Stan

Probabilistic Programming Language

Core Development Team *(20 people, ~4 FTE)*

Andrew Gelman, **Bob Carpenter**, Matt Hoffman, Daniel Lee,
Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo,
Peter Li, Allen Riddell, Marco Inacio, Jeffrey Arnold,
Mitzi Morris, Rob Trangucci, Rob Goedman, Brian Lau,
Jonah Sol Gabray, Alp Kucukelbir, Robert L. Grant, Dustin Tran

Stan 2.7.0 (July 2015) http://mc-stan.org

Why Stan?

- ***Application***: Fit rich Bayesian statistical models

- *Problem*: Gibbs and Metropolis too slow (diffusive)
- *Solution*: Hamiltonian Monte Carlo (flow)

- *Problem*: Interpreters slow and unscalable
- *Solution*: Compiled to C++

- *Problem*: Need gradients of log posterior for HMC
- *Solution*: Reverse-mode algorithmic differentation

Why? (cont.)

- *Problem*: Existing algo-diff slow, limited, unextensible
- *Solution*: Our own algo-diff

- *Problem*: Algo-diff requires functions templated on all args
- *Solution*: Our own density library, Eigen linear algebra

- *Problem*: Need unconstrained parameters for HMC
- *Solution*: Variable transforms w. Jacobian determinants

Why? (cont.)

- *Problem*: Need ease of use of BUGS
- *Solution*: Compile a domain-specific language

- *Problem*: Pure directed graphical language inflexible
- *Solution*: Imperative probabilistic programming language

- *Problem*: Need to tune parameters for HMC
- *Solution*: Tune step size and estimate mass matrix during warmup; on-the-fly number of steps (NUTS)

Why? (cont.)

- *Problem*: Efficient up-to-proportion density calcs
- *Solution*: Density template metaprogramming
- *Problem*: Limited error checking, recovery
- *Solution*: Static model typing, informative exceptions
- *Problem*: Poor numerical stability
- *Solutions*: Taylor expansions, e.g., `log1p()`
  compound functions, e.g., `log_sum_exp()`, `BernoulliLogit()`
  limits at boundaries, e.g., `multiply_log()`

Why? (continued)

- *Problem*: Nobody knows everything
- *Solution*: Expand project team with specialists

- *Problem*: Expanding code and project team
- *Solution*: GitHub: branch, pull request, code review
- *Solution*: Jenkins: continuous integration
- *Solution*: ongoing refactoring and code simplification

Why? (continued)

- *Problem*: Heterogeneous user base
- *Solution*: More interfaces (R, Python, MATLAB, Julia)
- *Solution*: domain-specific examples, tutorials

- *Problem*: Restrictive licensing limits use
- *Solution*: Code and doc open source (BSD, CC-BY)

What is Stan?

- Stan is an **imperative** probabilistic programming language
  - cf., BUGS: declarative; Church: functional; Figaro: object-oriented
- Stan **program**
  - declares data and (constrained) parameter variables
  - defines log posterior (or penalized likelihood)
- Stan **inference**
  - MCMC for full Bayesian inference
  - Black-Box VB for approximate Bayes
  - MLE for penalized maximum likelihood estimation

Platforms and Interfaces

- **Platforms**: Linux, Mac OS X, Windows
- **C++ API**: portable, standards compliant (C++03)
- **Interfaces**
  - **CmdStan**: Command-line or shell interface (direct executable)
  - **RStan**: R interface (Rcpp in memory)
  - **PyStan**: Python interface (Cython in memory)
  - **MatlabStan**: MATLAB interface (external process)
  - **Stan.jl**: Julia interface (external process)
  - **StataStan**: Stata interface (external process) [under testing]
- **Posterior Visualization & Exploration**
  - **ShinyStan**: Shiny (R) web-based

Who’s Using Stan?

- 1200 **users group** registrations; 10,000 manual **downloads** (2.5.0); 100+ published papers
- **Biological sciences**: clinical drug trials, entomology, opthalmology, neurology, genomics, agriculture, botany, fisheries, cancer biology, epidemiology, population ecology, neurology
- **Physical sciences**: astrophysics, molecular biology, oceanography, climatology
- **Social sciences**: population dynamics, psycholinguistics, social networks, political science
- **Other**: materials engineering, finance, actuarial, sports, public health, recommender systems, educational testing

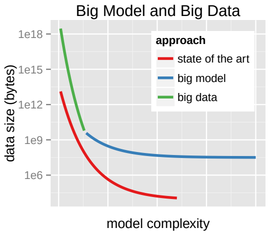
Documentation

- *Stan User's Guide and Reference Manual*
  - 500+ pages
  - Example models, modeling and programming advice
  - Introduction to Bayesian and frequentist statistics
  - Complete language specification and execution guide
  - Descriptions of algorithms (NUTS, R-hat, n_eff)
  - Guide to built-in distributions and functions
- Installation and getting started manuals by interface
  - RStan, PyStan, CmdStan, MatlabStan, Stan.jl, StataStan
  - RStan vignette

Books and Model Sets

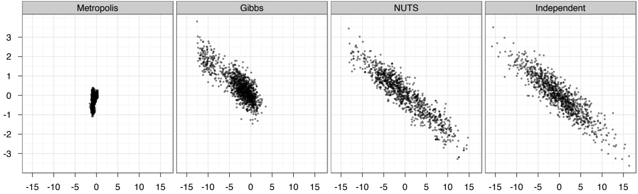
- **Model Sets** Translated to Stan
  - BUGS and JAGS examples (most of all 3 volumes)
  - Gelman and Hill (2009) *Data Analysis Using Regression and Multilevel/Hierarchical Models*
  - Wagenmakers and Lee (2014) *Bayesian Cognitive Modeling*
- **Books** with Sections on Stan
  - Gelman et al. (2013) *Bayesian Data Analysis*, 3rd Edition.
  - Kruschke (2014) *Doing Bayesian Data Analysis, Second Edition: A Tutorial with R, JAGS, and Stan*
  - Korner-Nievergelt et al. (2015) *Bayesian Data Analysis in Ecology Using Linear Models with R, BUGS, and Stan*

Scaling and Evaluation



- Types of Scaling: data, parameters, **models**
- Time to converge and per effective sample size:
  0.5–∞ times faster than BUGS & JAGS
- Memory usage: 1–10% of BUGS & JAGS

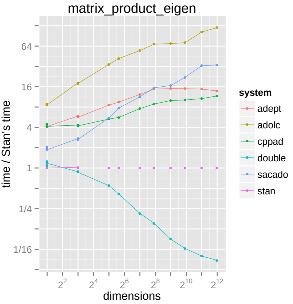
NUTS vs. Gibbs and Metropolis

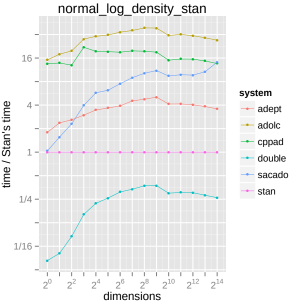


- Two dimensions of highly correlated 250-dim normal
- **1,000,000 draws** from Metropolis and Gibbs (thin to 1000)
- **1000 draws** from NUTS; 1000 independent draws

Stan's Autodiff vs. Alternatives

- Among **C++ open-source** offerings: Stan is **fastest** (for gradients), **most general** (functions supported), and **most easily extensible** (simple OO)





Part I
Stan Front End

Estimate Proportion

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

Maximum (Penalized) Likelihood

```
> library(rstan);
> N <- 5;
> y <- c(0,1,1,0,0);
> model <- stan_model("bernoulli.stan");
> mle <- optimizing(model, data=c("N", "y"));
...
> print(mle, digits=2)
$par                 $value  (log density)
theta                [1] -3.4
  0.4
```

- Posterior: Beta$(1+2, 1+3)$; mode 0.40; mean 0.43
- Density: MLE w/o Jacobian; MCMC with Jacobian

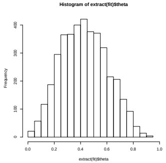
Bayesian Posterior

```
> 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    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 )
```



Default Priors and Vectorization

- All parameters are uniform by default
- Probability functions can be vectorized (more efficient)
- Thus

```
theta ~ uniform(0,1);
for (n in 1:N)
  y[n] ~ bernoulli(theta);
```

reduces to

```
y ~ bernoulli(theta);
```

Linear Regression

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

//  for (n in 1:N)
//     y[n] ~ normal(alpha + beta * x[n], sigma);
```

Logistic Regression (w. Matrices)

```
data {
  int<lower=1> K;
  int<lower=0> N;
  matrix[N,K] x;
  int<lower=0,upper=1> y[N];
}
parameters {
  vector[K] beta;
}
model {
   beta ~ cauchy(0, 2.5);          // prior
   y ~ bernoulli_logit(x * beta);  // likelihood
}
```

- vectorized default prior for regression coefficients
- vectorized, logit-scale; `y ~ bernoulli(inv_logit(x * beta))`

Time Series Autoregressive: AR(1)

```
data {
  int<lower=0> N;   vector[N] y;
}
parameters {
  real alpha;  real beta;  real sigma;
}
model {
  for (n in 2:N)
    y[n] ~ normal(alpha + beta * y[n-1], sigma);
}
```

- Likelihood more efficiently coded with vectorization as

```
tail(y, N - 1)
    ~ normal(alpha + beta * head(y, N - 1), sigma);
```

Generalized Linear Models

- Direct parameterizations more efficient and stable
- **Logistic regression** (boolean/binary data)
  - `y ~ bernoulli(inv_logit(eta));`
  - `y ~ bernoulli_logit(eta);`
  - Probit via `Phi` (normal cdf)
  - Robit (robust) via Student-*t* cdf
- **Poisson regression** (count data)
  - `y ~ poisson(exp(eta));`
  - `y ~ poisson_log(eta);`
  - Overdispersion with negative binomial

GLMS, continued

- **Multi-logit regression** (categorical data)
  - `y ~ categorical(softmax(eta));`
  - `y ~ categorical_logit(eta);`
- **Ordinal logistic regression** (ordered data)
  - Add cutpoints c
  - `y ~ ordered_logistic(eta, c);`
- **Robust linear regression** (overdispersed noise)
  - `y ~ student_t(nu, eta, sigma);`

Posterior Predictive Inference

- Parameters θ, observed data y, and data to predict $\tilde{y}$

$$p(\tilde{y}|y) = \int_{\Theta} p(\tilde{y}|\theta)\ p(\theta|y)\ d\theta$$

- ```
  data {
    int<lower=0> N_tilde;
    matrix[N_tilde,K] x_tilde;
    ...
  parameters {
    vector[N_tilde] y_tilde;
    ...
  model {
    y_tilde ~ normal(x_tilde * beta, sigma);
  ```
  ```

Predict w. Generated Quantities

- Replace sampling with pseudo-random number generation

```
generated quantities {
  vector[N_tilde] y_tilde;

  for (n in 1:N_tilde)
    y_tilde[n] <- normal_rng(x_tilde[n] * beta, sigma);
}
```

- Must include noise for predictive uncertainty
- PRNGs only allowed in generated quantities block
  - more computationally efficient per iteration
  - more statistically efficient with i.i.d. samples (i.e., MC, not MCMC)

Example: Gaussian Process Estimation

```
data {
  int<lower=1> N;  vector[N] x; vector[N] y;
} parameters {
  real<lower=0> eta_sq, inv_rho_sq, sigma_sq;
} transformed parameters {
  real<lower=0> rho_sq; rho_sq <- inv(inv_rho_sq);
} model {
  matrix[N,N] Sigma;
  for (i in 1:(N-1)) {
    for (j in (i+1):N) {
      Sigma[i,j] <- eta_sq * exp(-rho_sq * square(x[i] - x[j]));
      Sigma[j,i] <- Sigma[i,j];
  }}
  for (k in 1:N) Sigma[k,k] <- eta_sq + sigma_sq;
  eta_sq, inv_rho_sq, sigma_sq ~ cauchy(0,5);
  y ~ multi_normal(rep_vector(0,N), Sigma);
}
```

Gaussian Process Predictions

- Add predictors `x_tilde[M]` for points to predict
- Declare predicted values `y_tilde[M]` as unconstrained parameters
- Define `Sigma[M+N,M+N]` in terms of full `append_row(x, x_tilde)`
- Model remains the same

```
append_row(y,y_tilde)
  ~ multi_normal(rep(0,N+M),Sigma);
```

Mixture of Two Normals

```
for (n in 1:N) {
  real lp1;  real lp2;

  lp1 <- bernoulli_log(0, lambda)
           + normal_log(y[n], mu[1], sigma[1]);

  lp2 <- bernoulli_log(1, lambda)
           + normal_log(y[n], mu[2], sigma[2]);

  increment_log_prob(log_sum_exp(lp1,lp2));
```

- local variables reassigned; direct increment of log posterior
- $\mathrm{log\_sum\_exp}(\alpha, \beta) = \log(\exp(\alpha) + \exp(\beta))$
- **Much more efficient** than sampling (Rao-Blackwell Theorem)

Other Mixture Applications

- Other multimodal data
- Zero-inflated Poisson or hurdle models
- Model comparison or mixture
- Discrete change-point model
- Hidden Markov model, Kalman filter
- Almost anything with latent discrete parameters
- Other than variable choice, e.g., regression predictors
  - marginalization is exponential in number of vars

LKJ Density and Cholesky Factors

- Density on *correlation* matrices Ω
- $\text{LKJCorr}(\Omega \mid \nu) \propto \det(\Omega)^{(\nu-1)}$
  - $\nu = 1$ uniform
  - $\nu > 1$ concentrates around unit matrix
- Work with Cholesky factor L_Ω s.t. $\Omega = L_\Omega L_\Omega^\top$
  - Density: $\text{LKJCorrCholesky}(L_\Omega \mid \nu) \propto |J| \det(L_\Omega L_\Omega^\top)^{(\nu-1)}$
  - Jacobian adjustment for Cholesky factorization

Lewandowski, Kurowicka, and Joe (2009)

Covariance Random-Effects Priors

```
parameters {
  vector[2] beta[G];
  cholesky_factor_corr[2] L_Omega;
  vector<lower=0>[2] sigma;

model {
  sigma ~ cauchy(0, 2.5);
  L_Omega ~ lkj_cholesky(4);
  beta ~ multi_normal_cholesky(rep_vector(0, 2),
                       diag_pre_multiply(sigma, L_Omega));
  for (n in 1:N)
    y[n] ~ bernoulli_logit(... + x[n] * beta[gg[n]]);
```

- G groups with varying slope and intercept; gg indicates group

Dynamic Systems with Diff Eqs

- Simple harmonic oscillator

$$\frac{d}{dt} y_1 = -y_2 \qquad\qquad \frac{d}{dt} y_2 = -y_1 - \theta y_2$$

- Code as a function in Stan

```
functions {
  real[] sho(real t, real[] y, real[] theta,
             real[] x_r, int[] x_i) {
    real dydt[2];
    dydt[1] <- y[2];
    dydt[2] <- -y[1] - theta[1] * y[2];
    return dydt;
  }
}
```

Fit Noisy State Measurements

```
data {
  int<lower=1> T;        real y[T,2];
  real t0;               real ts[T];
}
parameters {
  real y0[2];                  // unknown initial state
  real theta[1];               // rates for equation
  vector<lower=0>[2] sigma;    // measurement error
}
model {
  real y_hat[T,2];
  ...priors...
  y_hat <- integrate_ode(sho, y0, t0, ts, theta, x_r, x_i);
  for (t in 1:T)
    y[t] ~ normal(y_hat[t], sigma);
}
```

Part II
What Stan Does

Full Bayes: No-U-Turn Sampler

- Adaptive **Hamiltonian Monte Carlo** (HMC)
  - **Potential Energy**: negative log posterior
  - **Kinetic Energy**: random standard normal per iteration
- Adaptation **during warmup**
  - step size adapted to target total acceptance rate
  - mass matrix (scale/rotation) estimated with regularization
- Adaptation **during sampling**
  - simulate forward and backward in time until U-turn
  - **slice sample** along path

(Hoffman and Gelman 2011, 2014)

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. MCMC std. error, std. dev., quantiles
  - split-$\hat{R}$ multi-chain convergence diagnostic (Gelman/Rubin)
  - multi-chain effective sample size estimation (FFT algorithm)
- Model comparison with **WAIC**
  - in-sample approximation to cross-validation

Penalized MLE

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

“Black Box” Variational Inference

- **Black box** so can fit any Stan model
- Multivariate **normal approx to unconstrained** posterior
  - covariance: diagonal mean-field or full rank
  - not Laplace approx — around posterior mean, not mode
  - transformed back to constrained space (built-in Jacobians)
- Stochastic **gradient-descent** optimization
  - ELBO gradient estimated via Monte Carlo + autdiff
- Returns **approximate posterior** mean / covariance
- Returns **sample** transformed to constrained space

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)

Part IV
Stan Language

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

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 first 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.

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));
  }

}
```

Differential Equation Solver

- System expressed as function
  - given state (y) time (t), parameters (θ), and data (x)
  - return derivatives ($\partial y / \partial t$) of state w.r.t. time
- Simple harmonic oscillator diff eq

```
real[] sho(real t,          // time
           real[] y,        // system state
           real[] theta,    // params
           real[] x_r,      // real data
           int[] x_i) {     // int data
  real dydt[2];
  dydt[1] <- y[2];
  dydt[2] <- -y[1] - theta[1] * y[2];
  return dydt;
}
```

Differential Equation Solver

- Solution via functional, given initial state (y0), initial time (t0), desired solution times (ts)

```
mu_y <- integrate_ode(sho, y0, t0, ts, theta, x_r, x_i);
```

- Use noisy measurements of y to estimate θ

```
y ~ normal(mu_y, sigma);
```

  - Pharmacokinetics/pharmacodynamics (PK/PD),
  - soil carbon respiration

Diff Eq Derivatives

- Need derivatives of solution w.r.t. parameters
- Couple derivatives of system w.r.t. parameters

$$\left(\frac{\partial}{\partial t} y, \quad \frac{\partial}{\partial t} \frac{\partial y}{\partial \theta} \right)$$

- Calculate coupled system via nested autodiff of second term

$$\frac{\partial}{\partial \theta} \frac{\partial y}{\partial t}$$

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);

Part V
Challenges for Stan

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)

Position-Dependent Curvature

- Mass matrix does **global** adaptation for
  - parameter scale (diagonal) and rotation (dense)
- Dense mass matrices hard to estimate ($\mathcal{O}(N^2)$ estimands)
- **Problem**: Position-dependent curvature
  - Example: banana-shaped densities
    - arise when parameter is product of other parameters
  - Example: hierarchical models
    - hierarhcical variance controls lower-level parameters
- Mitigate by reducing stepsize
  - initial (`stepsize`) and target acceptance (`adapt_delta`)

Part VI
Next for Stan

Higher-Order Auto-diff

- Forward-mode auto-diff for all functions
  - May punt some cumulative distribution functions
  - Black art iterative algorithms required

- Code complete; under testing

Riemannian Manifold HMC

- Best mixing MCMC method (fixed # of continuous params)
- Moves on Riemannian manifold rather than Euclidean
  - adapts to position-dependent curvature
- **geoNUTS** generalizes NUTS to RHMC (Betancourt *arXiv*)
- **SoftAbs** metric (Betancourt *arXiv*)
  - eigendecompose Hessian and condition
  - computationally feasible alternative to original Fisher info metric of Girolami and Calderhead (*JRSS, Series B*)
  - requires third-order derivatives and implicit integrator
- Code complete; awaiting higher-order auto-diff

Adiabatic Sampling

- 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
- Betancourt paper on *arXiv*
- Prototype complete

Maximum Marginal Likelihood

- Fast, Approximate Inference
- Marginalize out lower-level parameters
- Optimize higher-level parameters and fix
- Optimize lower-level parameters given higher-level
- Errors estimated as in MLE

- Design complete; awaiting parameter tagging

“Black Box” EP

- Fast, approximate inference (like VB)
  - VB and EP minimize divergence in opposite directions
  - especially useful for Gaussian processes
- Asynchronous, data-parallel **expectation propagation** (EP)
- Cavity distributions control subsample variance
- Prototypte stage
- collaborating with Seth Flaxman, Aki Vehtari, Pasi Jylänki, John Cunningham, Nicholas Chopin, Christian Robert

The End

Questions from Chad Scherer

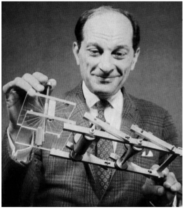
1. What were the **best design decisions** you made?
   - Anything you would do **differently**?
2. What new capability would most **change** the way probabilistic programming **is used**?
3. Any thoughts on characterizing some portion of the **design space** of probabilistic programming?
4. Any experience, ideas, or caveats on **“democratizing”** this kind of modeling?

Answers

- Best: define log density not graphical model, use C++
- Worst: use C++, define log density not graphical model
- Design: Stan programs use “random” variables, but only to define density and predictions imperatively (no metaprogramming, lacks modularity)
- Democratizing: remove the programming — scientists and statisticians want statistical/computational robustness, not a programming challenge
- New capabilities: Riemannian and adiabatic HMC for hard problems; VB, EP, and MML for fast approximation

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