Stan

a Probabilistic Programming Language

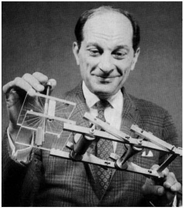
Core Development Team *($\sim$ 4 FTE):*

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 Gabry, Alp Kucukelbir, Robert L. Grant, Dustin Tran
Krzysztof Sakrejda, Aki Vehtari, Rayleigh Lei
Sebastian Weber

Stan 2.9.0 (April 2016) `http://mc-stan.org`

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

Stan Example

Repeated Binary Trials

Stan Program

```
data {
  int<lower=0> N;                    // number of trials
  int<lower=0, upper=1> y[N];        // success on trial n
}
parameters {
  real<lower=0, upper=1> theta;      // chance of success
}
model {
  theta ~ uniform(0, 1);             // prior
  for (n in 1:N)
    y[n] ~ bernoulli(theta);         // likelihood
}
```

A Stan Program

- Defines log (posterior) density up to constant, so...
- Equivalent to define log density directly:

```
model {
  increment_log_prob(0);
  for (n in 1:N)
    increment_log_prob(log(theta^y[n])
                          * (1 - theta)^(1 - y[n])));
}
```

- Also equivalent to (a) drop constant prior and (b) vectorize likelihood:

```
model {
  y ~ bernoulli(theta);
}
```

R: Simulate Data

- Generate data

```
> theta <- 0.30;
> N <- 20;
> y <- rbinom(N, 1, 0.3);

> y

 [1] 1 1 1 1 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1
```

- Calculate MLE as sample mean from data

```
> sum(y) / N

[1] 0.4
```

RStan: Fit

```
> library(rstan);

> fit <- stan("bern.stan",
              data = list(y = y, N = N));

> print(fit, probs=c(0.1, 0.9));

Inference for Stan model: bern.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000,
total post-warmup draws=4000.

        mean se_mean   sd   10%   90% n_eff Rhat
theta   0.41    0.00 0.10  0.28  0.55  1580    1
```

Plug in Posterior Draws

- Extracting the posterior draws

```
> theta_draws <- extract(fit)$theta;
```

- Calculating posterior mean (estimator)

```
> mean(theta_draws);

[1] 0.4128373
```

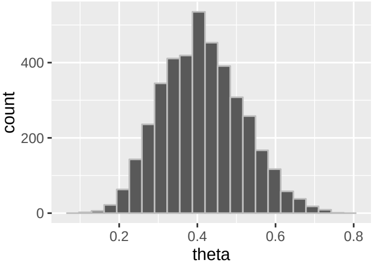
- Calculating posterior intervals

```
> quantile(theta_draws, probs=c(0.10, 0.90));

      10%       90%
0.2830349 0.5496858
```

ggplot2: Plotting

```
theta_draws_df <- data.frame(list(theta = theta_draws));
plot <-
  ggplot(theta_draws_df, aes(x = theta)) +
  geom_histogram(bins=20, color = "gray");
plot;
```



Example

Fisher "Exact" Test

Bayesian "Fisher Exact Test"

- Suppose we observe the following data on handedness

	sinister	*dexter*	TOTAL
male	9 (y_1)	43	52 (N_1)
female	4 (y_2)	44	48 (N_2)

- Assume likelihoods $\text{Binomial}(y_k | N_k, \theta_k)$, uniform priors
- Are men more likely to be lefthanded?

$$\begin{aligned} \Pr[\theta_1 > \theta_2 \mid y, N] &= \int_\Theta \mathrm{I}[\theta_1 > \theta_2]\, p(\theta | y, N)\, d\theta \\ &\approx \frac{1}{M} \sum_{m=1}^{M} \mathrm{I}[\theta_1^{(m)} > \theta_2^{(m)}]. \end{aligned}$$

Stan Binomial Comparison

```
data {
  int y[2];
  int N[2];
}
parameters {
  vector<lower=0,upper=1> theta[2];
}
model {
  y ~ binomial(N, y);
}
generated quantities {
  real boys_minus_girls;
  int boys_gt_girls;
  boys_minus_girls <- theta[1] - theta[2];
  boys_gt_girls <- (theta[1] > theta[2]);
}
```

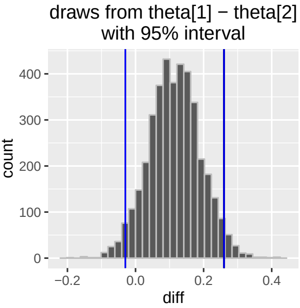
Results

	mean	2.5%	97.5%
theta[1]	0.22	0.12	0.35
theta[2]	0.11	0.04	0.21
boys_minus_girls	0.12	-0.03	0.26
boys_gt_girls	0.93	0.00	1.00

- $\Pr[\theta_1 > \theta_2 \mid y] \approx 0.93$
- $\Pr[(\theta_1 - \theta_2) \in (-0.03, 0.26) \mid y] = 95\%$

Visualizing Posterior Difference

- Plot of posterior difference, $p(\theta_1 - \theta_2 \mid y, N)$ (men - women)



- Vertical bars: central 95% posterior interval $(-0.03, 0.26)$

Example

More Stan Models

Posterior Predictive Distribution

- Predict new data ($\tilde{y}$) given observed data (y)
- Includes two kinds of uncertainty
  - parameter estimation uncertainty: $p(\theta|y)$
  - sampling uncertainty: $p(\tilde{y}|\theta)$

$$\begin{aligned} p(\tilde{y}|y) &= \int p(\tilde{y}|\theta)\ p(\theta|y)\ \mathrm{d}\theta \\ &\approx \frac{1}{M}\sum_{m=1}^{M} p(\tilde{y}|\theta^{(m)}) \end{aligned}$$

- Can generate predictions as sample of draws $\tilde{y}^{(m)}$ based on $\theta^{(m)}$

Linear Regression with Prediction

```
data {
  int<lower=0> N;                int<lower=0> K;
  matrix[N, K] x;                vector[N] y;
  matrix[N_tilde, K] x_tilde;
}
parameters {
  vector[K] beta;                real<lower=0> sigma;
}
model {
  y ~ normal(x * beta, sigma);
}
generated quantities {
  vector[N_tilde] y_tilde;
  for (n in 1:N_tilde)
    y_tilde[n] <- normal_rng(x_tilde[n] * beta, sigma);
}
```

Transforming Precision

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

Logistic Regression

```
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
}
```

Time Series Autoregressive: AR(1)

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

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]]);
```

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);
}
```

Non-Centered Parameterization

```
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);
}
```

Overview

What is Stan?

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; C++11 soon)
- **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)
- **Posterior Visualization & Exploration**
  - **ShinyStan**: Shiny (R) web-based

Higher-Level Interfaces

- **R Interfaces**
  - **RStanArm**: Regression modeling with R expressions
  - **ShinyStan**: Web-based posterior visualization, exploration
  - **Loo**: Approximate leave-one-out cross-validation
- **Containers**
  - Dockerized Jupyter (iPython) Notebooks (R, Python, or Julia)

Who's Using Stan?

- 1800+ **users group** registrations; 15,000+ **downloads** (per version just in Rstudio); 400+ Google scholar citations
- **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, biogeochemistry
- **Social sciences**: population dynamics, psycholinguistics, social networks, political science, surveys
- **Other**: materials engineering, finance, actuarial, sports, public health, recommender systems, educational testing, equipment maintenance

Documentation

- *Stan User's Guide and Reference Manual*
  - 550+ (short) 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

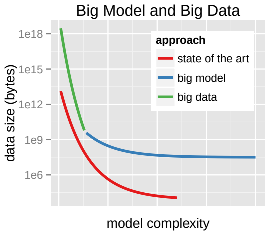
Model Sets Translated to Stan

- BUGS 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*
- Kéry and Schaub (2014) *Bayesian Population Analysis Using WinBUGS*

Books all or partly about Stan

- McElreath (2016) *Statistical Rethinking: A Bayesian course with R and Stan*
- Korner-Nievergelt et al. (2015) *Bayesian Data Analysis in Ecology Using Linear Models with R, BUGS, and Stan*
- Kruschke (2014) *Doing Bayesian Data Analysis, Second Edition: A Tutorial with R, JAGS, and Stan*
- Gelman et al. (2013) *Bayesian Data Analysis*, 3rd Edition.
- More in prep (including two written by the Stan developers, one basic and one for econometrics)

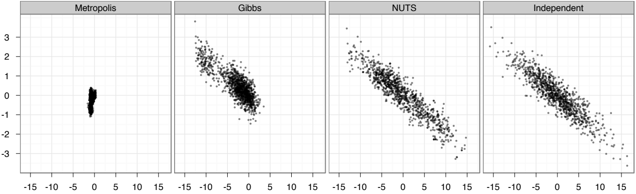
Scaling and Evaluation



- 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

Overview
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

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

Model: Read and Transform Data

- Only done once for optimization or sampling (per chain)
- Read data
  - read data variables from memory or file stream
  - validate data
- Generate transformed data
  - execute transformed data statements
  - validate variable constraints when done

Model: Log Density

- *Given* parameter values on unconstrained scale
- Builds expression graph for log density (start at 0)
- Inverse transform parameters to constrained scale
  - constraints involve non-linear transforms
  - e.g., positive constrained x to unconstrained $y = \log x$
- account for curvature in change of variables
  - e.g., unconstrained y to positive $x = \log^{-1}(y) = \exp(y)$
  - e.g., add log Jacobian determinant, $\log|\frac{d}{dy}\exp(y)| = y$
- Execute model block statements to increment log density

Model: Log Density Gradient

- Log density evaluation builds up expression graph
  - templated overloads of functions and operators
  - efficient arena-based memory management
- Compute gradient in backward pass on expression graph
  - propagate partial derivatives via chain rule
  - work backwards from final log density to parameters
  - dynamic programming for shared subexpressions
- Linear multiple of time to evalue log density

Model: Generated Quantities

- **Given** parameter values
- Once per iteration (not once per leapfrog step)
- May involve (pseudo) random-number generation
  - Executed generated quantity statements
  - Validate values satisfy constraints
- Typically used for
  - Event probability estimation
  - Predictive posterior estimation
- Efficient because evaluated with `double` types (no autodiff)

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
  - adjustment for MCMC and variational, not MLE
  - add log determinant of inverse transform Jacobian
  - automatically differentiable

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,

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);
  ```
  ```

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

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

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)

Diving Deeper

Stan's Autodiff

Stan's Reverse-Mode

- Easily extensible **object-oriented** design
- **Code nodes** in expression graph for primitive functions
  - requires **partial derivatives**
  - built-in flexible abstract base classes
  - **lazy evaluation** of chain rule saves memory
- Autodiff through templated C++ functions
  - templating on each argument avoids excess promotion

Stan's Reverse-Mode (cont.)

- Arena-based **memory management**
  - specialized C++ `operator new` for reverse-mode variables
  - custom functions inherit memory management through base
- Nested application to support ODE solver

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

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

- Based on Eigen's Odeint package (RK45 non-stiff solver)

Stiff Diff Eqs

- Coming in Stan 2.10 (any day)
- Based on CVODES implementation of BDF (Sundials)
- CVODES builds-in efficient structure for sensitivity
- Even more autodiff for system Jacobian

Stan’s Forward Mode

- Templated scalar type for value and tangent
  - allows higher-order derivatives
- Primitive functions propagate derivatives
- No need to build expression graph in memory
  - much less memory intensive than reverse mode
- Autodiff through templated functions (as reverse mode)

Second-Order Derivatives

- Compute Hessian (matrix of second-order partials)

$$H_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(x)$$

- Required for Laplace covariance approximation (MLE)
- Required for curvature (Riemannian HMC)
- Nest reverse-mode in forward for **second order**
- N forward passes: takes gradient of derivative

Third-Order Derivatives

- Compute gradients of Hessians (tensor of third-order partials)

$$\frac{\partial^3}{\partial x_i \partial x_j \partial x_k} f(x)$$

  - Required for SoftAbs metric (Riemannian HMC)
  - N^2 forward passes: gradient of derivative of derivative

- Can do this, but don't need it

- Clever way to compute what we need in quadratic time:
  - $\nabla \mathrm{tr}(H\,M)$
  - where H is the hessian and M is a fixed matrix

Jacobians

- Assume function $f : \mathbb{R}^N \to \mathbb{R}^M$
- Partials for multivariate function (matrix of first-order partials)

$$J_{i,j} = \frac{\partial}{\partial x_i} f_j(x)$$

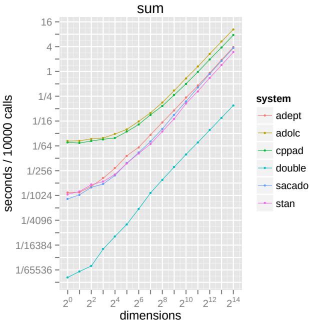
- Required for stiff ordinary differential equations
  - differentiate is coupled sensitivity autodiff for ODE system
- Two execution strategies
  1. Multiple reverse passes for rows
  2. Forward pass per column (required for stiff ODE)

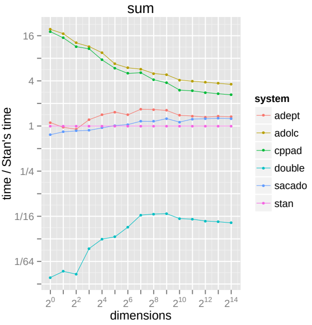
Autodiff Functionals

- Functionals map templated functors to derivatives
  - fully encapsulates and hides all autodiff types
- Autodiff functionals supported: cost relative to function
  - gradients: $\mathcal{O}(1)$
  - Jacobians: $\mathcal{O}(N)$
  - gradient-vector product (i.e., directional derivative): $\mathcal{O}(1)$
  - Hessian-vector product: $\mathcal{O}(N)$
  - Hessian: $\mathcal{O}(N)$
  - gradient of trace of matrix-Hessian product: $\mathcal{O}(N)$
    (for SoftAbs RHMC)

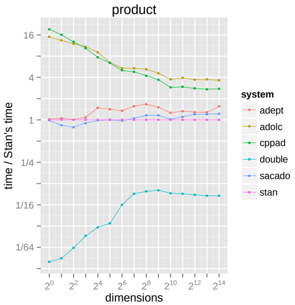
Stan's Autodiff vs. Alternatives

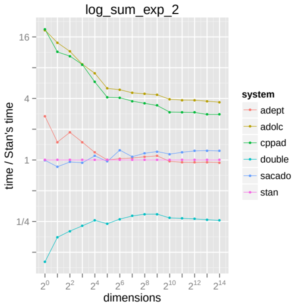
- Stan is **fastest** and uses least memory
  - among open-source C++ alternatives we managed to install





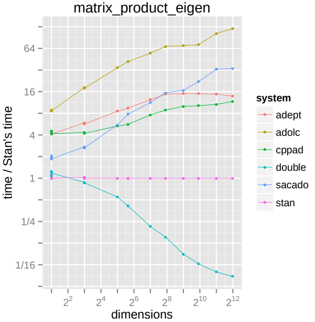
Product & Log-Sum-Exp

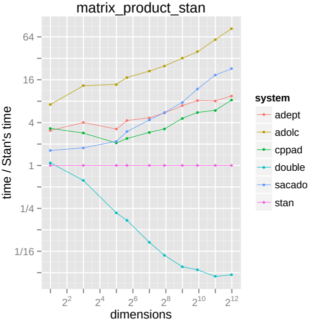




Stan’s Matrix Calculations

- Faster in Eigen, but takes more memory
- Best of both worlds coming soon





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

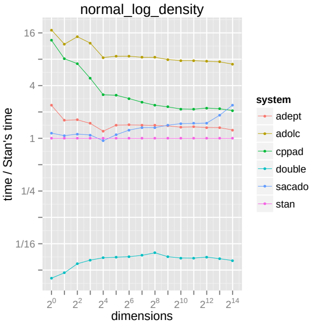
$$\begin{aligned}\log \text{Normal}(y|\mu,\sigma) &= \textstyle\sum_{n=1}^{N} \log \text{Normal}(y_n|\mu,\sigma) \\ &= \textstyle\sum_{n=1}^{N} -\log\sqrt{2\pi} \;-\log\sigma \;-\frac{y_n-\mu}{2\sigma^2}\end{aligned}$$

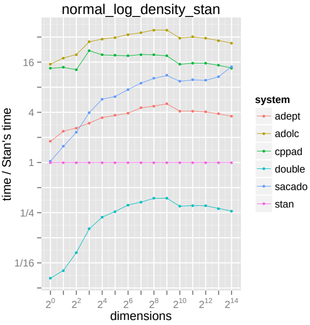
- recursive **expression templates** to broadcast and cache scalars, generalize containers (arrays, matrices, vectors)

- **traits** metaprogram to **drop constants** (e.g., $-\log\sqrt{2\pi}$ or $\log\sigma$ if constant) and calculate intermediate and return types

Stan's Density Calculations

- Vectorization a huge win





Deepr Still

Autodiff Coding

Variable Ptr to Impl

```
class var {
public:
  var() : vi_(static_cast<vari*>(0U)) { }
  var(double v) : vi_(new vari(v)) { }

  double val() const { return vi_->val_; }
  double adj() const { return vi_->adj_; }

private:
  vari* vi_;
};
```

Chainable Base Class

```
struct chainable {
  chainable() { }
  virtual ~chainable() { }

  virtual void chain() { }
  virtual void init_dependent() { }
  virtual void set_zero_adjoint() { }

  static inline void* operator new(size_t nbytes) {
    return ChainableStack::memalloc_.alloc(nbytes);
  }
};
```

Variable Implementation

```
class vari : public chainable {
public:
  const double val_;
  double adj_;

  vari(double v) : val_(v), adj_(0) {
    ChainableStack::var_stack_.push_back(this);
  }

  virtual ~vari() { }

  virtual void init_dependent() { adj_ = 1; }
  virtual void set_zero_adjoint() { adj_ = 0; }
};
```

Memory Management

```
struct AutodiffStackStorage {
  static std::vector<chainable*> var_stack_;
  static stack_alloc memalloc_;
};

class stack_alloc {
private:
  std::vector<char*> blocks_;
  std::vector<size_t> sizes_;
  size_t cur_block_;
  char* cur_block_end_;
  char* next_loc_;
  ...
};
```

Conditional Execution Paths

```
#ifdef __GNUC__
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
#else
#define likely(x)      (x)
#define unlikely(x)    (x)
#endif
```

Block Allocation

```
inline void* alloc(size_t len) {
  char* result = next_loc_;
  next_loc_ += len;
  if (unlikely(next_loc_ >= cur_block_end_))
    result = move_to_next_block(len);
  return static_cast<void*>(result);
}
```

Gradient Calculation

```
static void grad(chainable* vi) {
  typedef std::vector<chainable*>::reverse_iterator it_t;
  vi->init_dependent();
  it_t begin = ChainableStack::var_stack_.rbegin();
  it_t end = ChainableStack::var_stack_.rend();
  for (it_t it = begin; it < end; ++it)
    (*it)->chain();
}
```

Unary Function

```
struct op_v_vari : public vari {
  vari* avi_;

  op_v_vari(double f, vari* avi) : vari(f), avi_(avi) { }
};
```

Logarithm Implementation

```
struct log_vari : public op_v_vari {
  log_vari(vari* avi) :
    op_v_vari(std::log(avi->val_), avi) { }

  void chain() {
    avi_->adj_ += adj_ / avi_->val_;
  }
};

inline var log(const var& a) {
  return var(new log_vari(a.vi_));
}
```

Addition Operator

```
inline var operator+(const var& a, const var& b) {
  return var(new add_vv_vari(a.vi_, b.vi_));
}

struct add_vari  ...
  void chain() {
    avi_->adj_ += adj_;
    bvi_->adj_ += adj_;
  }

struct product_vari ...
  void chain() {
    avi_->adj_ += adj_ * b_.val();
    bvi_->adj_ += adj_ * a_.val();
  }
```

Functor for Function

```
struct normal_ll {
  const Matrix<double, Dynamic, 1> y_;

  normal_ll(const Matrix<double, Dynamic, 1>& y) : y_(y) { }

  template <typename T>
  T operator()(const Matrix<T, Dynamic, 1>& theta) const {
    T mu = theta[0];
    T sigma = theta[1];
    T lp = 0;
    for (int n = 0; n < y_.size(); ++n)
      lp += normal_log(y_[n], mu, sigma);
    return lp;
  }
};
```

Gradient Functional: Use

```
Matrix<double, Dynamic, 1> y(3);
y << 1.3, 2.7, -1.9;
normal_ll f(y);

Matrix<double, Dynamic, 1> x(2);
x << 1.3, 2.9;

double fx;
Matrix<double, Dynamic, 1> grad_fx;
stan::math::gradient(f, x, fx, grad_fx);
```

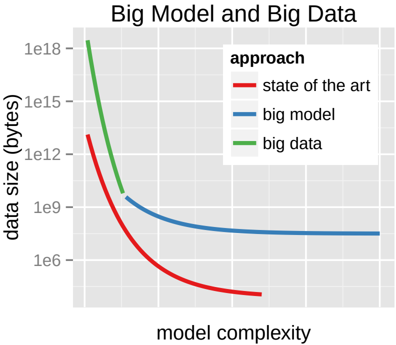
Gradient Functional

```
template <typename F>
void gradient(const F& f,  const VectorXd& x,
              double& fx,  VectorXd& grad_fx) {
  try {
    Matrix<var, Dynamic, 1> x_var(x.size());
    for (int i = 0; i < x.size(); ++i) x_var(i) = x(i);
    var fx_var = f(x_var);
    fx = fx_var.val();
    grad(fx_var.vi_);
    grad_fx.resize(x.size());
    for (int i = 0; i < x.size(); ++i)
      grad_fx(i) = x_var(i).adj();
  } catch (const std::exception& /*e*/) {
    recover_memory();   throw;
  }
  recover_memory();
}
```

Appendix

Stan for Big(ger) Data

Scaling and Evaluation



- Types of Scaling: data, parameters, **models**

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
- merged with develop branch

Maximum Marginal Likelihood

- Fast, approx. inference for hierarchical models: $p(\phi, \alpha)$
- Marginalize out lower-level params: $p(\phi) = \int p(\phi, \alpha) d\alpha$
- Optimize higher-level parameters ϕ^* and fix
- Optimize lower-level parameters given higher-level: $p(\phi^*, \alpha)$
- Errors estimated as in MLE
- aka "empirical Bayes"
  - but not fully Bayesian
  - and no more empirical than full Bayes
- Prototypes in R working

Laplace Approximation

- Multivariate normal approximation to posterior
- Compute posterior mode via optimization

$$\theta^* = \text{arg max}_\theta \; p(\theta|y)$$

- Laplace approximation to the posterior is

$$p(\theta|y) \approx \text{MultiNormal}(\theta^* | -H^{-1})$$

- H is the Hessian of the log posterior

$$H_{i,j} = \frac{\partial^2}{\partial\theta_i \; \partial\theta_j} \log p(\theta|y)$$

Stan's Laplace Approximation

- Operates on unconstrained parameters
- L-BFGS to compute posterior mode $\theta*$
- Automatic differentiation to compute H
  - current R: finite differences of gradients
  - soon: second-order automatic differentiation
- Draw a sample from approximate posterior
  - transfrom back to constrained scale
  - allows Monte Carlo computation of expectations

“Black Box” Variational Inference

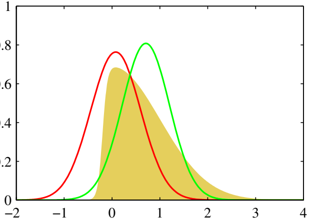
- **Black box** so can fit any Stan model
- Multivariate **normal approx to unconstrained** posterior
  - covariance: diagonal mean-field or full rank
  - not Laplace approx — around posterior mean, not mode
  - transformed back to constrained space (built-in Jacobians)
- Stochastic **gradient-descent**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

VB in a Nutshell

- y is observed data, θ parameters
- Goal is to approximate posterior $p(\theta|y)$
- with a convenient approximating density $g(\theta|\phi)$
  - ϕ is a vector of parameters of approximating density
- Given data y, VB computes ϕ^* minimizing KL-divergence

$$
\begin{aligned}
\phi^* &= \text{arg}\,\text{min}_\phi \ \text{KL}[g(\theta|\phi) \mid\mid p(\theta|y)] \\
&= \text{arg}\,\text{min}_\phi \int_\Theta \log\left(\frac{p(\theta \mid y)}{g(\theta \mid \phi)}\right) g(\theta|\phi)\ \mathrm{d}\theta \\
&= \text{arg}\,\text{min}_\phi \ \mathbb{E}_{g(\theta|\phi)}\left[\log p(\theta \mid y) - \log g(\theta \mid \phi)\right]
\end{aligned}
$$

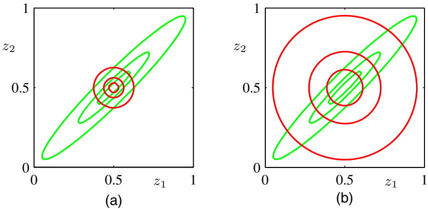
VB vs. Laplace



- *solid yellow*: target; *red*: Laplace; *green*: VB
- **Laplace** located at posterior mode
- **VB** located at approximate posterior mean

— Bishop (2006) *Pattern Recognition and Machine Learning*, fig. 10.1

KL-Divergence Example



- **Green**: true distribution p; **Red**: best approximation g
  - (a) VB-like: KL$[g\,||\,p]$
  - (b) EP-like: KL$[p\,||\,g]$
- VB systematically **understimates posterior variance**

— Bishop (2006) *Pattern Recognition and Machine Learning*, fig. 10.2

Stan's "Black-Box" VB

- Typically custom $g()$ per model
  - based on conjugacy and analytic updates
- Stan uses "black-box VB" with multivariate Gaussian g

$$g(\theta|\phi) = \mathsf{MultiNormal}(\theta \mid \mu, \Sigma)$$

  for the **unconstrained posterior**
  - e.g., scales σ log-transformed with Jacobian
- Stan provides two versions
  - Mean field: Σ diagonal
  - General: Σ dense

Stan's VB: Computation

- Use L-BFGS optimization to optimize θ
- Requires gradient of KL-divergence w.r.t. θ up to constant
- Approximate KL-divergence and gradient via Monte Carlo
  - only need approximate gradient calculation for soundness of L-BFGS
  - KL divergence is an expectation w.r.t. approximation $g(\theta|\phi)$
  - Monte Carlo draws i.i.d. from approximating multi-normal
  - derivatives with respect to true model log density via reverse-mode autodiff
  - so only a few Monte Carlo iterations are enough

Stan's VB: Computation (cont.)

- To support compatible plug-in inference
  - draw Monte Carlo sample $\theta^{(1)}, \ldots, \theta^{(M)}$ with

$$\theta^{(m)} \sim \text{MultiNormal}(\theta \mid \mu^{*}, \Sigma^{*})$$

  - inverse transfrom from unconstrained to constrained scale
  - report to user in same way as MCMC draws

- Future: reweight $\theta^{(m)}$ via importance sampling
  - with respect to true posterior
  - to improve expectation calculations

Near Future: Stochastic VB

- Data-streaming form of VB
  - Scales to billions of observations
  - Hoffman et al. (2013) Stochastic variational inference. *JMLR* 14.
- Mashup of stochastic gradient (Robbins and Monro 1951) and VB
  - subsample data (e.g., stream in minibatches)
  - upweight each minibatch to full data set size
  - use to make unbiased estimate of true gradient
  - take gradient step to minimimize KL-divergence
- Prototype code complete

The End