Stan

Probabilistic Programming Language

Core Development Team:

Andrew Gelman, **Bob Carpenter**, Matt Hoffman
Daniel Lee, Ben Goodrich, Michael Betancourt,
Marcus Brubaker, Jiqiang Guo, Peter Li,
Allen Riddell, Marco Inacio

Paris-Dauphine 2014

mc-stan.org

What is Stan?

- Stan is an **imperative** probabilistic programming language
  - cf., BUGS: declarative; Church: functional; Figaro: object-oriented
- Stan **program**
  - declares variables
  - codes log posterior (penalized likelihood)
- Stan **inference**
  - MCMC for full Bayesian inference
  - MLE for point estimation
- Stan is **open source** (BSD core C++, GPLv3 interfaces)

  hosted on GitHub; uses Eigen matrix lib, Boost C++ lib, googletest

Who's Using Stan?

- 750+ registrations for mailing list across the physical, biomedical, and social sciences, plus engineering, finance and marketing

- Subjects of Papers on Google Scholar in the sciences

  clinical drug trials, general computational statistics, entomology, opthalmology, neurology, sociology and population dynamics, genomics, agriculture, psycholinguistics, molecular biology, population dynamics, materials engineering, botany, astrophysics, oceanography, election prediction, fisheries, cancer biology, public health and epidemiology, population ecology, collaborative filtering for recommender systems, climatology, educational testing, natural language processing

Books and Model Sets

- 400+ page core language tutorial manual
- separate installation, getting started, interface manuals
  (RStan, PyStan, CmdStan)
- BUGS and JAGS examples (all 3 volumes),
- Gelman and Hill, *Data Analysis Using Regression and Multilevel/Hierarchical Models*
- Wagenmakers and Lee, *Bayesian Cognitive Modeling*
- two other books in progress

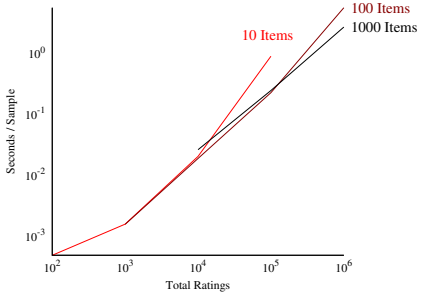
Scaling and Evaluation

- Types of scaling
  - more data
  - more parameters
  - **more complex models**
- for MCMC, measure (as function of scaling)
  - time to convergence
  - time per effective sample at mixing
  - memory usage

Stan vs. BUGS / JAGS

- JAGS is the former **state-of-the-art**, **black-box** MCMC method
  (Gibbs with slice sampling; chokes w. high posterior correlation)

- Stan (2.0) vs. BUGS on scaling IRT model
  - Stan about **10 times faster**
  - Stan memory and time scales linearly
  - linearly scales with number of question answers



				Stan		JAGS	
# items	# raters	# groups	# data	time	memory	time	memory
20	2,000	100	40,000	:02m	16MB	:03m	220MB
40	8,000	200	320,000	:16m	92MB	:40m	1400MB
80	32,000	400	2,560,000	4h:10m	580MB	:??m	?MB

- Stan can handle problems that choke BUGS and JAGS

Part I
Stan Front End

Platforms and Interfaces

- **Platforms**
  Linux, Mac OS X, Windows
- **C++ API**
  portable, standards compliant (C++03 now, moving to C++11)
- **Interfaces**
  - **CmdStan**: Command-line or shell interface (direct executable)
  - **RStan**: R interface (Rcpp in memory)
  - **PyStan**: Python interface (Cython in memory)
  - **MStan***: MATLAB interface (lightweight external process)
  - **JuliaStan***: Julia interface (lightweight external process)

* User contributed

Example: Bernoulli

```
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  y ~ bernoulli(theta);
}
```

notes: `theta` *uniform on* $[0, 1]$, `y` *vectorized*

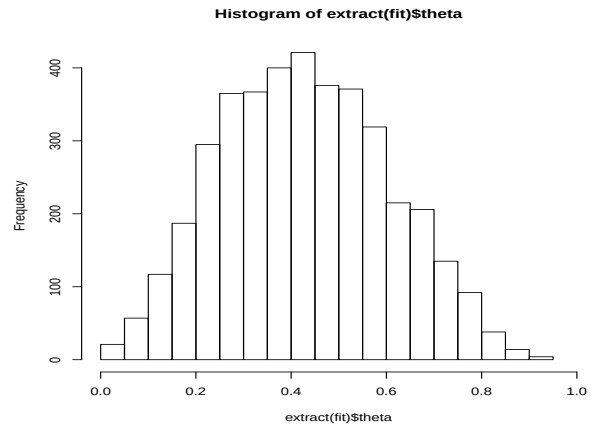
RStan Execution

```
> N <- 5;   y <- c(0,1,1,0,0);
> fit <- stan("bernoulli.stan", data = c("N", "y"));
> print(fit, digits=2)
```

Inference for Stan model: bernoulli.
4 chains, each with iter=2000; warmup=1000; thin=1;

	mean	se_mean	sd	2.5%	50%	97.5%	n_eff	Rhat
theta	0.43	0.01	0.18	0.11	0.42	0.78	1229	1
lp__	-5.33	0.02	0.80	-7.46	-5.04	-4.78	1201	1

```
> hist( extract(fit)$theta )
```



Basic Program Blocks

- **`data`** (once)
  - *content*: declare data types, sizes, and constraints
  - *execute*: read from data source, validate constraints
- **`parameters`** (every log prob eval)
  - *content*: declare parameter types, sizes, and constraints
  - *execute*: transform to constrained, Jacobian
- **`model`** (every log prob eval)
  - *content*: statements definining posterior density
  - *execute*: execute statements

Derived Variable Blocks

- **`transformed data`** (once after data)
  - *content*: declare and define transformed data variables
  - *execute*: execute definition statements, validate constraints
- **`transformed parameters`** (every log prob eval)
  - *content*: declare and define transformed parameter vars
  - *execute*: execute definition statements, validate constraints
- **`generated quantities`** (once per draw, `double` type)
  - *content*: declare and define generated quantity variables; includes pseudo-random number generators
    (for posterior predictions, event probabilities, decision making)
  - *execute*: execute definition statements, validate constraints

User-Defined Functions (Stan 2.3)

- **`functions`** (compiled with model)
  - *content*: declare and define general (recursive) functions (use them elsewhere in program)
  - *execute*: compile with model

- Example

```
functions {

  real relative_difference(real u, real v) {
    return 2 * fabs(u - v) / (fabs(u) + fabs(v));
  }

}
```

Variable and Expression Types

Variables and expressions are **strongly, statically typed**.

- **Primitive**: `int, real`
- **Matrix**: `matrix[M,N], vector[M], row_vector[N]`
- **Bounded**: primitive or matrix, with `<lower=L>, <upper=U>, <lower=L,upper=U>`
- **Constrained Vectors**: `simplex[K], ordered[N], positive_ordered[N], unit_length[N]`
- **Constrained Matrices**: `cov_matrix[K], corr_matrix[K], cholesky_factor_cov[M,N], cholesky_factor_corr[K]`
- **Arrays:** of any type (and dimensionality)

Logical Operators

Op.	*Prec.*	*Assoc.*	*Placement*	*Description*
||	9	left	binary infix	logical or
&&	8	left	binary infix	logical and
==	7	left	binary infix	equality
!=	7	left	binary infix	inequality
<	6	left	binary infix	less than
<=	6	left	binary infix	less than or equal
>	6	left	binary infix	greater than
>=	6	left	binary infix	greater than or equal

Arithmetic and Matrix Operators

Op.	*Prec.*	*Assoc.*	*Placement*	*Description*
+	5	left	binary infix	addition
-	5	left	binary infix	subtraction
*	4	left	binary infix	multiplication
/	4	left	binary infix	(right) division
\	3	left	binary infix	left division
.*	2	left	binary infix	elementwise multiplication
./	2	left	binary infix	elementwise division
!	1	n/a	unary prefix	logical negation
-	1	n/a	unary prefix	negation
+	1	n/a	unary prefix	promotion (no-op in Stan)
^	2	right	binary infix	exponentiation
'	0	n/a	unary postfix	transposition
()	0	n/a	prefix, wrap	function application
[]	0	left	prefix, wrap	array, matrix indexing

Built-in Math Functions

- All built-in **C++ functions and operators**

  C math, TR1, C++11, including all trig, pow, and special log1m, erf, erfc, fma, atan2, etc.

- Extensive library of **statistical functions**

  e.g., softmax, log gamma and digamma functions, beta functions, Bessel functions of and second kind, etc.

- Efficient, arithmetically stable **compound functions**

  e.g., multiply log, log sum of exponentials, log inverse logit

Built-in Matrix Functions

- **Basic arithmetic**: all arithmetic operators
- **Elementwise arithmetic**: vectorized operations
- **Solvers**: matrix division, (log) determinant, inverse
- **Decompositions**: QR, Eigenvalues and Eigenvectors, Cholesky factorization, singular value decomposition
- **Compound Operations**: quadratic forms, variance scaling
- **Ordering, Slicing, Broadcasting**: sort, rank, block, rep
- **Reductions**: sum, product, norms
- **Specializations**: triangular, positive-definite, etc.

Distribution Library

- Each distribution has
  - log density or mass function
  - cumulative distribution functions, plus complementary versions, plus log scale
  - pseudo Random number generators
- Alternative parameterizations

  (e.g., Cholesky-based multi-normal, log-scale Poisson, logit-scale Bernoulli)
- New multivariate correlation matrix density: LKJ

  degrees of freedom controls shrinkage to (expansion from) unit matrix

Statements

- **Sampling**: `y ~ normal(mu,sigma)` (increments log probability)
- **Log probability**: increment_log_prob(lp);
- **Assignment**: `y_hat <- x * beta;`
- **For loop**: `for (n in 1:N) ...`
- **While loop**: `while (cond) ...`
- **Conditional**: `if (cond) ...; else if (cond) ...; else ...;`
- **Block**: `{ ... }` (allows local variables)
- **Print**: print("theta=",theta);

Full Bayes with MCMC

- Adaptive **Hamiltonian Monte Carlo** (HMC)
- Adaptation **during warmup**
  - step size adapted to target Metropolis acceptance rate
  - mass matrix estimated with regularization
    sample covariance of second half of warmup iterations
    (assumes constant posterior curvature)
- Adaptation **during sampling**
  - number of steps
    aka no-U-turn sampler (NUTS)
- **Initialization** user-specified or random unconstrained

Posterior Inference

- Generated quantities block for **inference**

  (predictions, decisions, and event probabilities)

- **Extractors** for samples in RStan and PyStan

- Coda-like **posterior summary**
  - posterior mean w. standard error, standard deviation, quantiles
  - split-$\hat{R}$ multi-chain convergence diagnostic (Gelman and ...)
  - multi-chain effective sample size estimation (FFT algorithm)

- Model comparison with **WAIC**

  (internal log likelihoods; external cross-sample statistics)

Penalized MLE

- Posterior **mode finding** via BFGS optimization
  (uses model gradient, efficiently approximates Hessian)
- **Disables Jacobians** for parameter inverse transforms
- Models, data, initialization as in MCMC
- **Very Near Future**
  - **Standard errors** on unconstrained scale
    (estimated using curvature of penalized log likelihood function
  - Standard errors **on constrained scale**)
    (sample unconstrained approximation and inverse transform)
  - L-BFGS optimizer

Stan as a Research Tool

- Stan can be used to **explore algorithms**
- Models transformed to **unconstrained support** on $\mathbb{R}^n$
- Once a model is compiled, have
  - **log probability, gradient, and Hessian**
  - data I/O and parameter initialization
  - model provides variable names and dimensionalities
  - transforms to and from constrained representation (with or without Jacobian)
- Very Near Future:
  - second- and **higher-order derivatives** via auto-diff

Part II
Under the Hood

Euclidean Hamiltonian

- **Phase space**: q position (parameters); p momentum
- **Posterior density**: $\pi(q)$
- **Mass matrix**: M
- **Potential energy**: $V(q) = -\log \pi(q)$
- **Kinetic energy**: $T(p) = \frac{1}{2} p^\top M^{-1} p$
- **Hamiltonian**: $H(p, q) = V(q) + T(p)$
- **Diff eqs**:

$$\frac{dq}{dt} = +\frac{\partial H}{\partial p} \qquad \frac{dp}{dt} = -\frac{\partial H}{\partial q}$$

Leapfrog Integrator Steps

- Solves Hamilton's equations by **simulating dynamics**
  (symplectic [volume preserving]; ϵ^3 error per step, ϵ^2 total error)
- Given: **step size** ϵ, **mass matrix** M, **parameters** q
- **Initialize kinetic** energy, $p \sim \text{Normal}(0, \mathbf{I})$
- **Repeat** for L leapfrog steps:

$$p \leftarrow p - \frac{\epsilon}{2} \frac{\partial V(q)}{\partial q} \qquad \text{[half step in momentum]}$$

$$q \leftarrow q + \epsilon M^{-1} p \qquad \text{[full step in position]}$$

$$p \leftarrow p - \frac{\epsilon}{2} \frac{\partial V(q)}{\partial q} \qquad \text{[half step in momentum]}$$

Standard HMC

- **Initialize parameters** diffusely

  Stan's default: $q \sim \text{Uniform}(-2, 2)$ on unconstrained scale

- For each draw
  - leapfrog integrator **generates proposal**
  - **Metropolis accept** step ensures **detailed balance**

- **Balancing act**: small ϵ has low error, requires many steps

- Results **highly sensitive** to step size ϵ and mass matrix M

Tuning HMC During Warmup

- **Chicken-and-egg** problem
  - convergence to high mass volume requires adaptation
  - adaptation requires convergence
- During warmup, tune
  - **step size**: line search to achieve target acceptance rate
  - **mass matrix**: estimate with second half of warmup
- Use exponentially growing adaptation block sizes

Position-Independent Curvature

- **Euclidean** HMC uses **global mass matrix** M
- Works for densities with **position-independent curvature**
- **Counterexample**: hierarchical model
  - hierarchical variance parameter controls lower-level scale
  - mitigate by reducing target acceptance rate
- **Riemannian-manifold** HMC (coming soon)
  - automatically adapts to varying curvature
  - no need to estimate mass matrix
  - need to regularize Hessian-based curvature estimate
    (Betancourt *arXiv*; SoftAbs metric)

Adapting HMC During Sampling

- No-U-turn sampler (**NUTS**)
- Subtle algorithm to maintain **detailed balance**
- Move randomly **forward or backward in time**
- Double number of leapfrog steps each move (**binary tree**)
- Stop when a subtree makes a **U-turn**

  (rare: throw away second half if not end to end U-turn)
- **Slice sample** points along last branch of tree
- **Generalized** to Riemannian-manifold HMC

  (Betancourt, arXiv)

Reverse-Mode Auto Diff

- Eval gradient in small multiple of function eval time
  (independent of dimensionality)
- Templated **C++ overload** for all functions
- Code **partial derivatives** for basic operations
- Function evaluation builds up **expression tree**
- Dynamic program propagates **chain rule** in reverse pass
- Extensible w. **object-oriented** custom partial propagation
- Arena-based **memory management**
  (customize `operator new`)

Forward-Mode Auto Diff

- Templated **C++ overload** for all functions
- Code **partial derivatives** for basic operations
- Function evaluation propagates **chain rule** forward
- Nest reverse-mode in forward for **higher-order**
- **Jacobians**
  - Rerun propagation pass in reverse mode
  - Rerun forward construction with forward mode
- Faster autodiff rewrite coming in six months to one year

Autodiff Functionals

- Fully encapsulates autodiff in C++
- Autodiff operations are functionals (higher-order functions)
  - gradients, Jacobians, gradient-vector product
  - directional derivative
  - Hessian-vector product
  - Hessian
  - gradient of trace of matrix-Hessian product
    (for SoftAbs RHMC)
- Functions to differentiate coded as functors (or pointers)
  (enables dynamic C++ bind or lambda)

Variable Transforms

- Code HMC and optimization with $\mathbb{R}^n$ **support**
- Transform constrained parameters to unconstrained
  - lower (upper) bound: offset (negated) log transform
  - lower and upper bound: scaled, offset logit transform
  - simplex: centered, stick-breaking logit transform
  - ordered: free first element, log transform offsets
  - unit length: spherical coordinates
  - covariance matrix: Cholesky factor positive diagonal
  - correlation matrix: rows unit length via quadratic stick-breaking

Variable Transforms (cont.)

- Inverse transform from unconstrained $\mathbb{R}^n$
- Evaluate log probability in model block on natural scale
- Optionally adjust log probability for change of variables
  (add log determinant of inverse transform Jacobian)

Parsing and Compilation

- Stan code **parsed** to abstract syntax tree (AST)
  (Boost Spirit Qi, recursive descent, lazy semantic actions)
- C++ model class **code generation** from AST
  (Boost Variant)
- C++ code **compilation**
- **Dynamic linking** for RStan, PyStan

Coding Probability Functions

- **Vectorized** to allow scalar or container arguments
  (containers all same shape; scalars broadcast as necessary)
- Avoid **repeated computations**, e.g. $\log \sigma$ in

$$\begin{aligned} \log \text{Normal}(y|\mu,\sigma) &= \sum_{n=1}^{N} \log \text{Normal}(y_n|\mu,\sigma) \\ &= \sum_{n=1}^{N} -\log\sqrt{2\pi} - \log\sigma - \frac{y_n-\mu}{2\sigma^2} \end{aligned}$$

- recursive **expression templates** to broadcast and cache scalars, generalize containers (arrays, matrices, vectors)
- **traits** metaprogram to **drop constants** (e.g., $-\log\sqrt{2\pi}$) and calculate intermediate and return types

Models with Discrete Parameters

- e.g., simple mixture models, survival models, HMMs, discrete measurement error models, missing data
- **Marginalize out** discrete parameters
- Efficient sampling due to **Rao-Blackwellization**
- Inference straightforward with expectations

- Too **difficult** for many of our users
  (exploring encapsulation options)

Models with Missing Data

- In principle, missing data just **additional parameters**
- In practice, how to declare?
  - **observed** data as data variables
  - **missing** data as parameters
  - combine into single vector

    (in transformed parameters or local in model)

Part III
What's Next?

Differential Equation Solver

- Auto-diff solutions w.r.t. parameters
- Integrate coupled system for solution with partials
- Auto-diff coupled Jacobian for stiff systems

- C++ prototype integrated for large PK/PD models
  - Project with Novartis: longitudinal clinical trial w. multiple drugs, dosings, placebo control, hierarchical model of patient-level effects, meta-analysis
  - Collaborators: Frederic Bois, Amy Racine, Sebastian Weber

L-BFGS Optimizer

- Faster, lower memory, more stable than BFGS
- Plug-and-play in interfaces
- Code and doc complete; under testing

Higher-Order Auto-diff

- Finish higher-order auto-diff for probability functions
- May punt some cumulative distribution functions
  (Black art iterative algorithms required)

- Code complete; under testing

Riemannian Manifold HMC

- **NUTS** generalized to RHMC

  (Betancourt *arXiv* paper)

- **SoftAbs** metric
  - Eigendecompose Hessian
  - **positive definite** with positive eigenvalues
  - **condition** by narrowing eigenvalue range
  - Betancourt *arXiv* paper

- Code complete; awaiting higher-order auto-diff

Thermodynamic Sampler

- Physically motivated alternative to “simulated” **annealing and tempering** (not really simulated!)
- Supplies external **heat bath**
- Operates through **contact manifold**
- System relaxes more naturally between energy levels

- Prototype complete
  (Betancourt paper on *arXiv*; for geometers)

Marginal Maximum Likelihood

- Marginalize out hierarchical parameters
- Gradient-based nested optimization algorithm
- Errors estimated via curvature as in MLE

- Design complete; awaiting parameter tagging

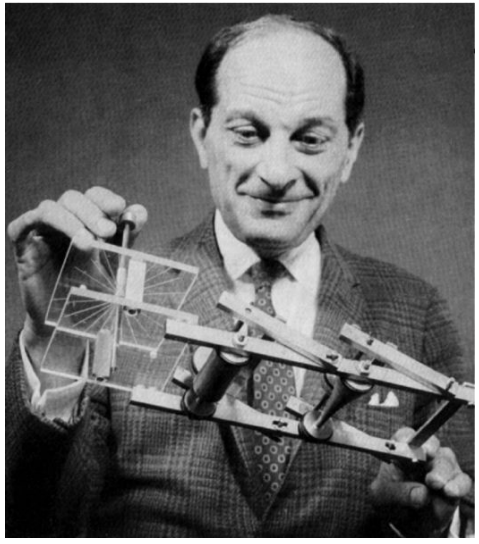
“Black Box” VB and EP

- **Black box** so can run any model
  (Laplace or other approximations)
- Stochastic, data-streaming **variational Bayes** (VB)
- Data-parallel **expectation propagation** (EP)
  (cavity distributions provide general shard combination)
- Both use point estimates of parametric approximation to posterior
- Optimize parameters to minimize KL divergence
  (VB, EP measure divergence in opposite directions)
- Prototype stage

The End

Stan's Namesake

- Stanislaw Ulam (1909–1984)
- Co-inventor of Monte Carlo method (and hydrogen bomb)



- Ulam holding the Fermiac, Enrico Fermi's physical Monte Carlo simulator for random neutron diffusion

Appendix I

Bayesian Data Analysis

Bayesian Data Analysis

- "By Bayesian data analysis, we mean practical methods for making inferences from data using probability models for quantities we observe and about which we wish to learn."
- "The essential characteristic of Bayesian methods is their **explict use of probability for quantifying uncertainty** in inferences based on statistical analysis."

Gelman et al., *Bayesian Data Analysis*, 3rd edition, 2013

Bayesian Mechanics

1. Set up full probability model
    - for all observable & unobservable quantities
    - consistent w. problem knowledge & data collection
2. Condition on observed data
    - caclulate posterior probability of unobserved quantities conditional on observed quantities
3. Evaluate
    - model fit
    - implications of posterior

Ibid.

Basic Quantities

- Basic Quantities
  - y: observed data
  - $\tilde{y}$: unknown, potentially observable quantities
  - θ: parameters (and other unobserved quantities)
  - x: constants, predictors for conditional models
- Random models for things that could've been otherwise
  - Everyone: Model data y as random
  - Bayesians: Model parameters θ as random

Distribution Naming Conventions

- **Joint**: $p(y, \theta)$
- **Sampling / Likelihood**: $p(y|\theta)$
- **Prior**: $p(\theta)$
- **Posterior**: $p(\theta|y)$
- **Data Marginal**: $p(y)$
- **Posterior Predictive**: $p(\tilde{y}|y)$

y modeled data, θ parameters, $\tilde{y}$ predictions,

implicit: $x, \tilde{x}$ unmodeled data (for y, $\tilde{y}$), size constants

Bayes's Rule for the Posterior

- Suppose the data y is fixed (i.e., observed). Then

$$
\begin{aligned}
p(\theta|y) &= \frac{p(y,\theta)}{p(y)} = \frac{p(y|\theta)\,p(\theta)}{p(y)} \\
&= \frac{p(y|\theta)\,p(\theta)}{\int p(y,\theta)\;d\theta} \\
&= \frac{p(y|\theta)\,p(\theta)}{\int p(y|\theta)\,p(\theta)\;d\theta} \\
&\propto p(y|\theta)\,p(\theta) = p(y,\theta)
\end{aligned}
$$

- Posterior proportional to likelihood times prior (i.e., joint)

Monte Carlo Methods

- For integrals that are impossible to solve analytically
- But for which sampling and evaluation is tractable
- Compute plug-in estimates of statistics based on randomly generated variates (e.g., means, variances, quantiles/intervals, comparisons)
- Accuracy with M (independent) samples proportional to

$$\frac{1}{\sqrt{M}}$$

  e.g., 100 times more samples per decimal place!

(Metropolis and Ulam 1949)

Monte Carlo Example

- Posterior expectation of θ:

$$\mathbb{E}[\theta|y] = \int \theta \; p(\theta|y) \; d\theta.$$

- Bayesian estimate minimizing expected square error:

$$\hat{\theta} = \arg\min_{\theta'} \mathbb{E}[(\theta - \theta')^2|y] = \mathbb{E}[\theta|y]$$

- Generate samples $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(M)}$ drawn from $p(\theta|y)$
- Monte Carlo Estimator plugs in average for expectation:

$$\mathbb{E}[\theta|y] \approx \frac{1}{M} \sum_{m=1}^{M} \theta^{(m)}$$

Monte Carlo Example II

- Bayesian alternative to frequentist hypothesis testing
- Use probability to summarize results
- Bayesian comparison: probability $\theta_1 > \theta_2$ given data y?

$$\begin{aligned}\Pr[\theta_1 > \theta_2 | y] &= \int\int \mathbb{I}(\theta_1 > \theta_2)\; p(\theta_1|y)\; p(\theta_2|y)\; d\theta_1\; d\theta_2 \\ &\approx \frac{1}{M}\sum_{m=1}^{M} \mathbb{I}(\theta_1^{(m)} > \theta_2^{(m)})\end{aligned}$$

- (Bayesian hierarchical model "adjusts" for multiple comparisons)

Markov Chain Monte Carlo

- When sampling independently from $p(\theta|y)$ impossible
- $\theta^{(m)}$ drawn via a Markov chain $p(\theta^{(m)}|y, \theta^{(m-1)})$
- Require MCMC marginal $p(\theta^{(m)}|y)$ equal to true posterior marginal
- Leads to auto-correlation in samples $\theta^{(1)}, \ldots, \theta^{(m)}$
- Effective sample size N_{eff} divides out autocorrelation (must be estimated)
- Estimation accuracy proportional to $1/\sqrt{N_{\text{eff}}}$

Gibbs Sampling

- Samples a parameter given data and other parameters
- Requires conditional posterior $p(\theta_n | y, \theta_{-n})$
- Conditional posterior easy in directed graphical model
- Requires general unidimensional sampler for non-conjugacy
  - JAGS uses slice sampler
  - BUGS uses adaptive rejection sampler
- Conditional sampling and general unidimensional sampler can both lead to slow convergence and mixing

(Geman and Geman 1984)

Metropolis-Hastings Sampling

- Proposes new point by changing all parameters randomly
- Computes accept probability of new point based on ratio of new to old log probability (and proposal density)
- Only requires evaluation of $p(\theta|y)$
- Requires good proposal mechanism to be effective
- Acceptance requires small changes in log probability
- But small step sizes lead to random walks and slow convergence and mixing

(Metropolis et al. 1953; Hastings 1970)