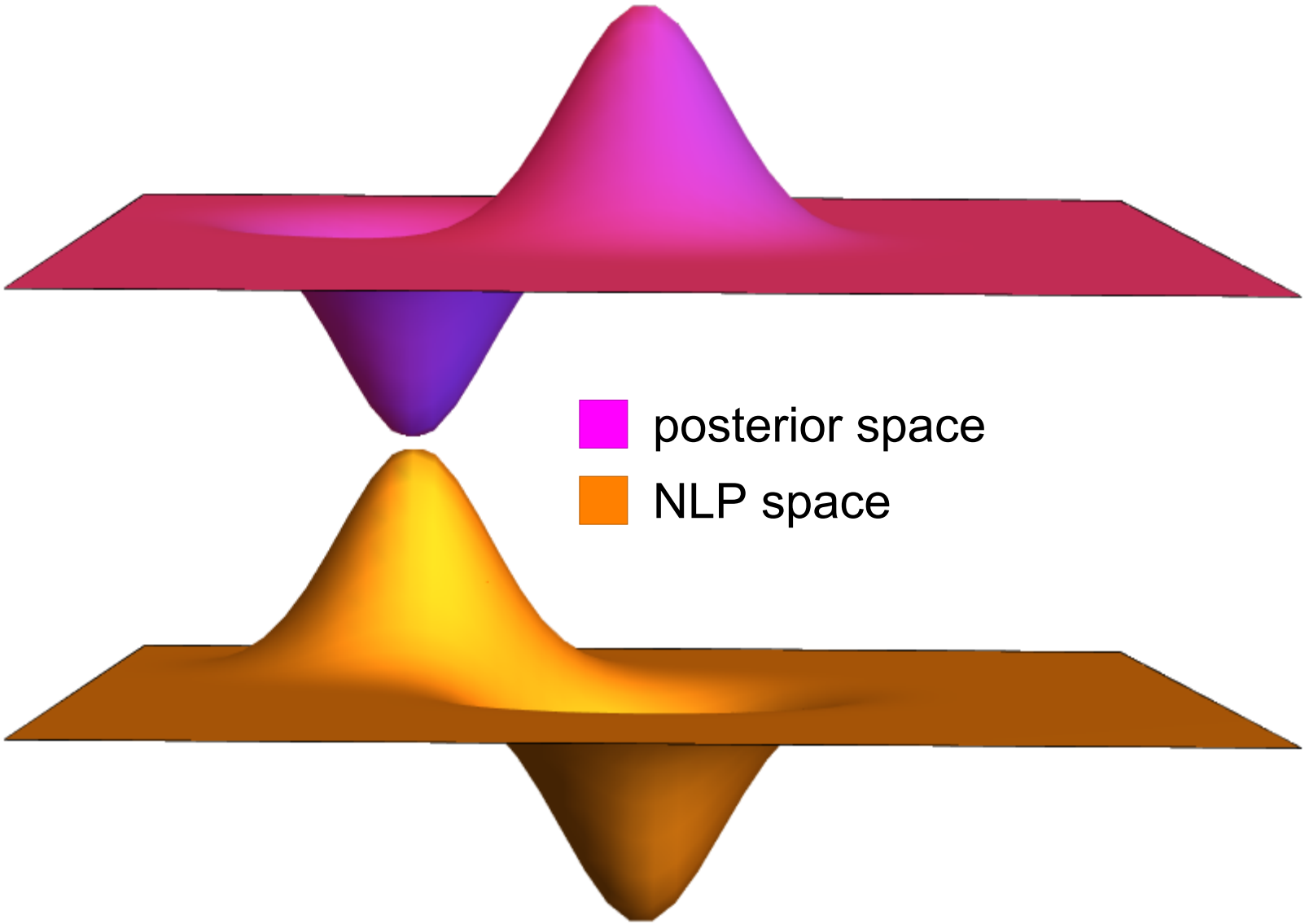
- Allow the potential energy of the puck to be determined partly by the posterior density.
- $\Longrightarrow$ puck will move in the "natural" directions dictated by the posterior geometry.
- And will visit areas of low NLP $\Longrightarrow$ high posterior density.
- **NLP** stands for the **negative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

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.)

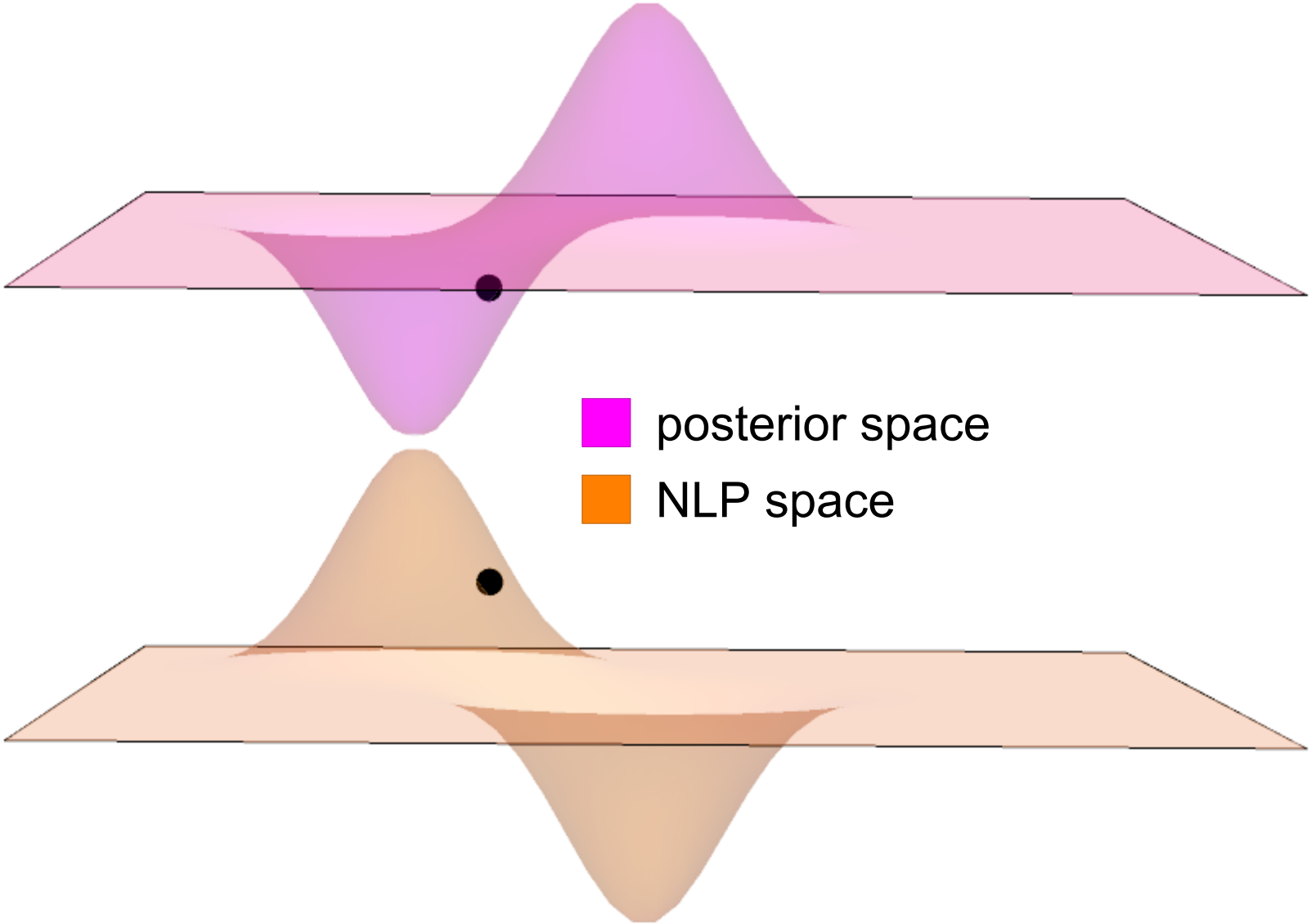
Simulating the puck's motion in NLP space: start with a posterior



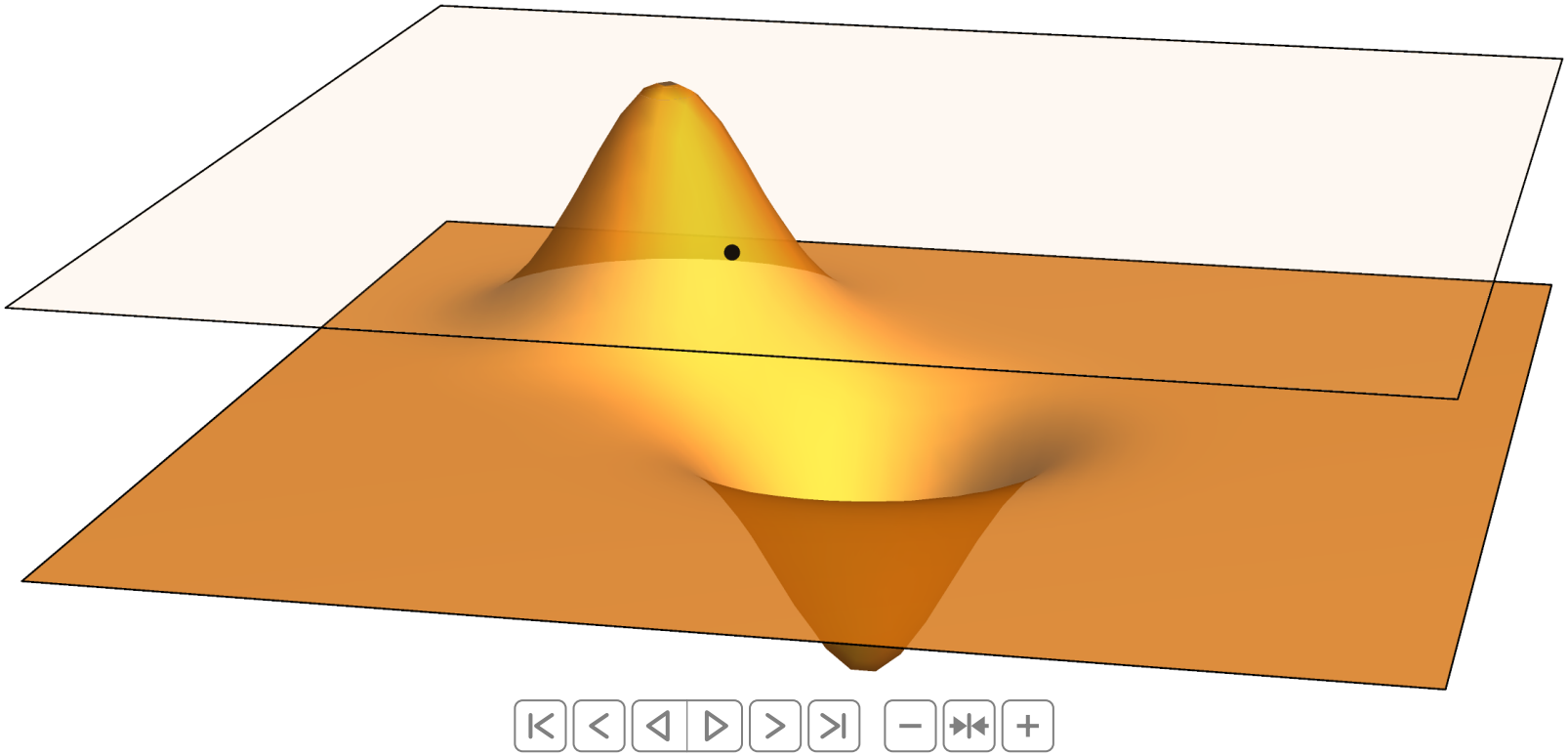
Simulating the puck's motion in NLP space: find NLP space



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



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



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:

$$Y_i \sim N(\mu, \sigma) \tag{1}$$

- 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:

$$p(\beta) = \sum_{i=1}^{K} p(\beta, \theta_i) \tag{2}$$

- Most 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

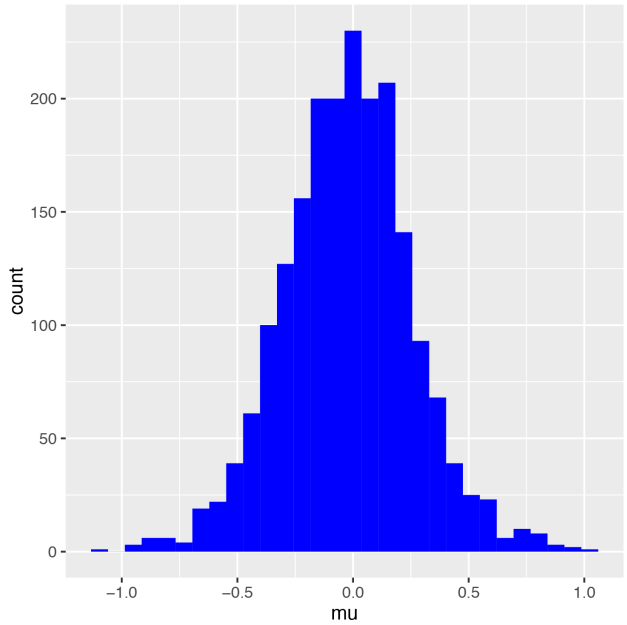
```
## Print summary statistics
print(fit,probs = c(0.25, 0.5, 0.75))

## 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
```

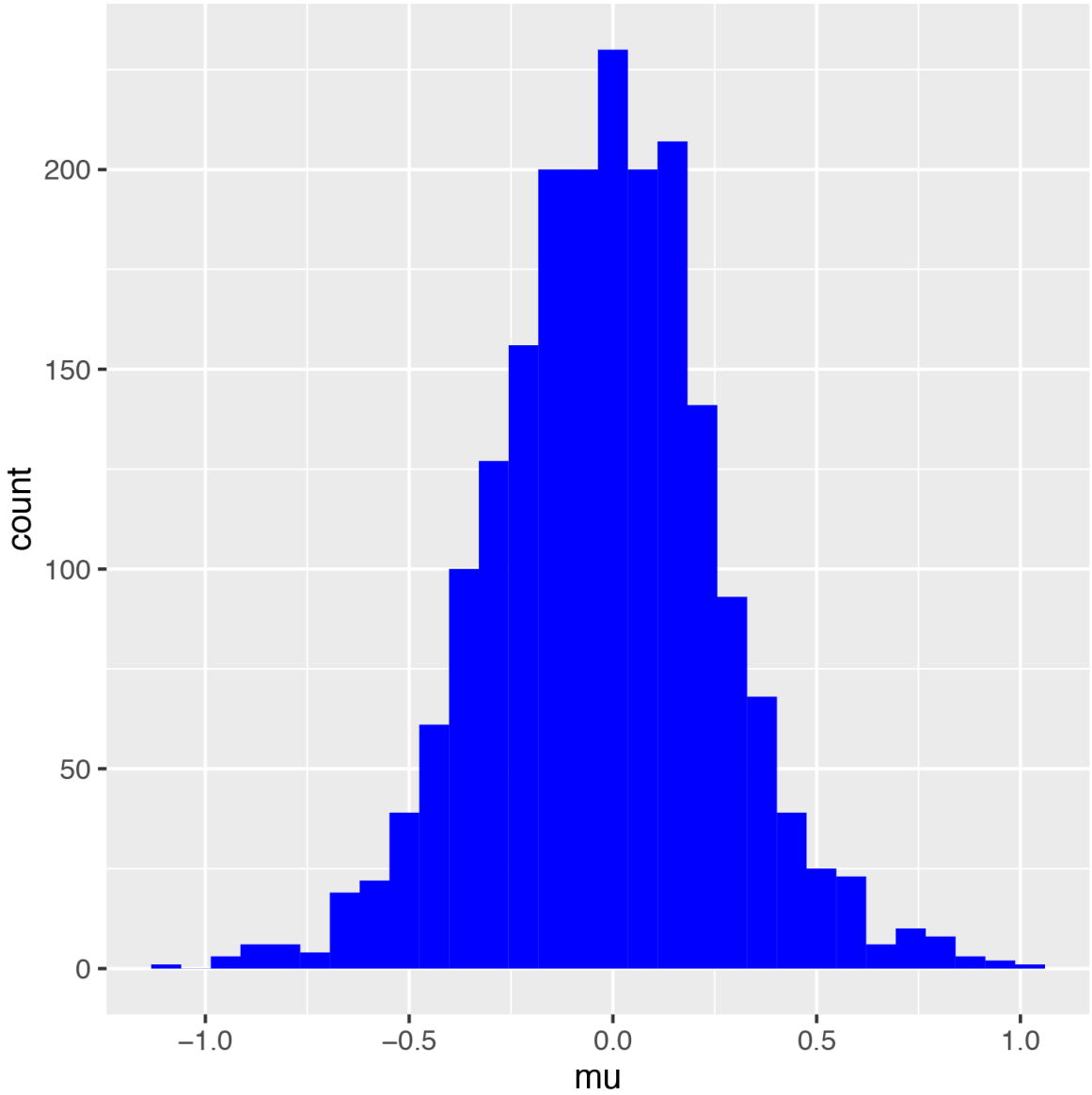
Example model: results

```
## 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
}
```

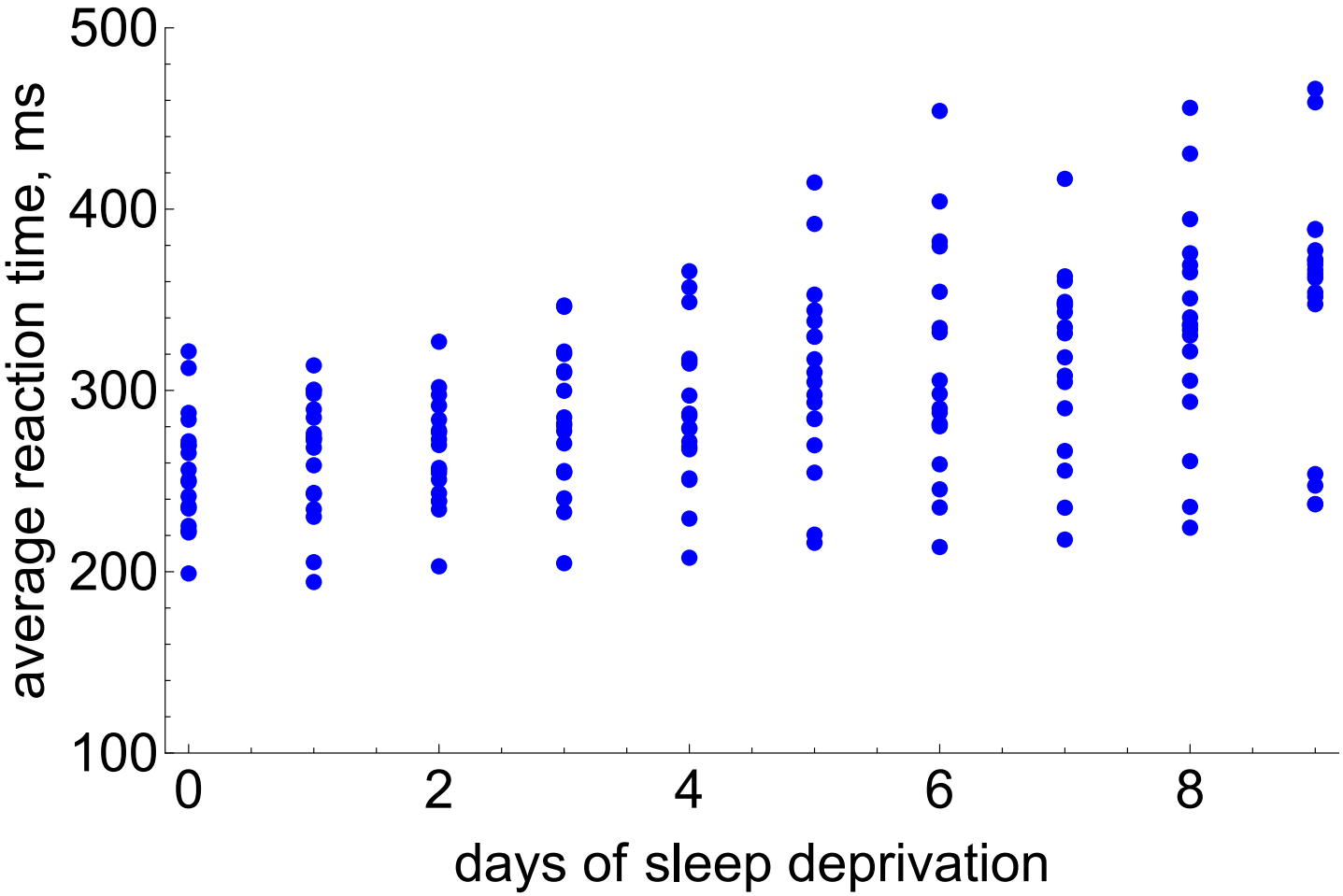
- $\sim$ doesn't mean "sampling" although for many circumstances it isn't terrible to think of it like that.
- Remember: HMC/NUTS uses the negative of the log-posterior to find potential energy of puck.
- $\implies$ logging the posterior converts it from a product into a sum.
- As such $\sim$ really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

$$R(t) \sim N(\alpha + \beta t, \sigma) \tag{3}$$

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);
    }
}
```

Important: "<-" is the assignment operator in Stan **not** "=".

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

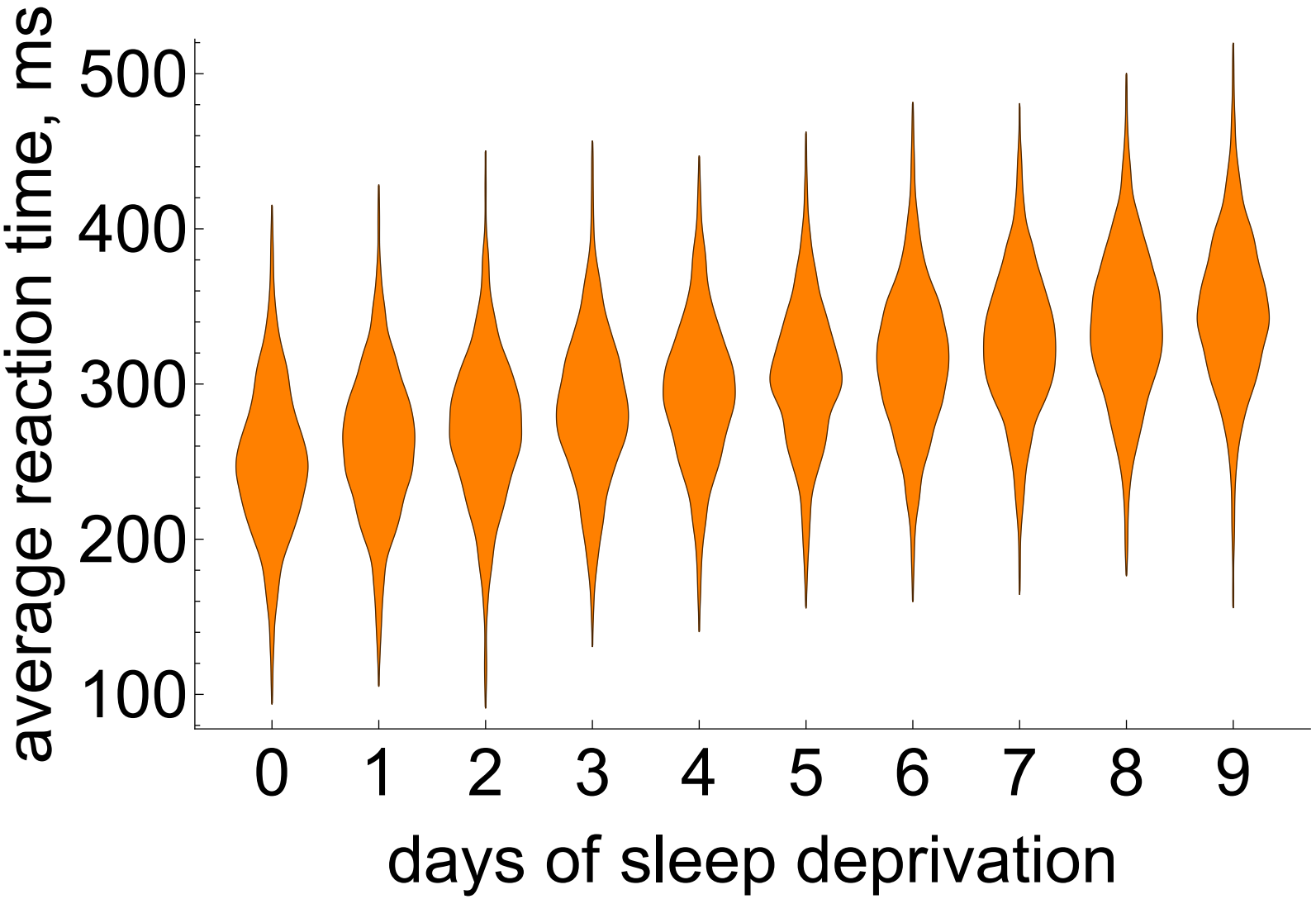
```
parameters {
    ...
    real<lower=0> nu;
}
model {
    ## likelihood
    R ~ student_t(nu, X * gamma, sigma);
    ...
    nu ~ gamma(1,1);
}
```

$\Longrightarrow$ three changes/additions necessary.

Answer: less time than it takes a swan to break your arm. Vicious buggers.

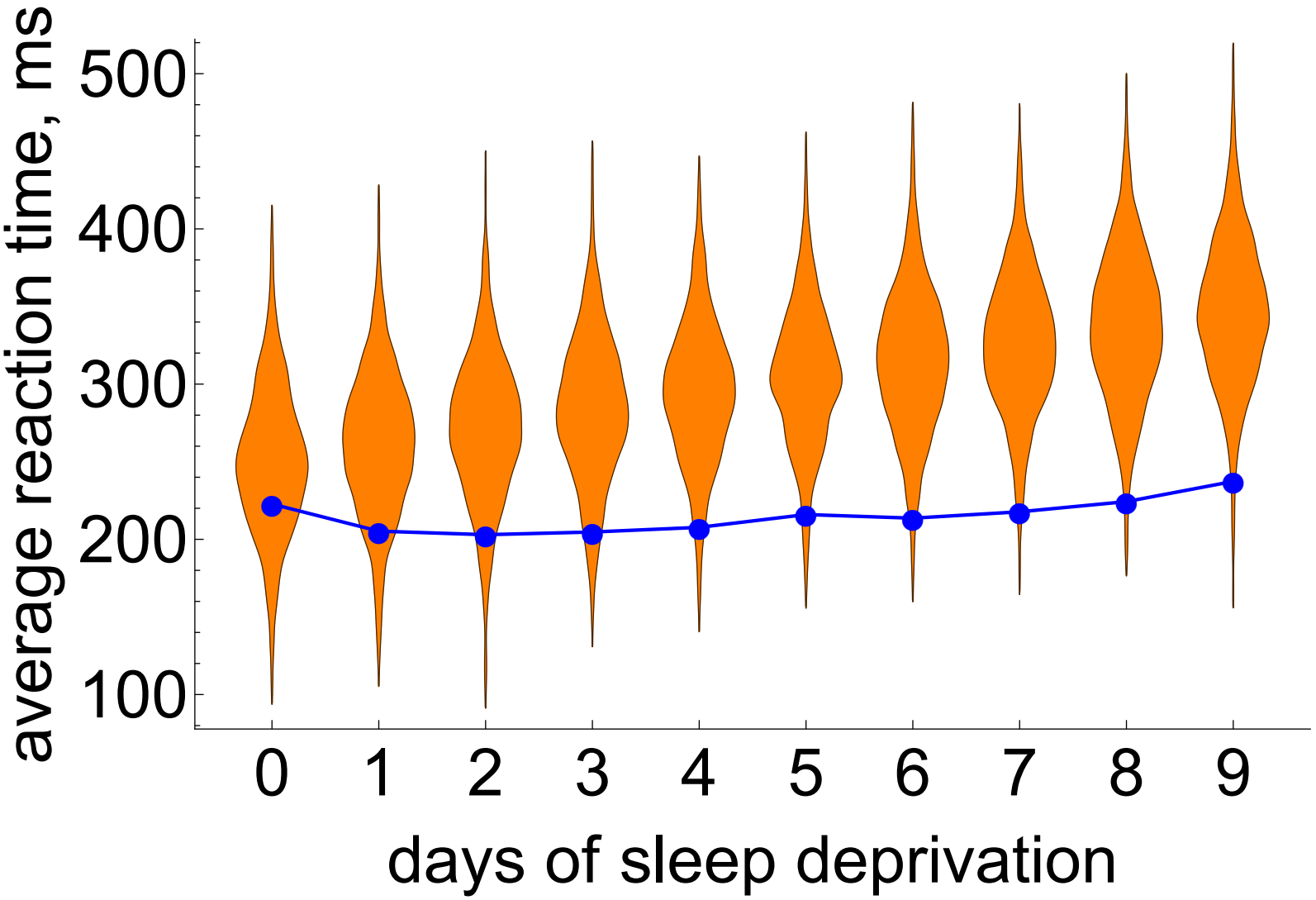
Sleep deprivation: posterior predictive checks

Posterior predictive distribution for Student T sampling.



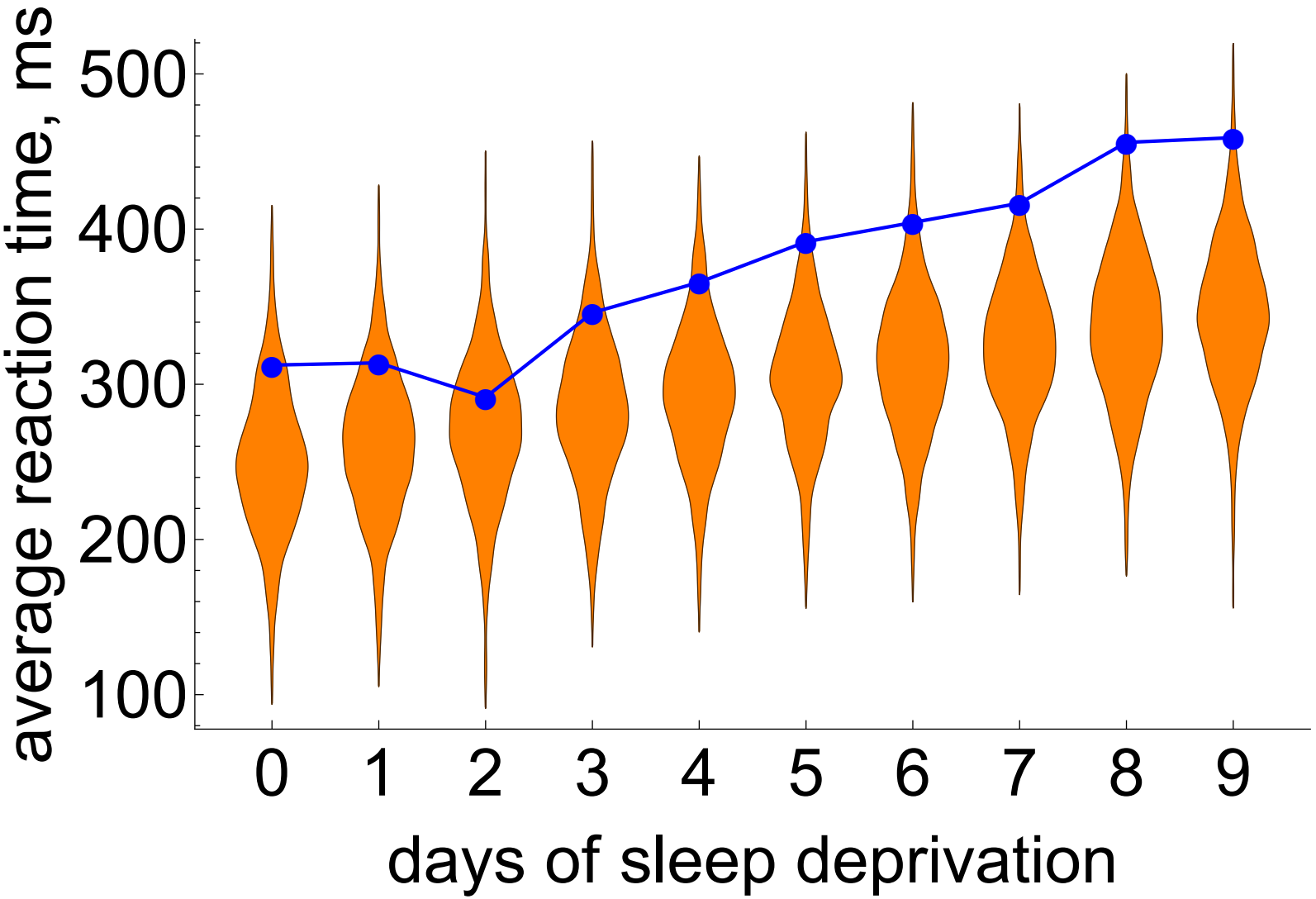
Sleep deprivation: posterior predictive checks

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.)

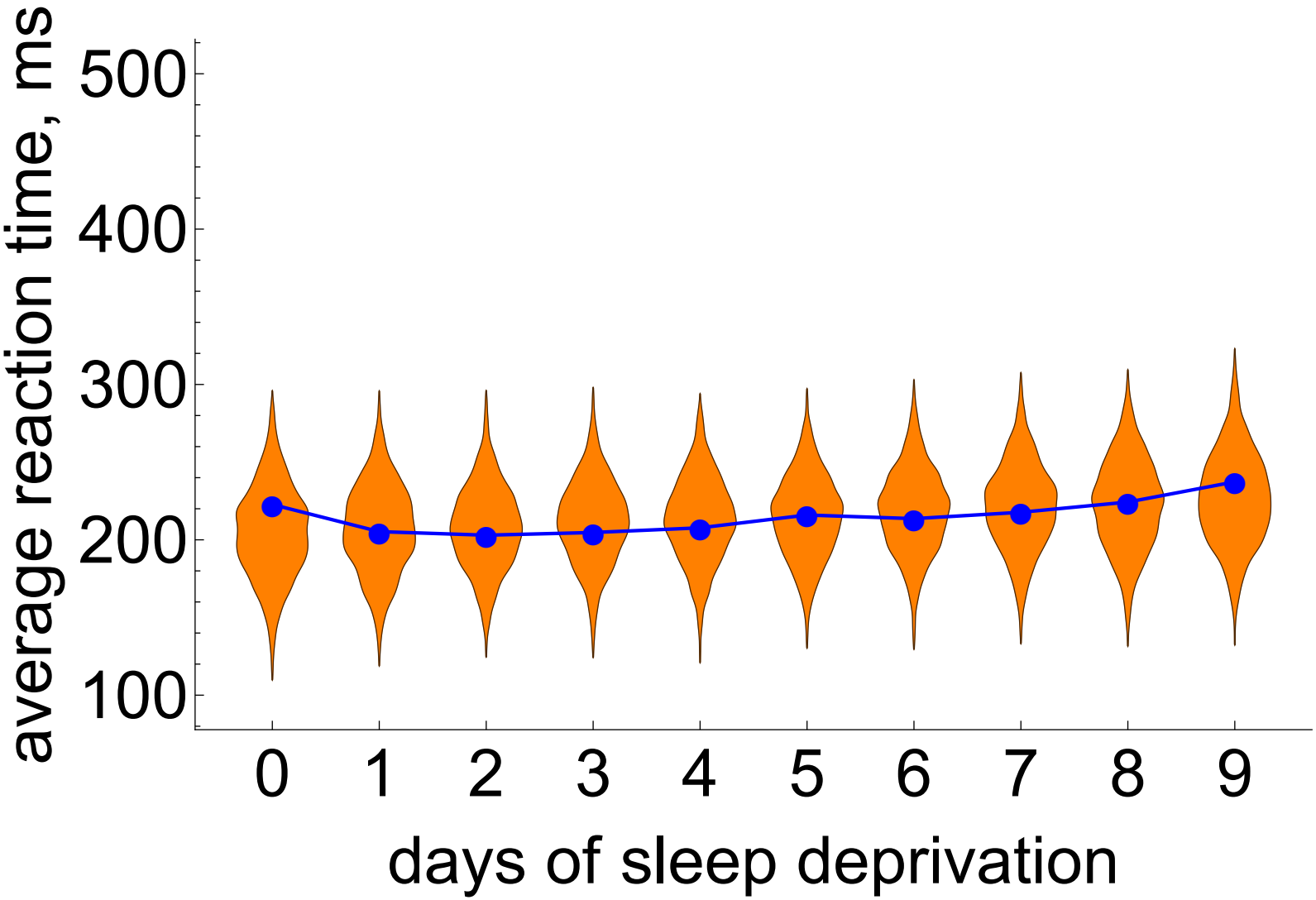
Question: how long does this modification take us?

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);
    }
}
```

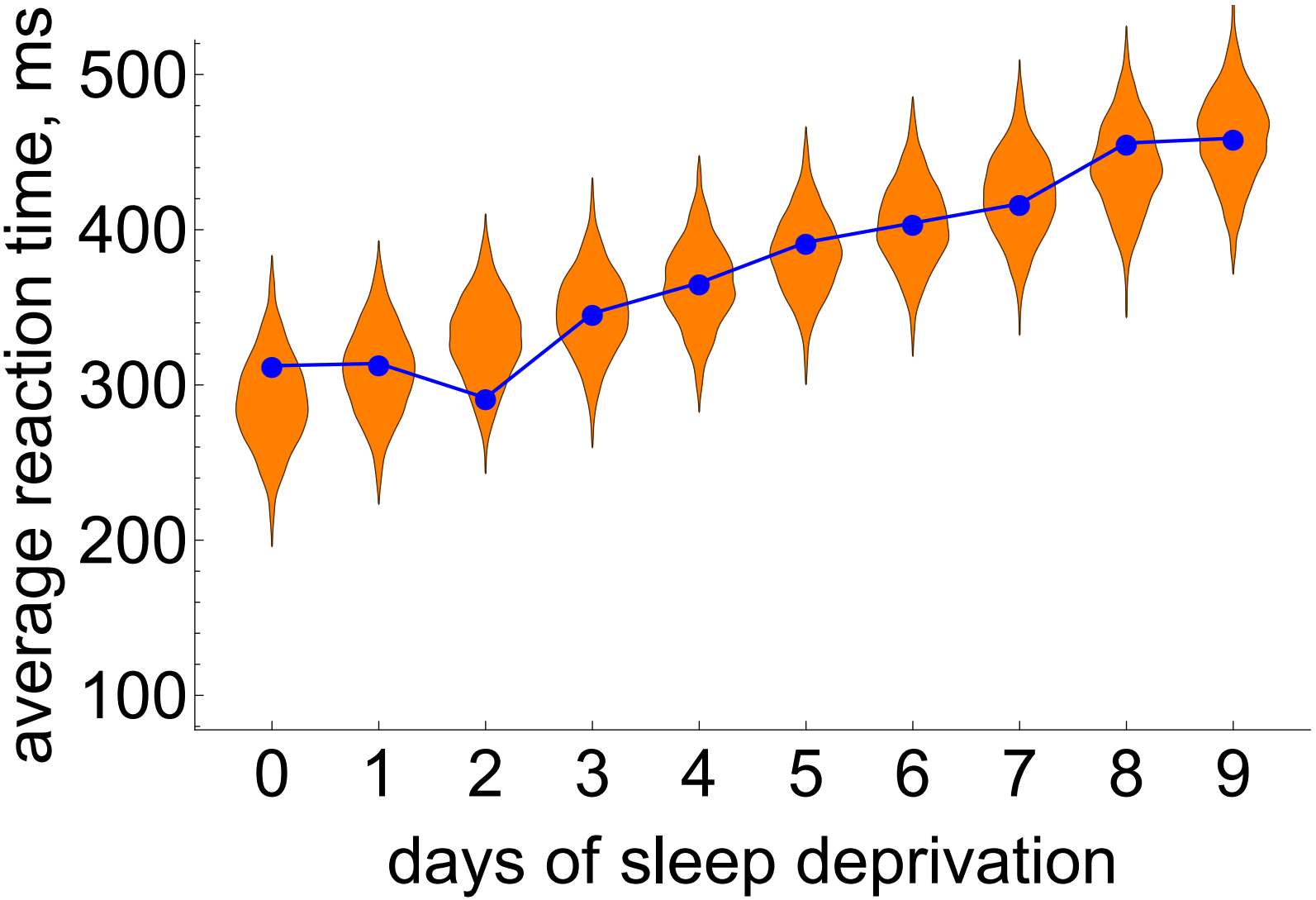
Sleep deprivation: posterior predictive checks

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!

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 places 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.

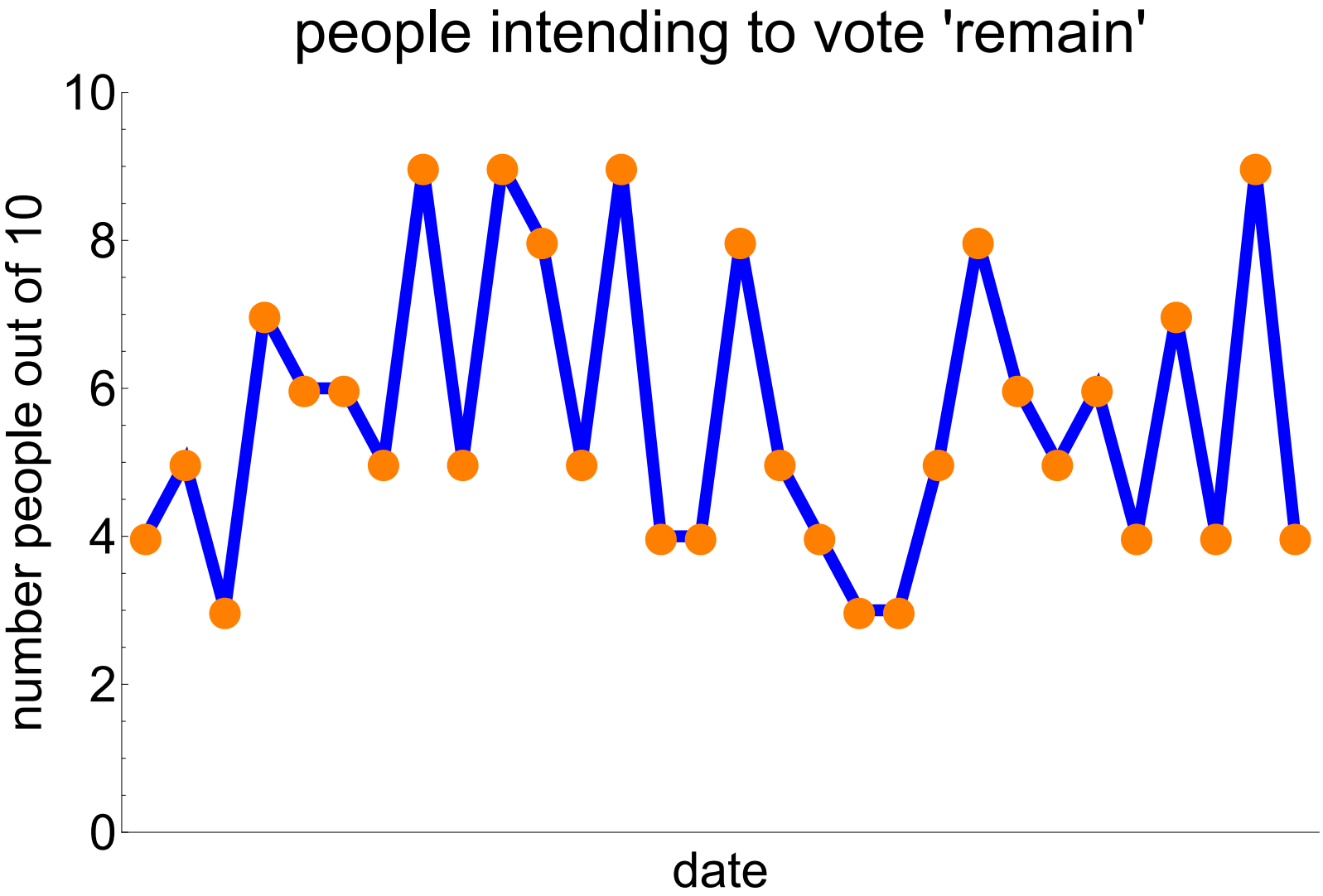
What to do when things go wrong: summary

- 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.

Example: EU referendum

- Imagine a dystopian future where the UK decides to hold a referendum on its membership of the EU.
- Have data from 20 polls on EU membership carried out over the past month (fake data.)
- Polls conducted by a range of different agencies.
- The sample size of each poll is 10.
- Ultimately want to use model to forecast the result of the EU referendum.

EU referendum: data



EU referendum: complete pooling model

Suppose that the we assume that the data across all polls are:

- Independent.
- Identically-distributed.

Sample size is fixed and data are discrete $\implies$ binomial likelihood:

$$Pr(X = X_i|\theta) = \theta^{X_i}(1-\theta)^{10-X_i} \tag{4}$$

where X_i is the number of people voting 'remain' in poll i, and θ is the probability that a randomly-chosen person will vote 'remain'.

Important: we are assuming that θ is the same across all polls.

EU referendum: complete pooling model in Stan

```
data {
    int<lower=1> K; ## number of polls
    int<lower=0> X[K]; ## numbers voting 'remain'
    int<lower=1> N; ## sample size
}
parameters {
    real<lower=0,upper=1> theta;
}
model {
    for (i in 1:K){
        X[i] ~ binomial(N,theta);
    }
}
```

Implicitly $\Longrightarrow$ that we are using a uniform prior on (0,1) for theta.

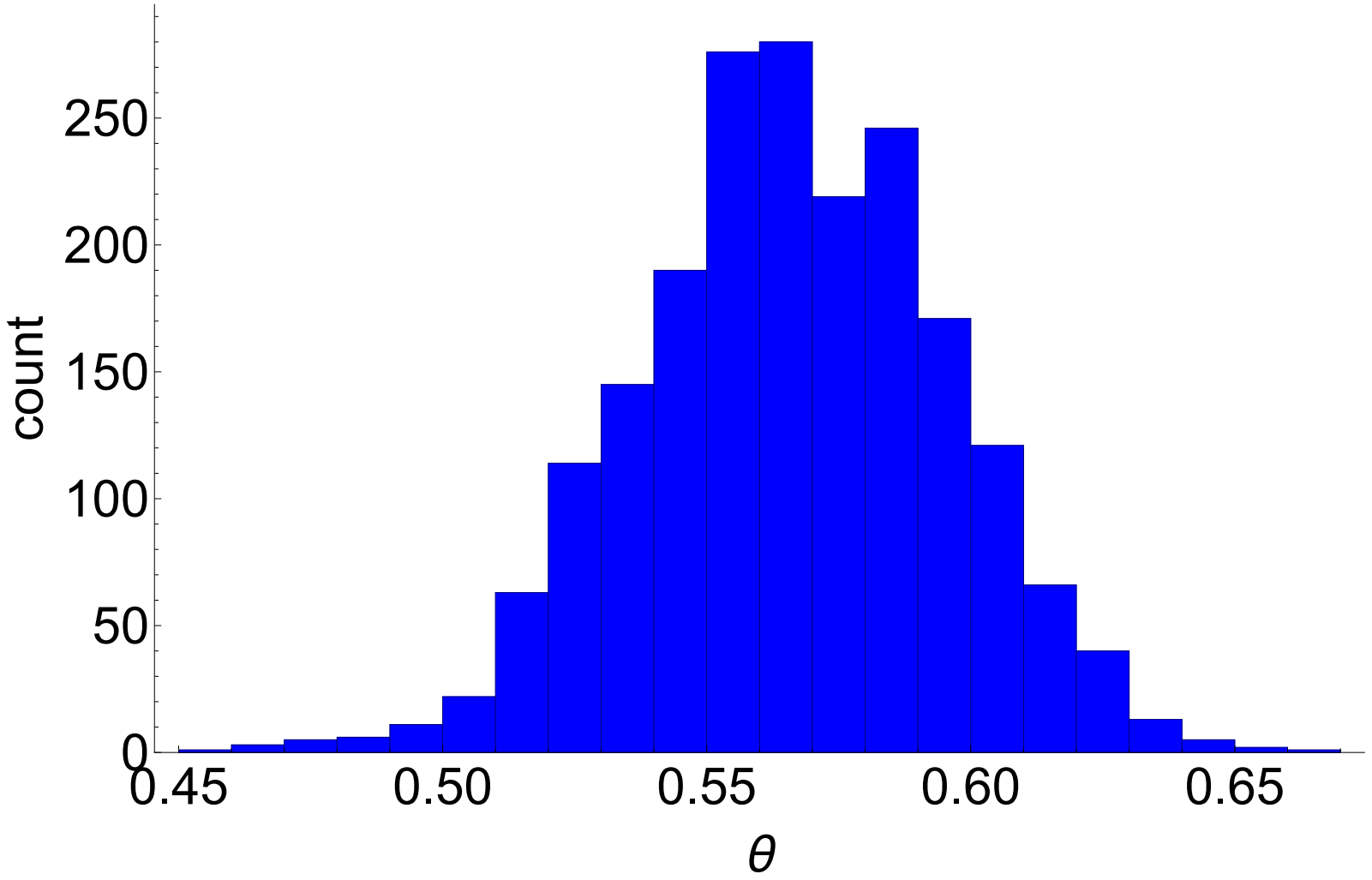
EU referendum: complete pooling model results

```
print(fit,probs = c(0.25, 0.5, 0.75))

## Inference for Stan model: lec6_euBinomialHomgeneousSimple.
## 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
## theta    0.57    0.00 0.03    0.55    0.57    0.59   656 1.01
## lp__  -206.59    0.02 0.65 -206.75 -206.32 -206.17   774 1.00
```

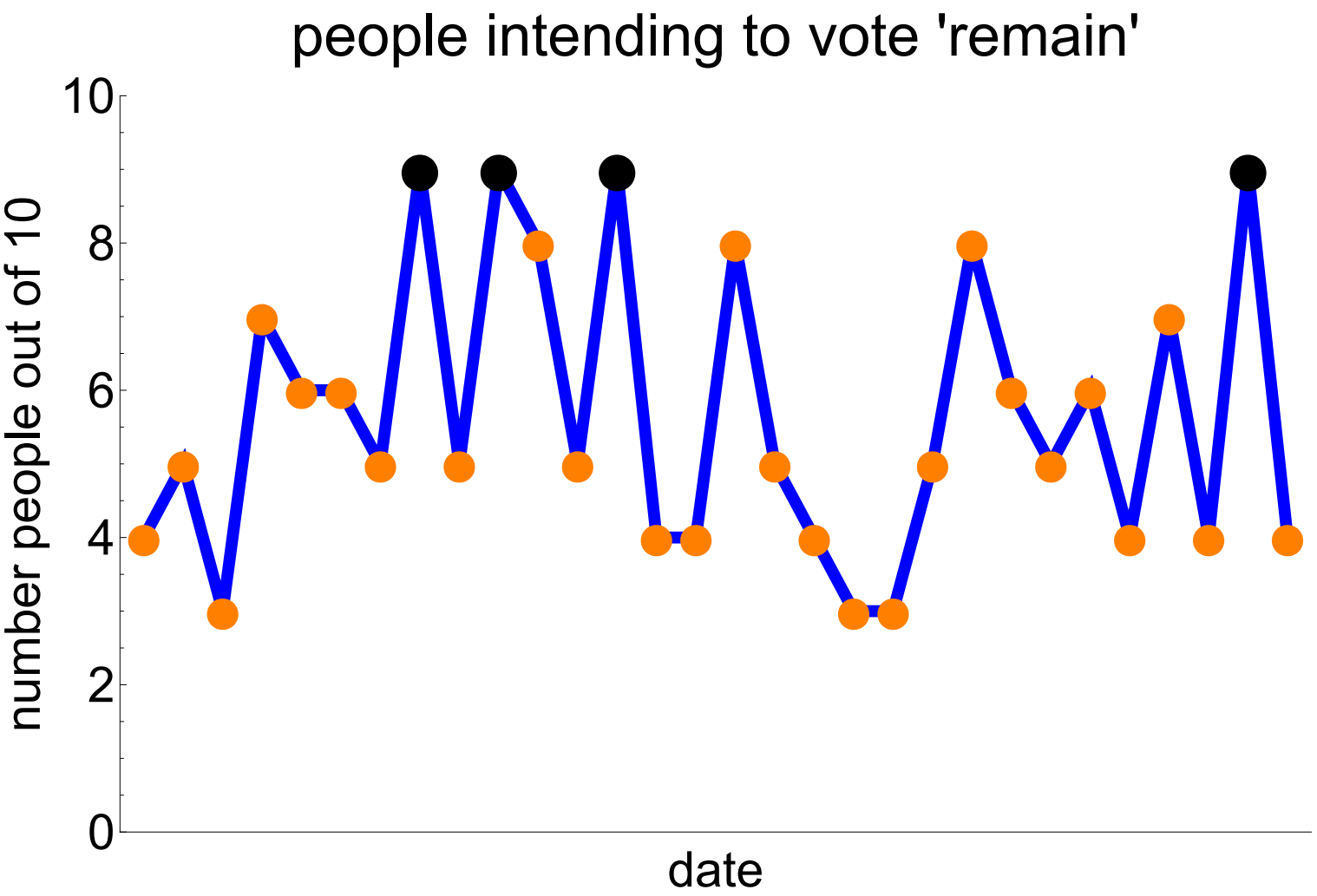
$\Longrightarrow$ all looks ok.

EU referendum: complete pooling model results



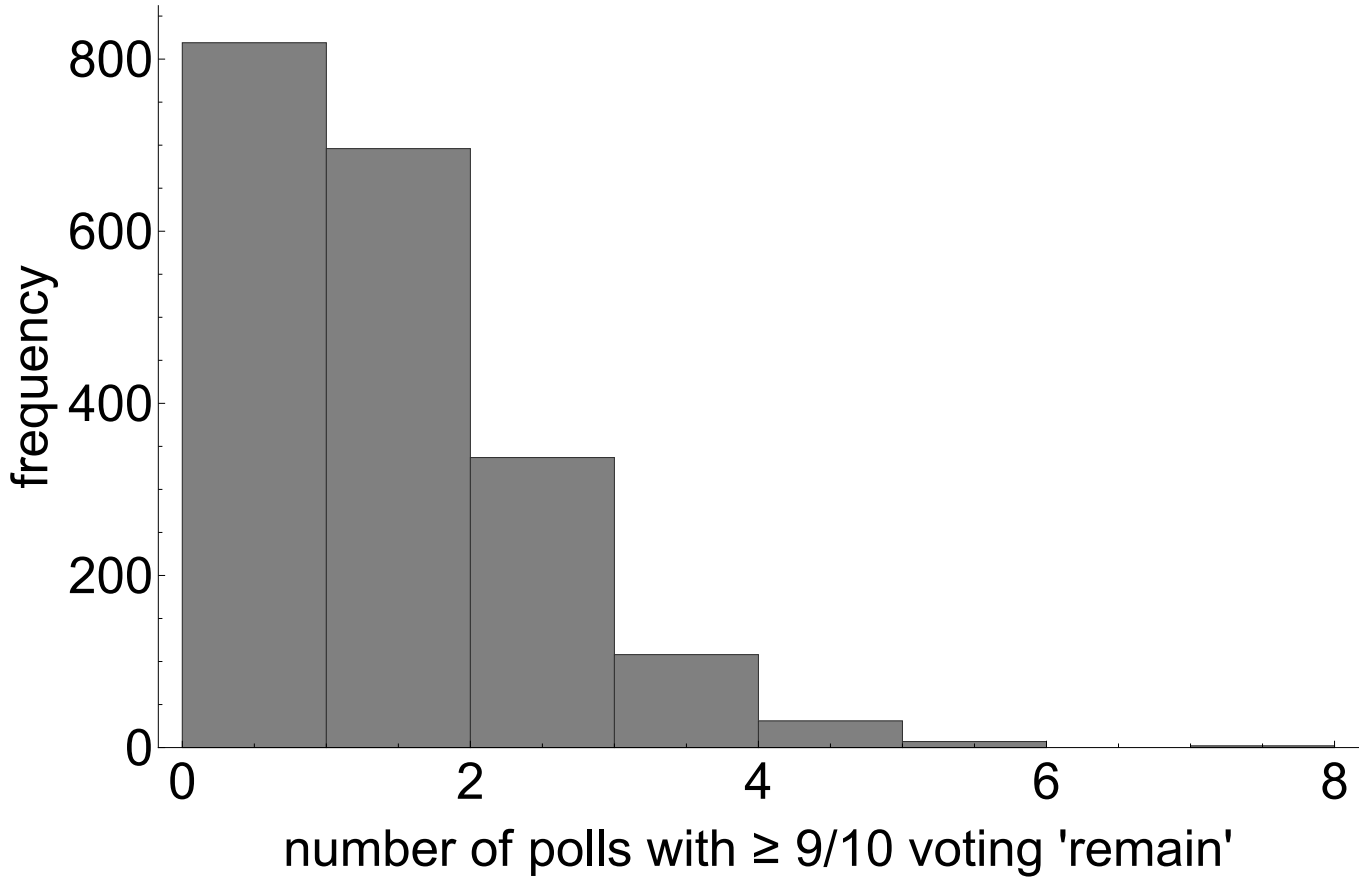
EU referendum: complete pooling model posterior predictive checks

Count number of times that 9 or more people vote 'remain', and find 4/30 cases.



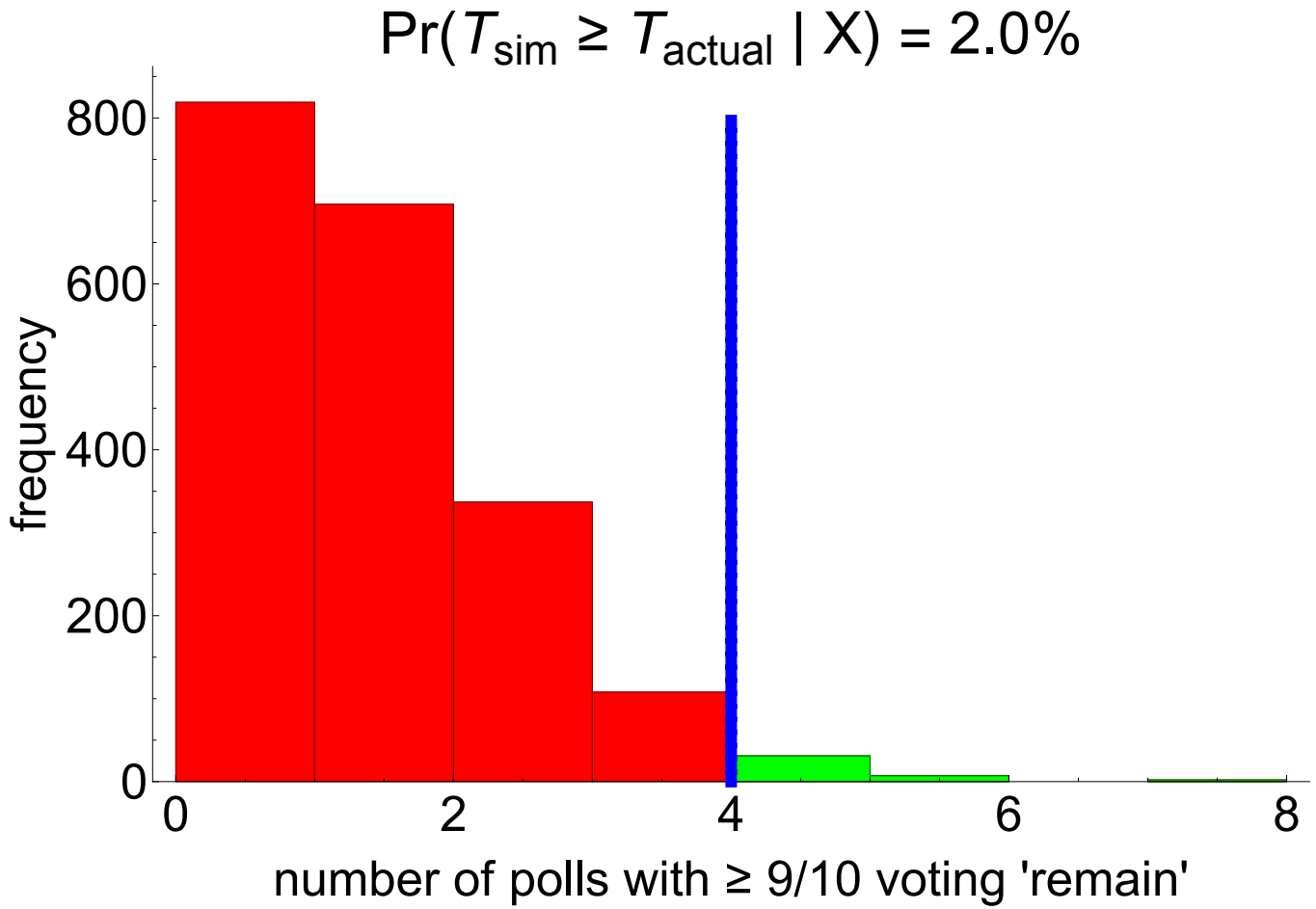
EU referendum: complete pooling model posterior predictive checks

Repeat for 2000 simulated data series; for each simulated dataset counting the number of $X_i \geq 9$.



EU referendum: complete pooling model posterior predictive checks

Low probability of replicating this aspect of real data.



EU referendum: complete pooling model summary

- Assumed a model where the probability of a polled individual intending to vote 'remain' is **identical** across all polls.
- However polls were conducted over a range of time by a range of agencies, each with their own methodology $\implies$ sampling method, exact interview process etc vary.
- Therefore assuming a common θ across polls is too strong an assumption.
- $\implies$ model will **understate** true uncertainty.

$\implies$ try the opposite extreme where we allow a different θ_i for each poll.

EU referendum: heterogeneous model

For each poll i we assume a binomial likelihood:

$$Pr(X = X_i|\theta_i) = \theta_i^{X_i}(1-\theta_i)^{10-X_i} \tag{5}$$

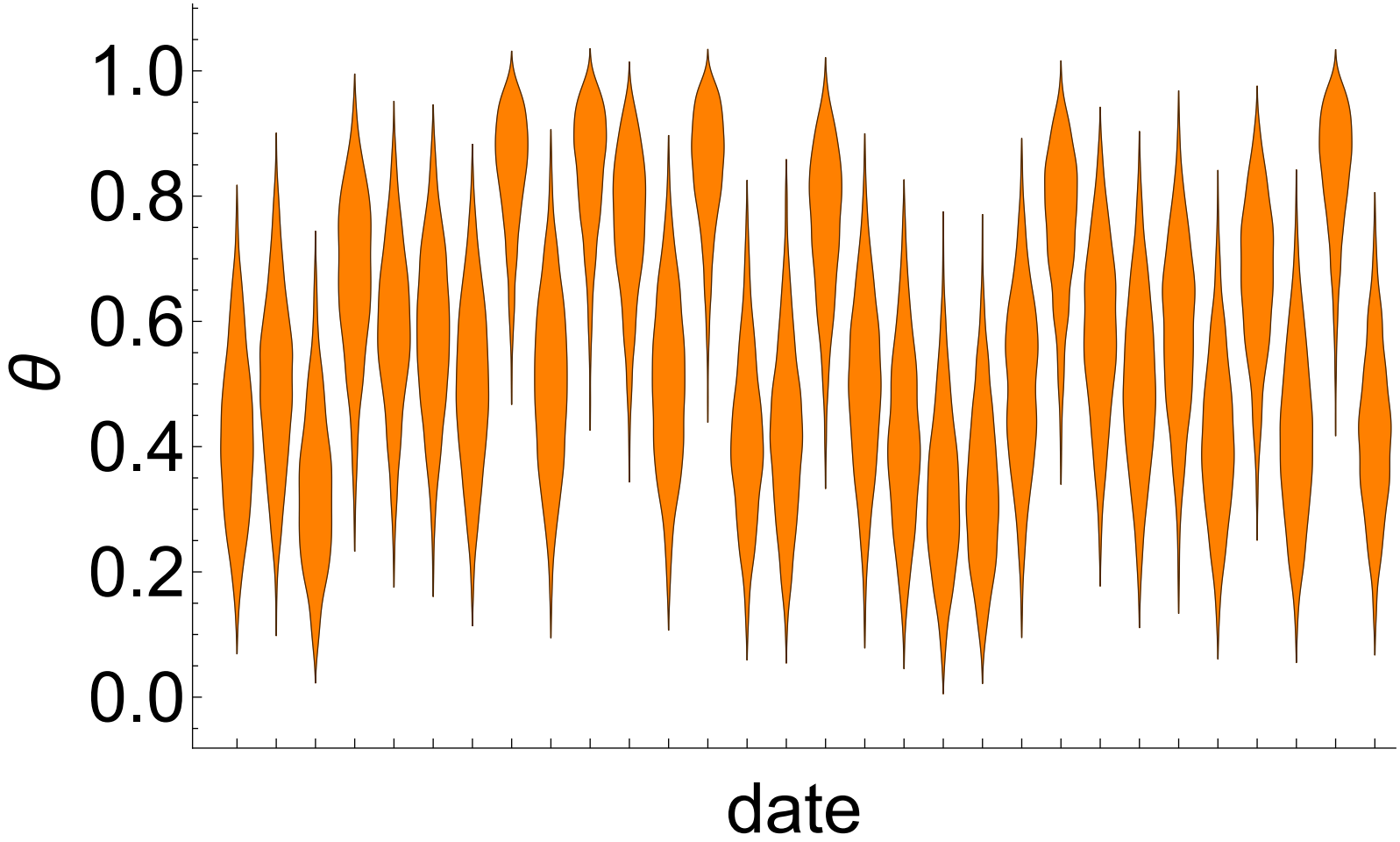
where θ_i is the probability of a interviewee indicating they intend to vote 'remain' in the referendum.
By allowing θ_i to vary across polls $\implies$ **different** data generating processes for each poll!

EU referendum: heterogeneous model

```
data {
    int<lower=1> K; ## number of polls
    int<lower=0> X[K]; ## numbers voting 'remain'
    int<lower=1> N; ## sample size
}
parameters {
    real<lower=0,upper=1> theta[K]; ## now array
}
model {
    for (i in 1:K){
        X[i] ~ binomial(N,theta[i]);## select element
    }
}
```

$\Longrightarrow$ only two changes to homogeneous model necessary.

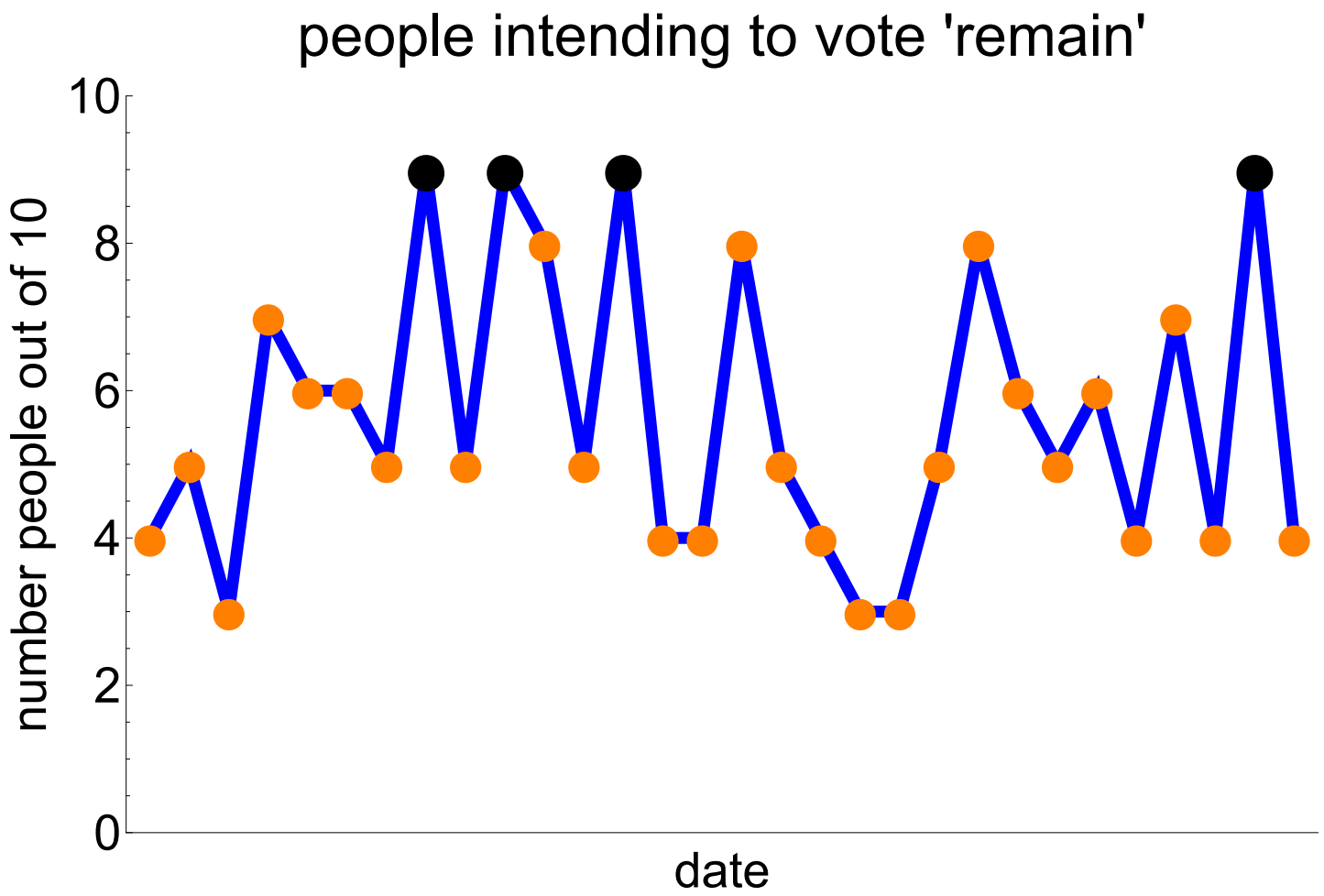
Heterogeneous model results



$\Longrightarrow$ large uncertainty associated with each θ_i.

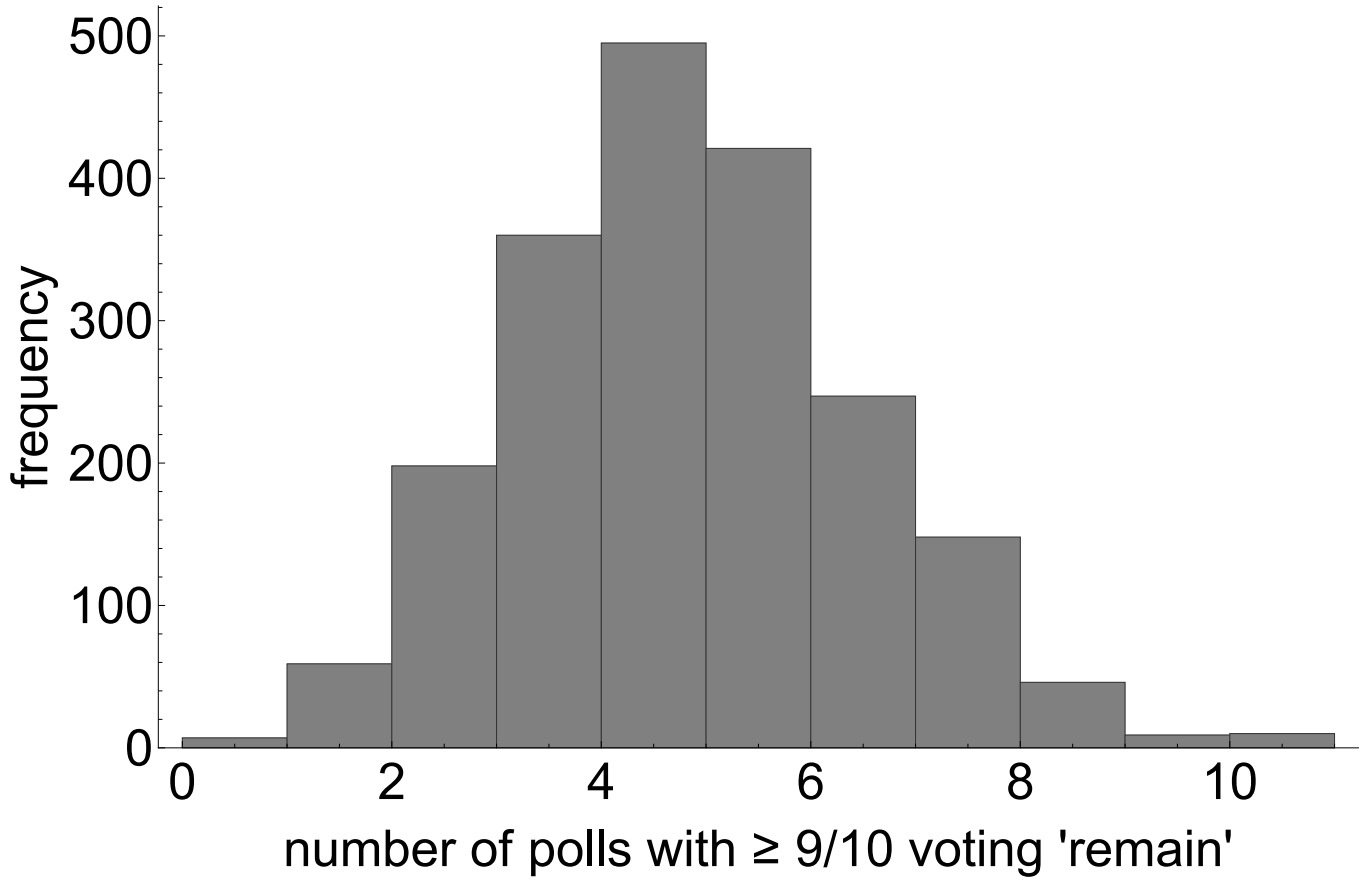
Heterogeneous model posterior predictive checks

Compare again occurrence of 9+/10 'remain' voters with real data; where 4/30.



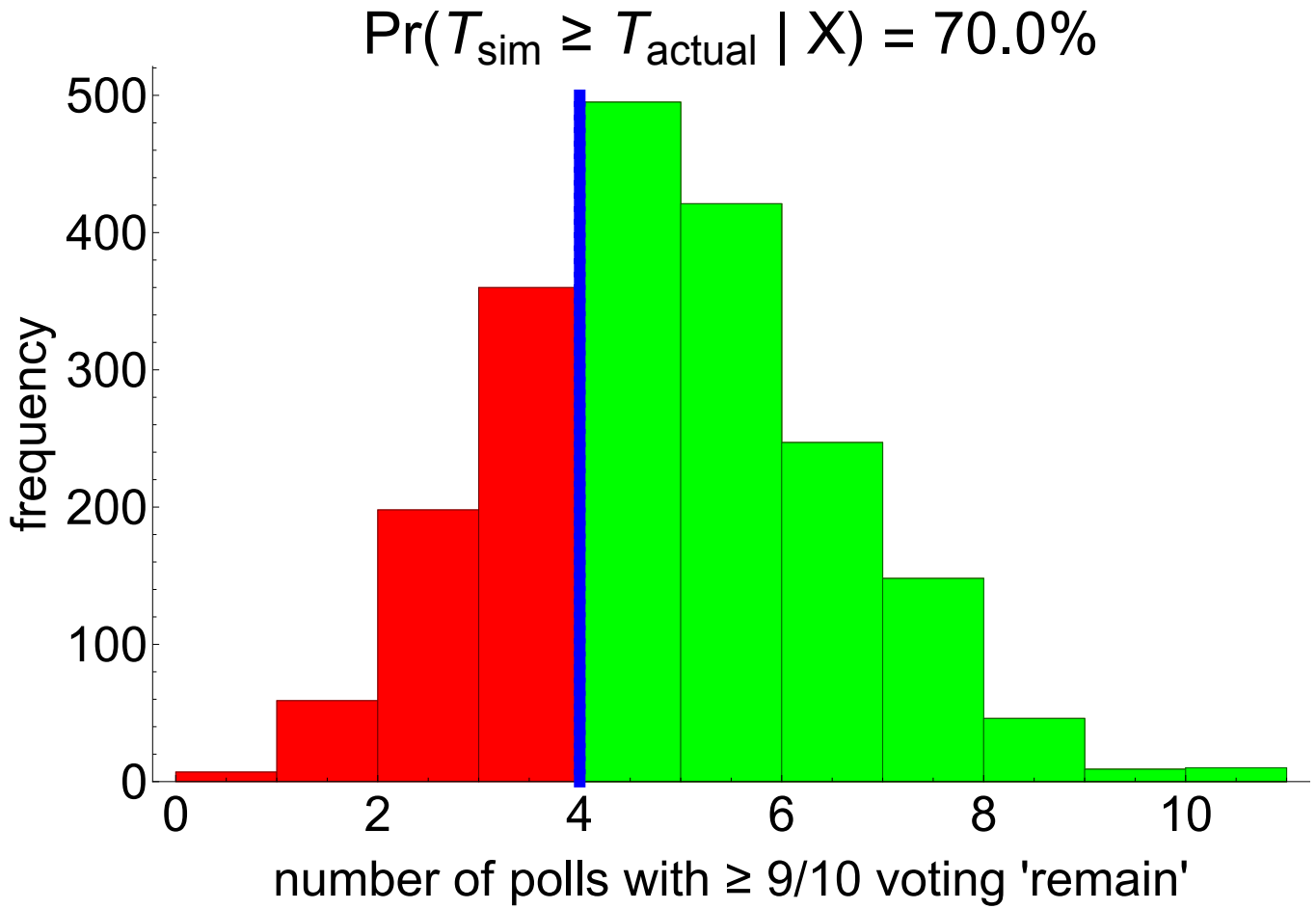
Heterogeneous model posterior predictive checks

For 2000 posterior predictive samples.



Heterogeneous model posterior predictive checks

$\Longrightarrow$ better than complete pooling model.



Heterogeneous model problems

- Heterogeneous model is a better fit to the data than fully-pooled $\Longleftarrow$ is overfitting the data?
- However not clear what we should now forecast for the polls? Should we use:
  - Estimate from one poll.
  - Average across point estimates for all polls.
- Further how should we quantify our uncertainty?

Heterogeneous model: summary

- Same binomial likelihood as before but allowed θ to vary across polls.
- $\implies$ obtained separate estimates of θ across each poll.
- Found considerable variability and uncertainty in estimates of θ_i.
- Heterogeneous θ model better captured the extremes seen in the data $\implies$ fully-pooled model was too strong.
- However key question: **what do we forecast will be the result of the EU referendum, and with what uncertainty?**

Introducing a hierarchical model

- In fully-pooled model case we assumed that data from all the polls was the same; i.e. θ was constant.
- However there are differences between polling methodologies, and time when polls were taken $\implies$ data generated by different processes.
- $\implies$ we estimated a separate θ_i for each poll; i.e. assumed data from different polls was completely unrelated.

Introducing a hierarchical model

Question 1: do we really think that data from different polls is completely unrelated? **Answer 1:** no! After all the polls measure the same thing, at around the same point in time.

Question 2: do we really think that the polls are exactly the same? **Answer 2** no! We rejected this initially because of differences between polling agencies and time over which the polls were done.

Introducing a hierarchical model

Question: isn't there somewhere between the extremes of complete-separation and fully-pooled?

Answer:

Hierarchical model for EU referendum polls

As for heterogeneous model we allow θ to vary by poll:

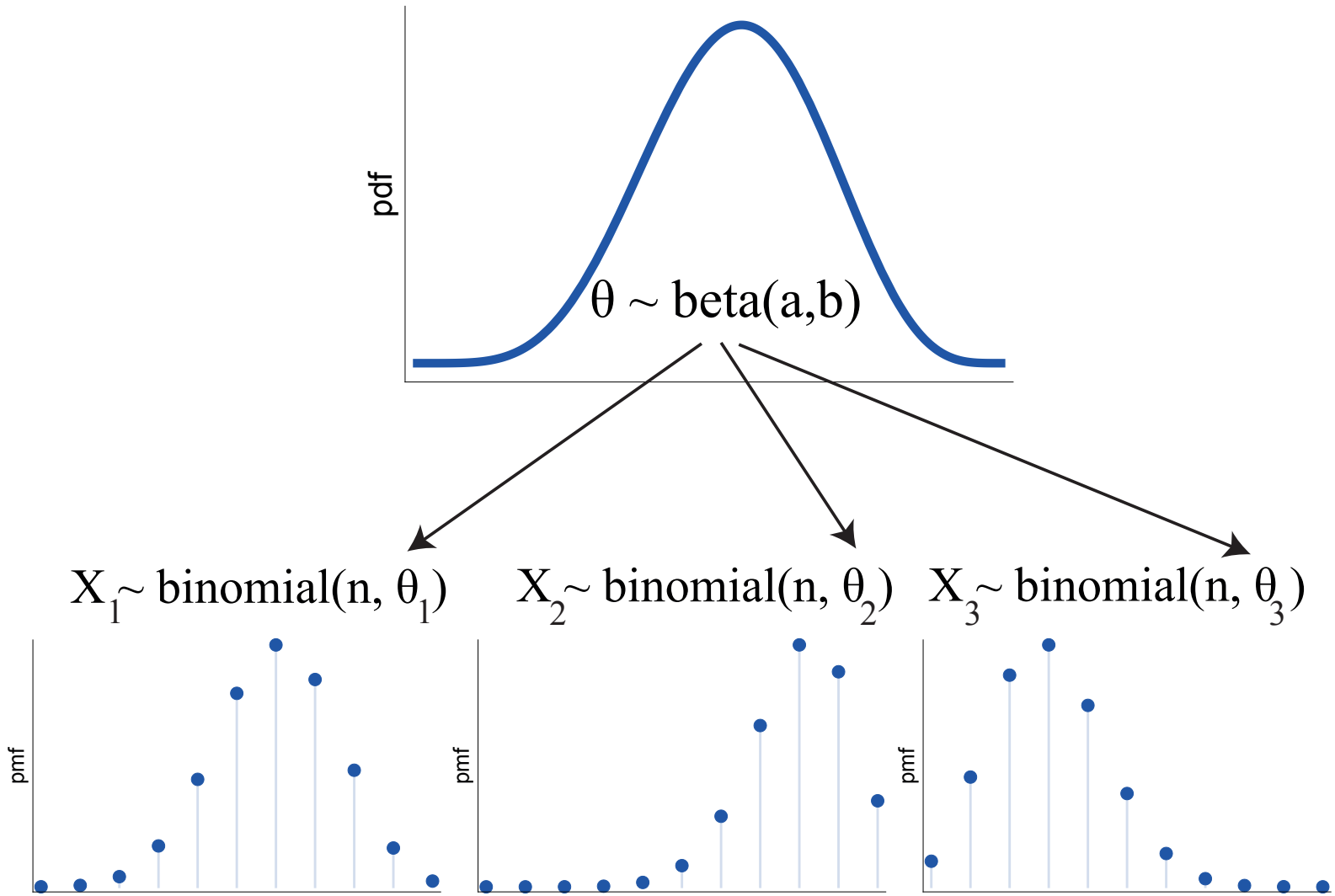
$$Pr(X = X_i|\theta_i) = {\theta_i}^{X_i}(1-\theta_i)^{10-X_i} \tag{6}$$

However now we assume that the θ_i are drawn from a common "population" distribution:

$$\theta_i \sim beta(a, b) \tag{7}$$

where a and b are parameters that define the population-level beta distribution.

Hierarchical model for EU referendum polls



Hierarchical model for EU referendum polls

$$\theta_i \sim \mathit{beta}(a, b) \tag{8}$$

(a, b) are parameters just like any other in Bayesian inference $\Longrightarrow$ assign them **priors**!
Actually easier to set priors for transformed parameters:

$$a = \alpha \times \kappa$$
$$b = (1 - \alpha) \times \kappa$$

where α represents the "population" chance of voting 'remain' and κ measures the concentration $\Longrightarrow$

$$\theta_i \sim \mathit{beta}(\alpha \times \kappa, (1 - \alpha) \times \kappa) \tag{9}$$

Hierarchical model for EU referendum polls

$$\theta_i \sim \textit{beta}(\alpha \times \kappa, (1-\alpha) \times \kappa) \tag{10}$$

Set independent priors:

$$p(\alpha, \kappa) = p(\alpha) \times p(\kappa) \tag{11}$$

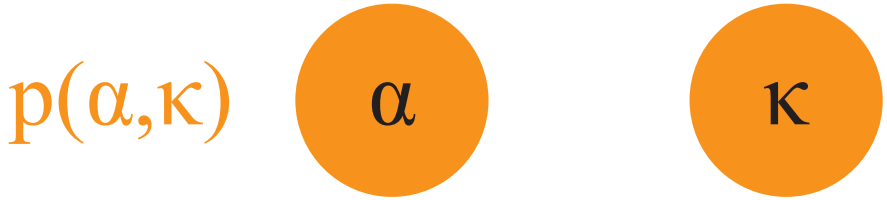
Question: what is the numerator of Bayes' rule for this problem?
Answer: the joint distribution of the data X and parameters; i.e. $p(X, \theta, \alpha, \kappa)$
Another question: how do we find this here? **Another answer:** exploit conditional independence of the problem!

Hierarchical model for EU referendum polls

Start with "population" level parameters.

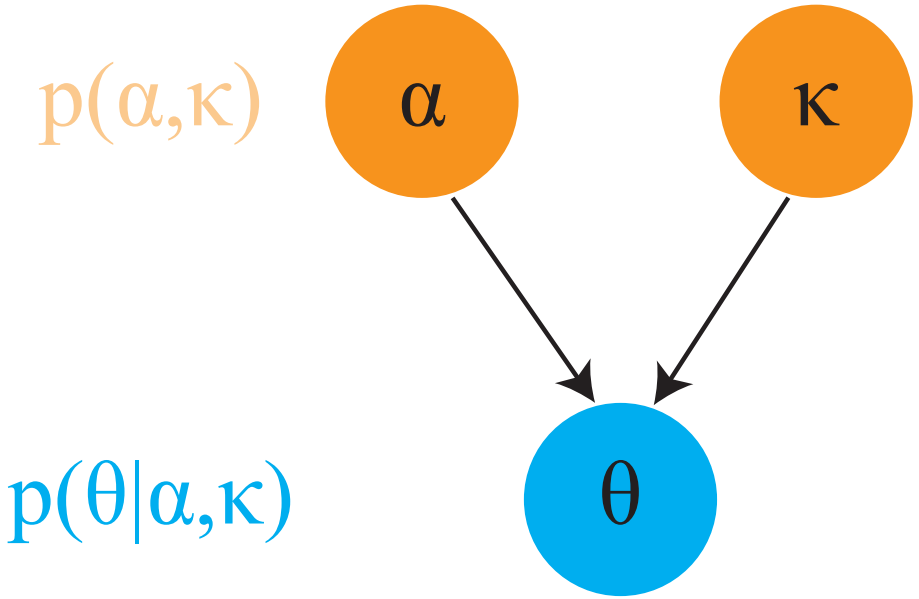
Hierarchical model for EU referendum polls

And determine their joint probability.



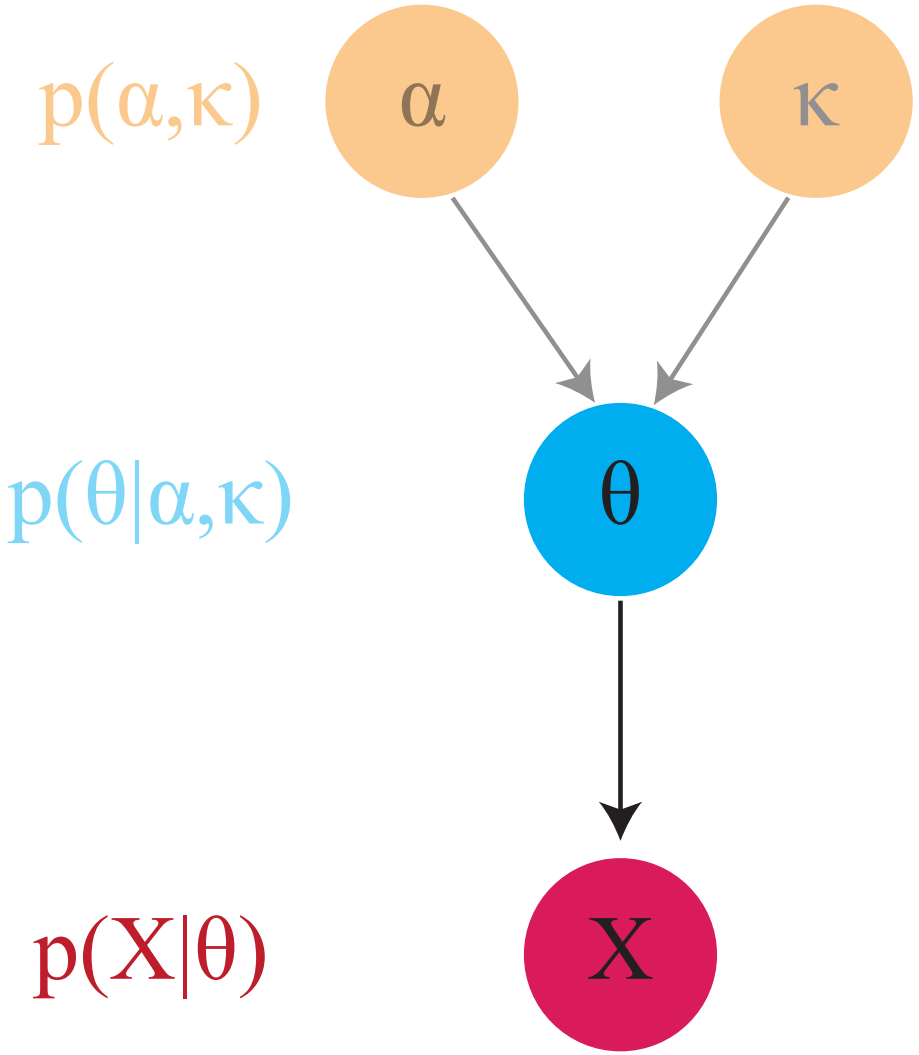
Hierarchical model for EU referendum polls

Find the probability of θ **conditional** on α and κ.



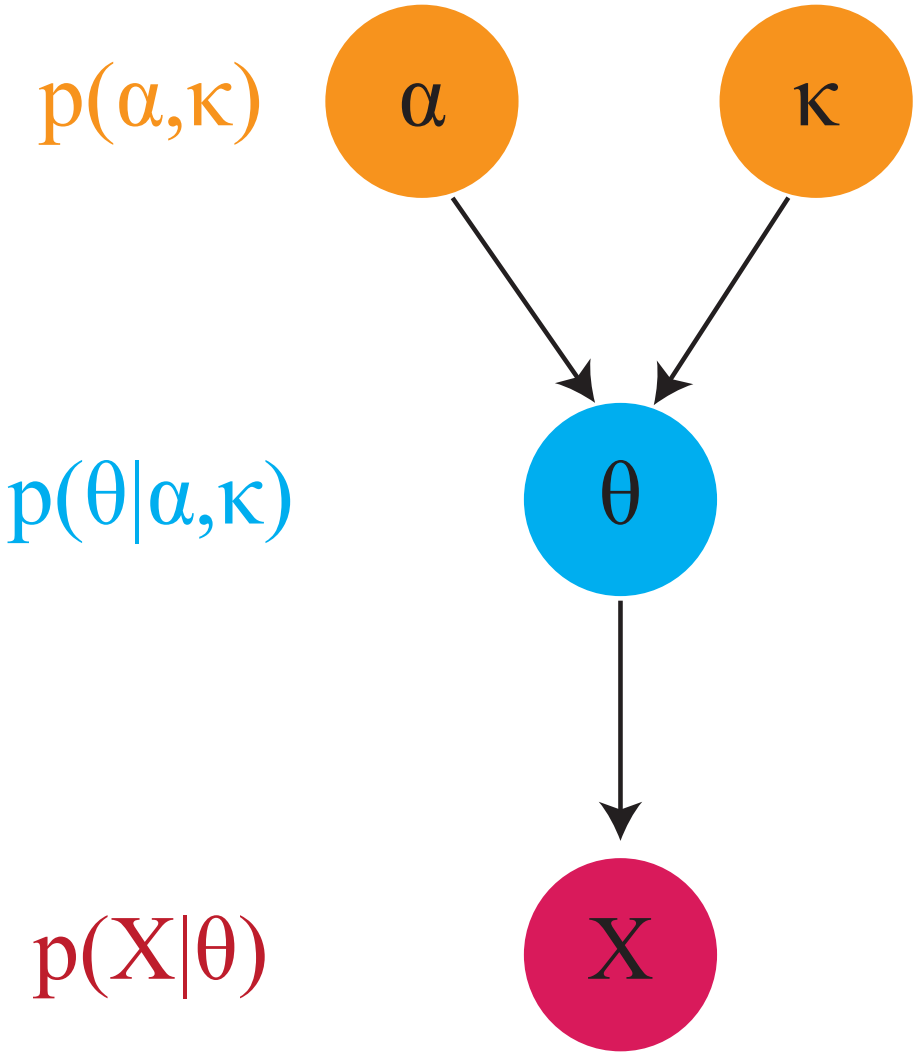
Hierarchical model for EU referendum polls

Find the probability of X **conditional** on θ .



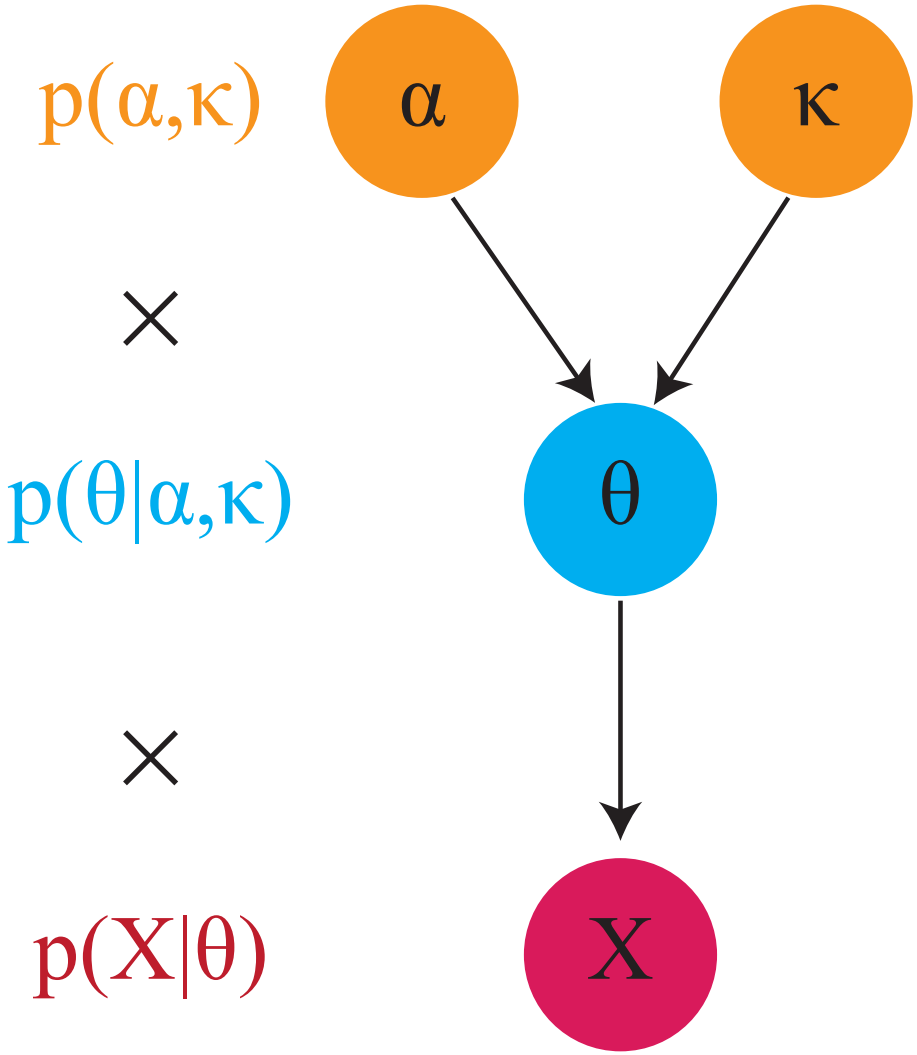
Hierarchical model for EU referendum polls

Finally to obtain the overall probability...



Hierarchical model for EU referendum polls

...multiply together all the terms.



Hierarchical model for EU referendum polls

So the posterior is found as:

$$p(\theta, \alpha, \kappa | X) \propto p(X, \theta, \alpha, \kappa)$$
$$= \underbrace{p(X|\theta)}_{\text{likelihood}} \times \underbrace{p(\theta|\alpha, \kappa)}_{\text{prior}} \times \underbrace{p(\alpha, \kappa)}_{\text{hyper-prior}}$$

- $p(X|\theta)$ is just the **likelihood**.
- $p(\theta|\alpha, \kappa)$ is the **prior** on θ.
- $p(\alpha, \kappa)$ is the **hyper-prior** on the **hyper-parameters** α and κ.
- However the word "hyper" is really just a fancy word we use to represent priors on "population" level parameters.
- In hierarchical models there is a blurring of the likelihood/prior boundary.

Hierarchical model for EU referendum polls: back to the problem

We set the following (hyper-)priors on α and κ:

$$\alpha \sim \textit{beta}(5, 5)$$
$$\kappa \sim \textit{pareto}(1, 0.3)$$

where:

- beta(5,5) has a mean of 0.5, and only has support for $0 \leq \alpha \leq 1$.
- pareto(1,0.3) is a distribution only with support for values of $\kappa \geq 1$.

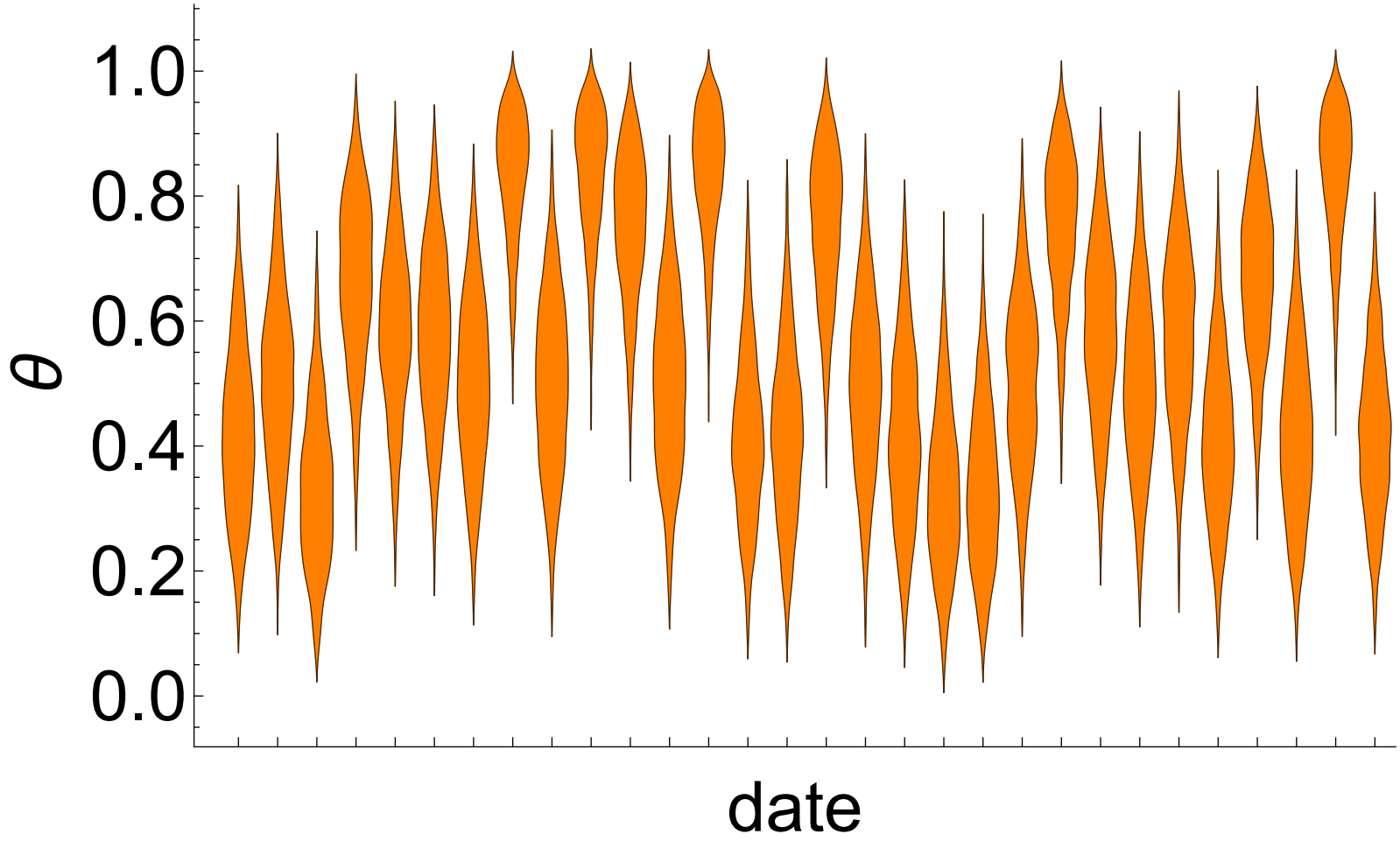
Hierarchical model for EU referendum polls: coding up model in Stan

```
parameters {
    real<lower=0, upper=1> alpha;
    real<lower=1> kappa;
    vector<lower=0, upper=1>[K] theta;
}
model {
    for (i in 1:K){
        Y[i] ~ binomial(N[i],theta[i]);## Likelihood
}
    ## prior
    theta ~ beta(alpha * kappa, (1 - alpha) * kappa);

    ## hyper-priors
    kappa ~ pareto(1, 0.3);
    alpha ~ beta(5,5);
}
```

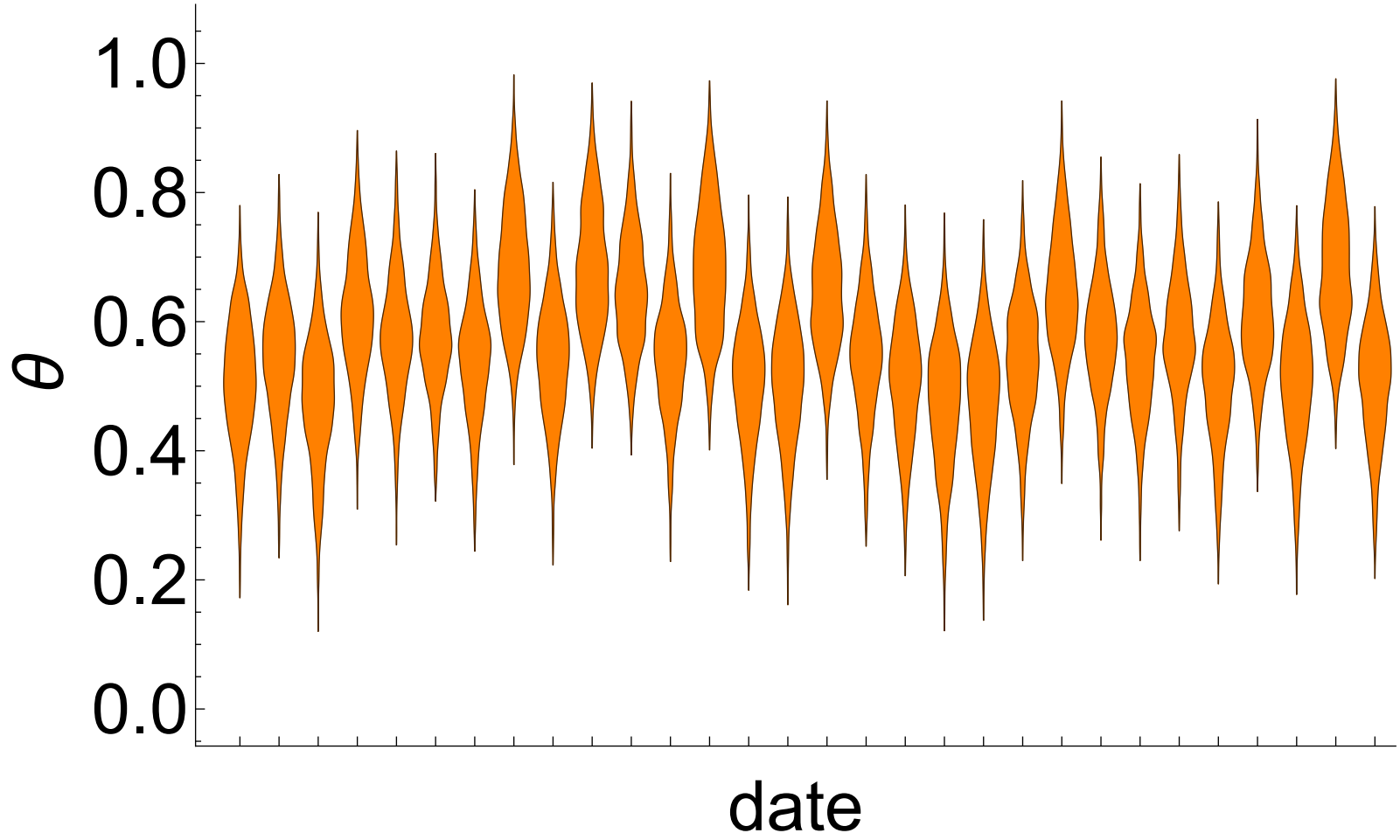
Hierarchical model for EU referendum polls: heterogeneous model estimates

Remember the posterior from the heterogeneous model?



Hierarchical model for EU referendum polls: hierarchical model estimates

Hierarchical model estimates.



Hierarchical model for EU referendum polls: hierarchical model estimates

Two effects evident:

- Shrinkage to **grand** mean.
- Shrinkage in variance.

Shrinkage to grand mean because hierarchical models lie on a spectrum between completely heterogeneous estimates and fully pooled.
Important: the data determines where on the spectrum we exactly end up! No choice in analysis.
Important: shrinkage helps reduce the variance of estimates $\implies$ outliers have less impact.

Also "partially-pooling" information across groups $\implies$ essentially a larger sample size and lower variance.

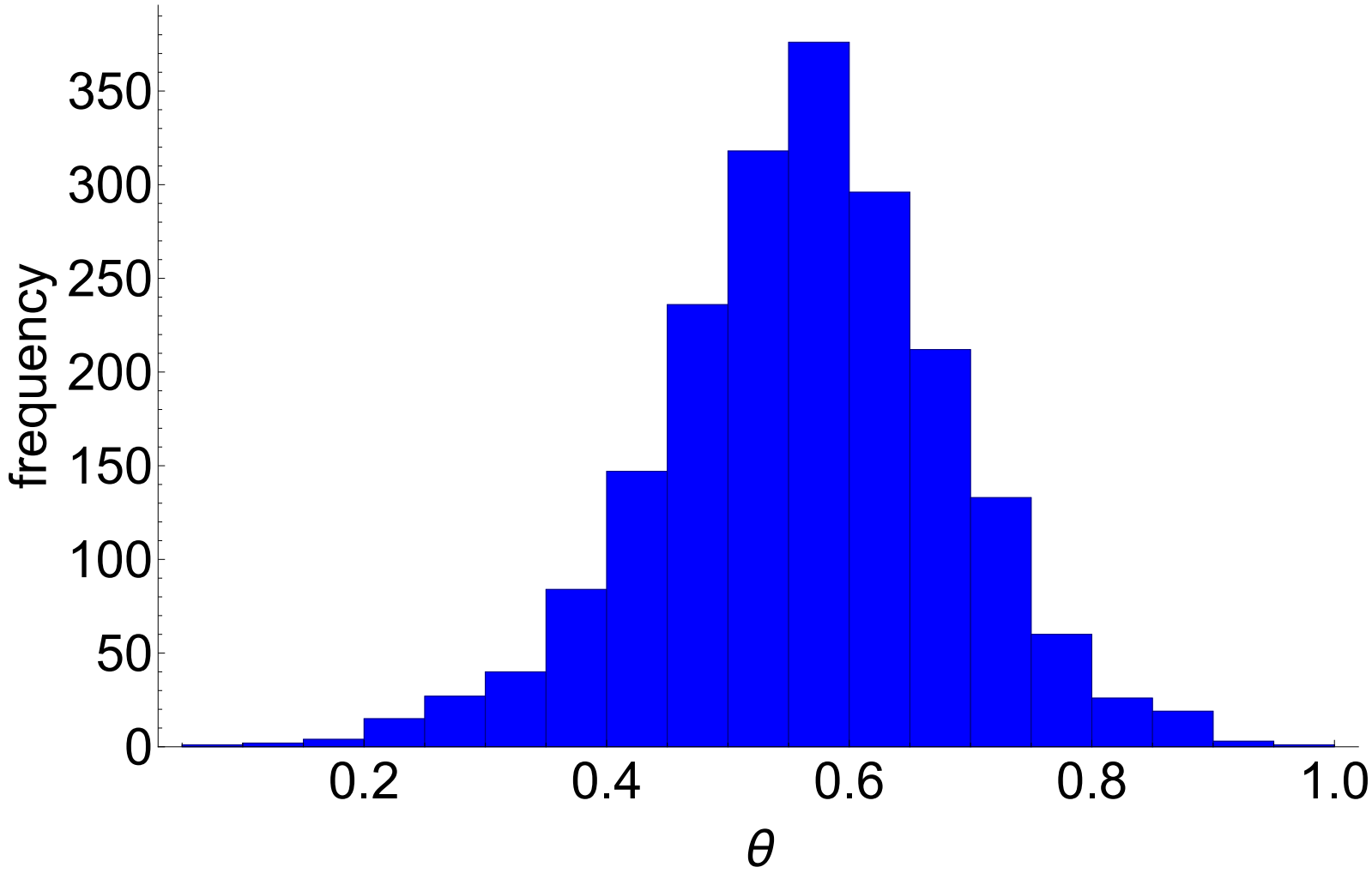
Hierarchical model: forecasting outcome of overall elections

Want to estimate the value of θ for the last poll: the referendum itself!
To do this do the following:

1. Sample (α, κ) from their posteriors.
2. Sample $\theta \sim \mathit{beta}(\alpha\kappa, (1-\alpha)\kappa)$.

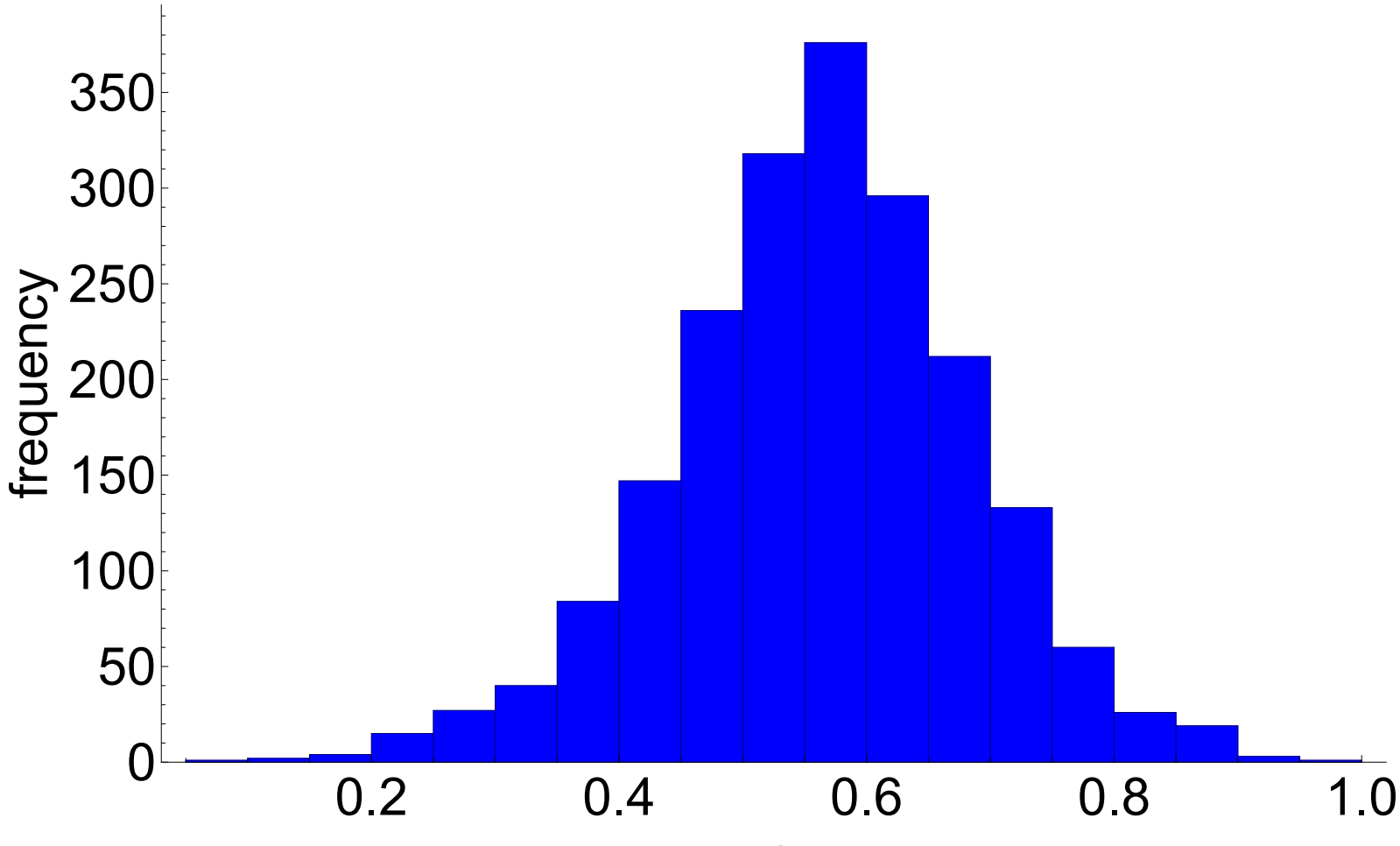
Hierarchical model: forecasting outcome of overall elections

Yields the posterior below (gulp):



Hierarchical model: forecasting outcome of overall elections

Conclusion: the result could go either way (I did a similar analysis for real poll data and it produced a posterior very much like the one below!)



Hierarchical models: summary

- Model with **same** parameters for all groups $\Longrightarrow$ data generating process is the **same**.
- Model with separately estimated group-level parameters $\Longrightarrow$ data generating process is completely **different**.
- Frequently neither of the aforementioned models are appropriate; i.e. we want some dependence between parameters but not 100%.
- $\Longrightarrow$ use hierarchical model where the data determines parameter dependence across groups.
- In hierarchical models $\Longrightarrow$ **shrinkage** of **group** means towards the **grand** mean.
- Also **shrinkage** of group variance $\Longleftarrow$ sample size$\uparrow$ by **partial-pooling** information across groups.

Lecture summary

- Stan carries out inference by a variant of HMC called NUTS $\implies$ very fast!
- Stan is (hopefully) a fairly intuitive language to learn.
- Makes MCMC sampling easier for a wide class of problems in Bayesian inference.
- Non-hierarchical models have failings: understate uncertainty, or overfit.
- Hierarchical models $\implies$ "happy" medium between fully-pooled and completely heterogeneous models.
- Data determines the amount of dependence across groups.

Reading list

Olympians breakfast:

- Chapters 5 (hierarchical models) and 15 (hierarchical linear models) in "Bayesian data analysis", by Gelman et al. (2014), 3rd edition.
- "Case studies" section of Stan website: `http://mc-stan.org/documentation/case-studies.html` - walk through inference for many hierarchical/non-hierarchical models.
- Chapter 12 (multilevel models) in "Statistical Rethinking", by McElreath (2016).
- Chapter 1 (+ rest) in "Data analysis using regression and multilevel/hierarchical models" by Gelman and Hill (2007).

Not sure I underSTANd

Overfit.

Not sure I underSTANd

Underfit.

Not sure I underSTANd

Fit.