Stan:

a Probabilistic Programming Language

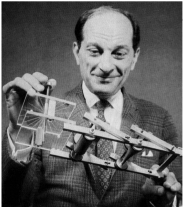
Stan Development Team (in order of joining):

Andrew Gelman, **Bob Carpenter**, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Allen Riddell, Marco Inacio, Jeffrey Arnold, **Mitzi Morris**, Rob Trangucci, Rob Goedman, Brian Lau, Jonah Sol Gabry, Robert L. Grant, Krzysztof Sakrejda, Aki Vehtari, Rayleigh Lei, Sebastian Weber, Charles Margossian, Vincent Picaud Imad Ali, Sean Talts, Ben Bales, Ari Hartikainen

Stan 2.17.0 (October 2017) 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

Prerequisite

Bayesian Inference

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 Methodology

- Set up **full probability model**
  - for all observable & unobservable quantities
  - consistent w. problem knowledge & data collection
- **Condition** on observed data (where Stan comes in!)
  - to caclulate posterior probability of unobserved quantities (e.g., parameters, predictions, missing data)
- **Evaluate**
  - model fit and implications of posterior
- **Repeat** as necessary

Ibid.

Where do Models Come from?

- Sometimes model comes first, based on substantive considerations
  - toxicology, economics, ecology, physics, . . .
- Sometimes model chosen based on data collection
  - traditional statistics of surveys and experiments
- Other times the data comes first
  - observational studies, meta-analysis, . . .
- Usually its a mix

Model Checking

- Do the inferences make sense?
  - are parameter values consistent with model's prior?
  - does simulating from parameter values produce reasoable fake data?
  - are marginal predictions consistent with the data?
- Do predictions and event probabilities for new data make sense?
- ***Not***: Is the model true?
- ***Not***: What is Pr[model is true]?
- ***Not***: Can we "reject" the model?

Model Improvement

- Expanding the model
  - hierarchical and multilevel structure . . .
  - more flexible distributions (overdispersion, covariance)
  - more structure (geospatial, time series)
  - more modeling of measurement methods and errors
  - . . .
- Including more data
  - breadth (more predictors or kinds of observations)
  - depth (more observations)

Using Bayesian Inference

- Explores full range of parameters consistent with prior info and data*
  - * if such agreement is possible
  - Stan automates this procedure with diagnostics
- Inferences can be plugged in directly for
  - parameter estimates minimizing expected error
  - predictions for future outcomes with uncertainty
  - event probability updates conditioned on data
  - risk assesment / decision analysis conditioned on uncertainty

Notation for Basic Quantities

- **Basic Quantities**
  - y: observed data
  - θ: parameters (and other unobserved quantities)
  - x: constants, predictors for conditional (aka "discriminative") models
- **Basic Predictive Quantities**
  - $\tilde{y}$: unknown, potentially observable quantities
  - $\tilde{x}$: constants, predictors for unknown quantities

Naming Conventions

- **Joint**: $p(y, \theta)$
- **Sampling / Likelihood**: $p(y|\theta)$
  - Sampling is function of y with θ fixed (prob function)
  - Likelihood is function of θ with y fixed (*not* prob function)
- **Prior**: $p(\theta)$
- **Posterior**: $p(\theta|y)$
- **Data Marginal (Evidence)**: $p(y)$
- **Posterior Predictive**: $p(\tilde{y}|y)$

Bayes's Rule for Posterior

$$
\begin{aligned}
p(\theta|y) &= \frac{p(y,\theta)}{p(y)} && \text{[def of conditional]} \\
&= \frac{p(y|\theta)\,p(\theta)}{p(y)} && \text{[chain rule]} \\
&= \frac{p(y|\theta)\,p(\theta)}{\int_\Theta p(y,\theta')\,d\theta'} && \text{[law of total prob]} \\
&= \frac{p(y|\theta)\,p(\theta)}{\int_\Theta p(y|\theta')\,p(\theta')\,d\theta'} && \text{[chain rule]}
\end{aligned}
$$

- *Inversion:* Final result depends only on sampling distribution (likelihood) $p(y|\theta)$ and prior $p(\theta)$

Bayes's Rule up to Proportion

- If data y is fixed, then

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

- Posterior proportional to likelihood times prior
- Equivalently, posterior proportional to joint
- The nasty integral for data marginal $p(y)$ goes away

Posterior Predictive Distribution

- Predict new data $\tilde{y}$ based on observed data y
- Marginalize parameters θ out of posterior and likelihood

$$
\begin{aligned}
p(\tilde{y} \mid y) &= \mathbb{E}[p(\tilde{y}|\theta) \mid Y = y] \\
&= \int p(\tilde{y}|\theta)\, p(\theta|y)\, d\theta.
\end{aligned}
$$

- Weights predictions $p(\tilde{y}|\theta)$, by posterior $p(\theta|y)$
- Integral notation shorthand for sums and/or integrals

Event Probabilities

- Events are fundamental probability bearing units which
  - are defined by sets of outcomes
  - which occur or not with some probability
- Events typically defined as conditions on random variables, e.g.,
  - $\theta_b > \theta_g$ for more boy births than girl births
  - $z_k = 1$ for team A beating team B in game k

Event Probabilities, cont.

- $\theta = (\theta_1, \ldots, \theta_N)$ is a sequence of random variables
- c is a condition on θ, so that $c(\theta_1, \ldots, \theta_N) \in \{0, 1\}$
- Suppose $Y = y$ is some observed data
- The probability that the event holds conditioned on the data is given by

$$\begin{aligned}\text{Pr}[c(\theta_1, \ldots, \theta_N) | Y = y] &= \mathbb{E}[c(\theta_1, \ldots, \theta_N) \mid Y] \\ &= \int c(\theta)\, p(\theta|y)\, \mathrm{d}\theta\end{aligned}$$

- Not frequentist, because involves parameter probabilities

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)

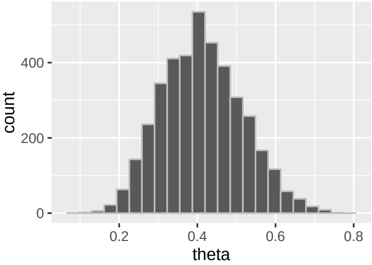
```
> mean(theta_draws);
[1] 0.4128373
```

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

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

Stan Binomial Comparison

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

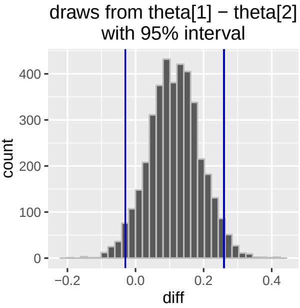
Results

	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
    = normal_rng(x_tilde * 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

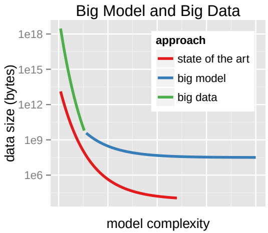
Model Sets Translated to Stan

- 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



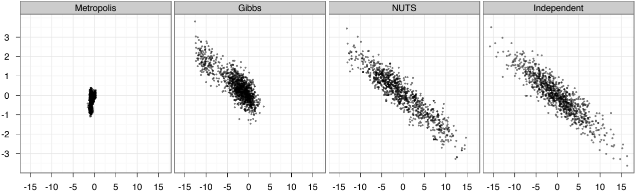
- Types of Scaling: data, parameters, **models**

- 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)

Questions?