The Stan Core Roadmap

Stan Development Team (in order of joining):


34 active devs, ≈ 10 full-time equivalents

Stan 2.18.0 (June 2018) http://mc-stan.org
Part I

Rear-View Mirror
Stan 2.18 Released

- Math, Stan, CmdStan 2.18 currently
- RStan and PyStan 2.18 out soon
- Stan 2.19 to follow soon after
Multi-core Processing has Landed!

- Not just parallel chains

- Distribute log density and gradient calculations over
  - multiple cores on a single machine using **C++11 threading**
  - multiple cores on a single machine or cluster using **MPI**
  - also runs sequentially with memory-locality savings

- Nearly **embarassingly parallel**
  - In representative experiments, 100 cores ran 80+ times faster than a single core with MPI on a standard cluster
Multi-Process Parallelism

- Implemented with the message passing interface (MPI)
- Runs cross-platform with standards-compliant MPI
  - tested on Linux and Mac OS X
  - based on a generalized higher-order map function, e.g.,
    \[ \text{map}(f)(x_1, \ldots, x_N) = (f(x_1), \ldots, f(x_N)) \]
  - applies \( f \) to each element of a sequence \((x_1, \ldots, x_N)\)
- pushes data arguments to processors once
- pushes arguments to processors per eval (map)
- synchronizes reassembly in root expression graph (reduce)
Map Function

- The mapped function has signature

  \[
  \text{vector } f(\text{vector, vector, data real[]}, \text{data int[]})
  \]

- The higher-order map function has signature

  \[
  \text{vector map_rect(F } f, \text{ vector phi, vector[] theta, data real[], } \text{x_r, data int[], } \text{x_i)}
  \]

- The result is computed as follows

  \[
  \text{map_rect(f, phi, theta, x_r, x_i)}
  = \text{append_col}(f(\text{phi, theta}[1], \text{x_r}[1], \text{x_i}[1]),
  \ldots,\n  f(\text{phi, theta}[N], \text{x_r}[N], \text{x_i}[N]))
  \]
New Built-in Functions

- multivariate normal RNG and Cholesky normal RNG
- many RNGs now vectorized (the rest to come soon)
- thin QR decomposition
- matrix-exponential multiply action plus scaled version
- Adams ODE integrator
- generalize log mixture function beyond two arguments
- standard normal distribution
- vectorized ordered probit and logistic
Manuals to HTML

- Breaking 2.17 manual into three parts:
  - *Stan Reference Manual*: specification of the language and algorithms
  - *Stan Functions Reference Manual*: specification of built-in functions
  - *Stan User’s Guide*: programming techniques and example model

- Reference manual in bookdown for HTML and pdf
  - user’s guide, function manual HTML soon

- Expand *User’s Guide* to reproducible *Stan Book*
Improved Effective Sample Size

- Aki Vehtari has been working on better calibration
- NUTS can produce anti-correlated draws
  - effective sample size may exceed number of iterations!
- pushed to CmdStan, RStan, and PyStan
Foreach Loops

- Loop over elements of container rather than numbers
- Works for any array type, looping over elements
- Also works for vector and matrix types

```cpp
matrix[3, 4] ys[7];
for (matrix y : ys) {
    ... do something with y...
}
```

replaces

```cpp
for (i in 1:7) {
    matrix[3, 4] y = ys[i];
    ... do something with y ...
}
```
Data-qualified Arguments

- Allow data qualifier on function arguments
- Requires argument to be data-only expression
- User-defined functions w. algebraic solver, ODEs, map-reduce, etc.
- For example, to parallelize logistic regression, define

```cpp
class logistic_glm{
    vector beta;
    int dummy;
    data real x_r[N];
    data int y[N];

    real logistic_glm(vector beta, vector dummy, data real x_r, data int y) {
        return bernoulli_logit_lpmf(y | to_matrix(x_r) * beta);
    }
}
```
Bug Fixes and Enhancements

- Lots of little things in the parser
  - better parser error messages
  - fixed compound arithmetic/assignment and ternary operator syntax edge cases
- Allow initialization to continue through constraint violation in transformed parameters
- Exceptions/rejections in generated quantities produce all not-a-number values rather than failure
Math Library Enhancements

- In 2.18 math lib, scheduled for Stan 2.19
- Covariance functions
  - squared exponential
  - dot product
  - periodic
- Definite integrator (one dimensional)
- Add-diagonal and log-inverse-logit-difference functions
- GLM primitives for Bernoulli-logit and Poisson-log
- Vectorized ordered logit and probit
CmdStan Enhancements

• Allow Euclidean metric (inverse mass matrix) specification

• Precompiled header support for faster compilation
  – C++ compilers are getting slower with more optimizations
GPU Support

- OpenCL for double-precision arithmetic & portability
  - may also eventually include a CUDA interface

- Initial rollout in Stan 2.19 for
  - matrix-matrix multiply ($N^3$ data, $N^2$ computation)
  - Cholesky factorization ($N^3$ data, $N^2$ computation)
  - matrix-vector multiply ($N^2$ data, $N^2$ computation)

- Order of magnitude speedup without loss of precision for large problems
  - Gaussian processes, factor models, etc.
GPU Speedup, Cholesky (40+ times)

- Time to solve for $L$ for positive-definite $\Sigma = LL^\top$
- with an affordable GPU and Linux desktop
PDEs, DAEs & Definite Integrals

• Partial differential equation (PDE) solver framework
  – common framework for pluggable solvers
  – problem-specific solvers for PDEs

• Differential algebraic equation (DAE) solver
  – extends the existing algorithmic solver
  – differential implicit functions

• Definite integral solver
  – Density normalization inside language
Tuples (i.e., Product Types)

- Hold sequences of heterogeneous types
- Like typed, unnamed R lists or Python dictionaries

```cpp
#include <tuple>

tuple<matrix, vector> eigen_decompose(matrix x);

matrix z;
tuple<matrix, vector> ed = eigen_decompose(z);

// accessors
matrix z_eigenvecs = ed.1;
vector z_eigenvals = ed.2;

// constructors
tuple<matrix, vector> ed2 = (ed.1, ed.2);
```
Ragged Arrays

- Arrays where
  - all elements are the same shape (e.g., ‘real[,]‘)
  - not all elements are the same size

- Critical for a range of applications

- Declared with array of sizes

```c
int<lower = 0> M; // rows
int<lower = 0> N[M]; // cols for row
real[N] y; // y has M rows; row m has N[m] cols
```
Lambdas and Function Types

- Define anonymous inline functions (may be assigned, passed)
- Define higher-order functions
- Closures capture variables (static, lexical scope)
  - no more data arguments to ODE system functions
- Transpile directly to C++ closures
- Example uses manual function syntax

```c
int n = 3;
(real):real cube = (real x).x^n;  // binds n
real x = 2.5;
real x_cubed = cube(x);  // x_cubed == 15.625
```
Independent Generated Quants

- Stored posterior sample with new generated quantities
  - parameter declarations must match
  - model block is ignored
  - generated quantities may vary

- Provides flexible posterior predictive inference
  - e.g., allows streaming posterior predictions for new items
  - e.g., decouples decision theory from posterior generation
  - e.g., allows exploratory posterior predictive checks

- Already built into C++ core; needs pull from interfaces
Adjoint-Jacobian Product Functor

- Efficient matrix autodiff without fiddling code/memory
  - supports direct matrix derivative code
  - reduces reverse pass to single virtual function call
  - lazy adjoint-Jacobian product avoids storing Jacobian
  - store state during functor operator() call
  - multiply-adjoint-Jacobian may be called multiple times

```cpp
struct my_vector_fun {
  VectorXd operator()(const VectorXd& x) { ... }

  VectorXd multiply_adjoint_jacobian(const VectorXd& fx_adj)
  const { ... }
};
```
Mass Matrix/Step Size Init

- User may provide mass matrix (inverse Euclidean metric)
  - may already provide step size (temporal discretization)

- Allows metric initialization with known parameter scales

- Allows restart after adaptation or with more adaptation
  - requires save of RNG state for exact match

- Already built into C++ core; needs pull from interfaces
Variadic Functions, not Packing

- coded with parameter packs in C++11
- map won’t need to pack shared parameters
  \[
  \text{map\_rect}(f, \ \theta, \ \phi_1, \ldots, \ \phi_N, \ x_1, \ldots, \ x_M)
  = \text{append\_col}(f(\theta[1], \ \phi_1, \ldots, \ \phi_N, \ x_1, \ldots, \ x_M)
                   \ldots,
                   f(\theta[N], \ \phi_1, \ldots, \ \phi_N, \ x_1, \ldots, \ x_M))
  \]
- integrate ODE won’t need to pack parameters or data
  \[
  \text{system: } f(t, \ y, \ \theta_1, \ldots, \ \theta_N, \ x_1, \ldots, \ x_M)
  \]
  \[
  \text{ode\_integrate}(f, \ y_0, \ t_0, \ ts, \ \theta_1, \ldots, \ \theta_N,
                   \ x_1, \ldots, \ x_M)
  \]
Part III

The Longer Road
Faster Compile Times

• **Key is** replacing model template with base class
  - Stan program translated to a specific C++ class
  - algorithms and service functions templated for class
  - math library primarily header only
  - so everything recompiles for each Stan program
  - model base class eliminates most recompilation

• **And precompiling** as much of math library as possible
  - vectorized operations combinatorially prohibitive
Blockless Stan Language

• No required block declarations
  – optional qualification for **backward compatibility**
  – infer block structure for rest
  – allow missing data a la BUGS (continuous only)
  – allow modules with parameters, e.g., non-centered prior
  – retain imperative execution order, functions, etc.

• Inspired by composability in language theory

• Inspired by & partially realized by transpilers
  – **StataStan**: CiBO Technologies, open source
  – **SlicStan**: Maria Gorinova’s M.S. thesis
Blockless Linear Regression

real alpha ~ normal(0, 4); // param
real beta ~ normal(0, 4); // param
int<lower = 0> N; // data (unmodeled)
vector[N] x; // data (unmodeled)
vector[N] mu_y = alpha + beta * x; // trans param
real<lower = 0> sigma_y ~ normal(0, 2); // param
vector[N] y ~ normal(mu_y, sigma_y); // data (modeled)

- Allow model in generative order (parameters to data)
- Variable use moves closer to declarations
Non-Centered Normal Module

- Declare module for non-centered normal prior

- Parameters and transformed parameters in module

```plaintext
module non_ctr_normal(int N, real mu, real sigma) {
    vector[N] alpha_std ~ normal(0, 1); // param
    vector[N] alpha = mu + sigma * alpha_std; // trans param
}
real mu_alpha ~ normal(0, 5); // param
real<lower = 0> sigma_alpha ~ normal(0, 5); // param
int <lower = 1> K; // data
module ncn = non_ctr_normal(K, mu_alpha, sigma_alpha);
int<lower = 0> N; // data
int<lower = 1, upper = K> ii[N]; // data
vector[n] y ~ normal(ncn.alpha[ii] + beta * x, tau); // data
```
Protocol Buffer I/O

- Protobuf is a **standardized**, widely supported
- Efficient **binary representation**
- Originally developed by Google
- **Schema driven**
  - efficient binary output formats without extraneous metadata
- Will replace the current hacked R dump format for input
- *Probably* replacing numerical outputs
- Auto-convertible to/from human-readable **JSON**
Logging Standards

• Add logger for console-type output

• Allow finer control of verbosity through interfaces
  – DEBUG (?): information to help developers
  – INFO: regular output reminders
  – WARN: warnings
  – ERROR: errors
  – FATAL: fatal errors

• Configurable static logger eases algorithm dev
Program Transformations

- For **optimization**
  - reducing common subexpressions
  - eliminate dead code
  - transform block location
  - auto-vectorize

- For **arithmetic stability**
  - log-scale and special functions

- Transform intermediate **abstract syntax tree**
  - refactor from C++ variant types to S-expressions
Questions?
Suggestions?