Section 2.
Stan Components

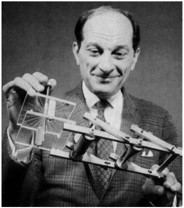
Bob Carpenter

Columbia University

Part I
Stan Top Level

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

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
  - VB for approximate Bayesian inference
  - 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); 300 Google scholar citations (100+ fitting)
- **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

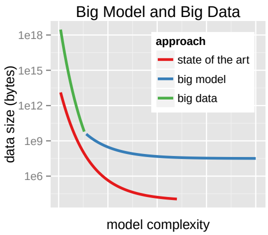
Documentation

- *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
  - 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*

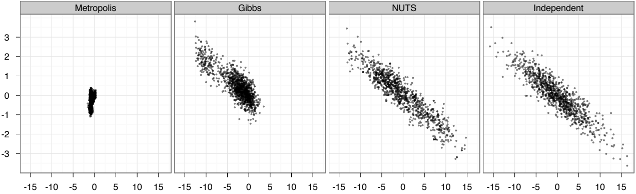
Scaling and Evaluation



- 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

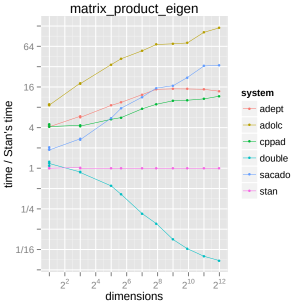
NUTS vs. Gibbs and Metropolis

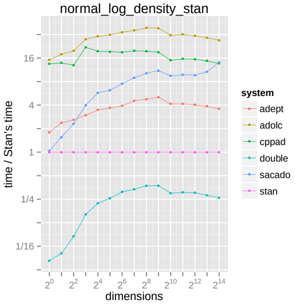


- Two dimensions of highly correlated 250-dim normal
- **1,000,000 draws** from Metropolis and Gibbs (thin to 1000)
- **1000 draws** from NUTS; 1000 independent draws

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 II
Stan Language

Stan is a Programming Language

- **Not** a graphical specification language like BUGS or JAGS
- Stan is a Turing-complete imperative programming language for specifying differentiable log densities
  - reassignable local variables and scoping
  - full conditionals and loops
  - functions (including recursion)
- With automatic "black-box" inference on top (though even that is tunable)
- Programs computing same thing may have different efficiency

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)

Integers vs. Reals

- Different types (conflated in BUGS, JAGS, and R)
- Distributions and assignments care
- Integers may be assigned to reals but not vice-versa
- Reals have not-a-number, and positive and negative infinity
- Integers single-precision up to +/- 2 billion
- Integer division rounds (Stan provides warning)
- Real arithmetic is inexact and reals should not be (usually) compared with ==

Arrays vs. Matrices

- Stan separates arrays, matrices, vectors, row vectors
- Which to use?
- Arrays allow most efficient access (no copying)
- Arrays stored first-index major (i.e., 2D are row major)
- Vectors and matrices required for matrix and linear algebra functions
- Matrices stored column-major
- Are not assignable to each other, but there are conversion functions

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, etc.
- **Ordering, Slicing, Broadcasting**: sort, rank, block, rep
- **Reductions**: sum, product, norms
- **Specializations**: triangular, positive-definite,

User-Defined Functions

- **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 with biomass input and breakdown

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);`
- **Reject**: `reject("arg to foo must be positive, found y=", y);`

“Sampling” Increments Log Prob

- A Stan program defines a log posterior
  - typically through log joint and Bayes’s rule
- Sampling statements are just “syntactic sugar”
- A shorthand for incrementing the log posterior
- The following define the same* posterior
  - `y ~ poisson(lambda);`
  - `increment_log_prob(poisson_log(y, lamda));`
- * up to a constant
- Sampling statement drops constant terms

Local Variable Scope Blocks

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

  is more efficient with sufficient statistics

  ```
  {
    real sum_y;  // local variable
    sum_y <- 0;
    for (n in 1:N)
      sum_y <- a + y[n];   // reassignment
    sum_y ~ binomial(N, theta);
  }
  ```

- Simpler, but roughly same efficiency:

  ```
  sum(y) ~ binomial(N, theta);
  ```
  ```

Print and Reject

- Print statements are for **debugging**
  - printed every log prob evaluation
  - print values in the middle of programs
  - check when log density becomes undefined
  - can embed in conditionals
- Reject statements are for **error checking**
  - typically function argument checks
  - cause a rejection of current state (0 density)

Prob Function Vectorization

- Stan's probability functions are vectorized for speed
  - removes repeated computations (e.g., $-\log\sigma$ in normal)
  - reduces size of expression graph for differentation
- Consider: `y ~ normal(mu, sigma);`
- Each of `y`, `mu`, and `sigma` may be any of
  - scalars (integer or real)
  - vectors (row or column)
  - 1D arrays
- All dimensions must be scalars or having matching sizes
- Scalars are broadcast (repeated)

Part III
Transformed Parameters

Transforms: Precision

```
parameters {
  real<lower=0> tau;  // precision
  ...
}
transformed parameters {
  real<lower=0> sigma;  // sd
  sigma <- 1 / sqrt(tau);
}
```

Transforms: “Matt Trick”

```
parameters {
  vector[K] beta_raw;  // non-centered
  real mu;
  real<lower=0> sigma;
}
transformed parameters {
  vector[K] beta;  // centered
  beta <- mu + sigma * beta_raw;
}
model {
  mu ~ cauchy(0, 2.5);
  sigma ~ cauchy(0, 2.5);
  beta_raw ~ normal(0, 1);
}
```

Part IV
Generated Quantities for Prediction

Linear Regression (Normal Noise)

- **Likelihood**:
  - $y_n = \alpha + \beta x_n + \epsilon_n$
  - $\epsilon_n \sim \text{Normal}(0, \sigma)$ for $n \in 1{:}N$
- Equivalently,
  - $y_n \sim \text{Normal}(\alpha + \beta x_n, \sigma)$
- **Priors** (improper)
  - $\sigma \sim \text{Uniform}(0, \infty)$
  - $\alpha, \beta \sim \text{Uniform}(-\infty, \infty)$
- Stan allows **improper prior**; requires **proper posterior**.

Linear Regression in Stan

```
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);
```

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)

End (Section 2)