Stan:
Probabilistic Modeling & Bayesian Inference

Development Team

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, Matthijs Vàkàr, Andrew Johnson, Dan Simpson

Stan 2.17 (November 2017) `http://mc-stan.org`

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 defining 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

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 evaluate 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. Vectors & 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 (memory locality matters)
- 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

Assignment Operators

Op.	*Description*
`=`	assignment
`+=`	compound add and assign
`-=`	compound subtract and assign
`*=`	compound multiply and assign
`/=`	compound divide and assign
`.*=`	compound elementwise multiply and assign
`./=`	compound elementwise divide and assign

- these work with all relevant matrix types
  - e.g., `matrix *= matrix;`

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,

Atomic Statements

- **Sampling**: `y ~ normal(mu,sigma)` (**increments log probability**)
- **Increment log density**: target += lp;
- **Assignment**: `y_hat = x * beta`;

Block of Statements

- **Block**: `{ ...; ...; ...; }` (allows initial local variables)

Control Statements

- **For loop**: `for (n in 1:N) ...`
- **While loop**: `while (cond) ...`
- **Conditional**: `if (cond) ...; else if (cond) ...; else ...;`
- **Break**: `break`
- **Continue**: `continue`

Side-Effect Statements

- **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, lambda));`
- * 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 = sum_y + 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));
  }

}
```

Special User-Defined Functions

- When declared with appropriate naming, user-defined functions may
  - be used in sampling statements: `real` return and suffix `_lpdf` or `_lpmf`
  - use RNGs: suffix `_rng`
  - use target accumulator: suffix `_lp`

User-Defined PDFs and CDFs

- May be used with sampling and PDF/PMF notation
    - **density**: suffix `_lpdf` if variate (first arg) is real
    - **mass**: Suffix `_lpmf` if variate (first arg) is integer

```
functions {
  real centered_normal_lpdf(real y, real sigma) {
    return -0.5 * (y / sigma)^2;
  }

model {
  y ~ centered_normal(2.5);                 // sampling

  target += centered_normal_lpdf(y | 2.5);  // pdf notation
```

Target Incrementing Functions

- May access target or use sampling statements
- Only usable in model block
  - must end in _lp

```
functions {
  vector non_center_lp(vector beta_std, real mu, real sigma) {
    beta_std ~ normal(0, 1);
    return mu + sigma * beta_raw;
  }
parameters {
  vector[K] beta_std;
model {
  vector[K] beta = non_center_lp(beta_std);
  y ~ normal(x * beta, sigma);
```

RNG Functions

- Only usable in generated quantities block
  - must end in _rng

```
functions {
  real centered_normal_rng(real sigma) {
    return normal_rng(0, sigma);
  }

generated quantities {
  real alpha = centered_normal_rng(2.7);
```

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

Differential Equation Solver (cont.)

- 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

Built-in Diff Eq Solvers

- **Non-stiff solver**: Runge-Kutta 4th/5th order (RK45)
- **Stiff solver**: backward-differentiation formula (BDF)
  - slower
  - more robust for derivatives of different scales or high curvature
- specified by suffix `_bdf` or `_rk45`

Diff Eq Derivatives

- User defines system $\frac{\partial}{\partial t} y$
- Need derivatives of solution y w.r.t. parameters θ
- Couple derivatives of system w.r.t. parameters

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

- Calculate coupled system via nested autodiff of second term

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

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 differentiation
- 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)

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

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

MAP / 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 + autodiff
- 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)

Under Stan’s Hood

Euclidean Hamiltonian Monte Carlo

- **Phase space**: q position (parameters); p momentum
- **Posterior density**: $\pi(q)$
- **Mass matrix**: M
- **Potential energy**: $V(q) = -\log \pi(q)$
- **Kinetic energy**: $T(p) = \frac{1}{2} p^\top M^{-1} p$
- **Hamiltonian**: $H(p, q) = V(q) + T(p)$
- **Diff eqs**:

$$\frac{dq}{dt} = +\frac{\partial H}{\partial p} \qquad\qquad \frac{dp}{dt} = -\frac{\partial H}{\partial q}$$

Leapfrog Integrator Steps

- Solves Hamilton's equations by **simulating dynamics**
  (symplectic [volume preserving]; ϵ^3 error per step, ϵ^2 total error)
- Given: **step size** ϵ, **mass matrix** M, **parameters** q
- **Initialize kinetic** energy, $p \sim \text{Normal}(0, \mathbf{I})$
- **Repeat** for L leapfrog steps:

$$p \leftarrow p - \frac{\epsilon}{2} \frac{\partial V(q)}{\partial q} \qquad \text{[half step in momentum]}$$

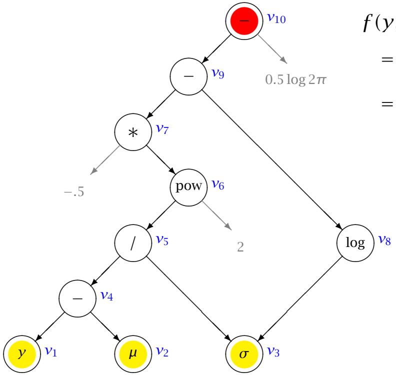
$$q \leftarrow q + \epsilon M^{-1} p \qquad \text{[full step in position]}$$

$$p \leftarrow p - \frac{\epsilon}{2} \frac{\partial V(q)}{\partial q} \qquad \text{[half step in momentum]}$$

Reverse-Mode Auto Diff

- Eval gradient in (usually small) multiple of function eval time
  - independent of dimensionality
  - time proportional to number of expressions evaluated
- Result accurate to machine precision (cf. finite diffs)
- Function evaluation builds up **expression tree**
- Dynamic program propagates **chain rule** in reverse pass
- Reverse mode computes ∇g in one pass for a function $f : \mathbb{R}^N \rightarrow \mathbb{R}$

Autodiff Expression Graph



$$
\begin{aligned}
f(y,\mu,\sigma) \\
&= \log\left(\mathsf{Normal}(y|\mu,\sigma)\right) \\
&= -\tfrac{1}{2}\left(\tfrac{y-\mu}{\sigma}\right)^2 - \log\sigma - \tfrac{1}{2}\log(2\pi)
\end{aligned}
$$

$$
\begin{aligned}
\tfrac{\partial}{\partial y} f(y,\mu,\sigma) \\
&= -(y-\mu)\sigma^{-2}
\end{aligned}
$$

$$
\begin{aligned}
\tfrac{\partial}{\partial \mu} f(y,\mu,\sigma) \\
&= (y-\mu)\sigma^{-2}
\end{aligned}
$$

$$
\begin{aligned}
\tfrac{\partial}{\partial \sigma} f(y,\mu,\sigma) \\
&= (y-\mu)^2\sigma^{-3} - \sigma^{-1}
\end{aligned}
$$

Autodiff Partials

var	*value*	*partials*	
v_1	y		
v_2	μ		
v_3	σ		
v_4	$v_1 - v_2$	$\partial v_4 / \partial v_1 = 1$	$\partial v_4 / \partial v_2 = -1$
v_5	v_4 / v_3	$\partial v_5 / \partial v_4 = 1 / v_3$	$\partial v_5 / \partial v_3 = -v_4 v_3^{-2}$
v_6	$(v_5)^2$	$\partial v_6 / \partial v_5 = 2 v_5$	
v_7	$(-0.5) v_6$	$\partial v_7 / \partial v_6 = -0.5$	
v_8	$\log v_3$	$\partial v_8 / \partial v_3 = 1 / v_3$	
v_9	$v_7 - v_8$	$\partial v_9 / \partial v_7 = 1$	$\partial v_9 / \partial v_8 = -1$
v_{10}	$v_9 - (0.5 \log 2\pi)$	$\partial v_{10} / \partial v_9 = 1$	

Autodiff: Reverse Pass

var	*operation*	*adjoint*	*result*
$a_{1:9}$	$=$	0	$a_{1:9} = 0$
a_{10}	$=$	1	$a_{10} = 1$
a_9	$+=$	$a_{10} \times (1)$	$a_9 = 1$
a_7	$+=$	$a_9 \times (1)$	$a_7 = 1$
a_8	$+=$	$a_9 \times (-1)$	$a_8 = -1$
a_3	$+=$	$a_8 \times (1/v_3)$	$a_3 = -1/v_3$
a_6	$+=$	$a_7 \times (-0.5)$	$a_6 = -0.5$
a_5	$+=$	$a_6 \times (2v_5)$	$a_5 = -v_5$
a_4	$+=$	$a_5 \times (1/v_3)$	$a_4 = -v_5/v_3$
a_3	$+=$	$a_5 \times (-v_4 v_3^{-2})$	$a_3 = -1/v_3 + v_5 v_4 v_3^{-2}$
a_1	$+=$	$a_4 \times (1)$	$a_1 = -v_5/v_3$
a_2	$+=$	$a_4 \times (-1)$	$a_2 = v_5/v_3$

Stan’s Reverse-Mode

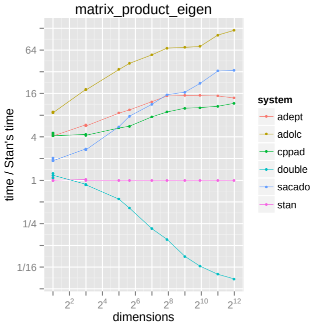
- Easily extensible **object-oriented** design
- **Code nodes** in expression graph for primitive functions
  - requires **partial derivatives**
  - built-in flexible abstract base classes
  - **lazy evaluation** of chain rule saves memory
- Autodiff through templated C++ functions
  - templating on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

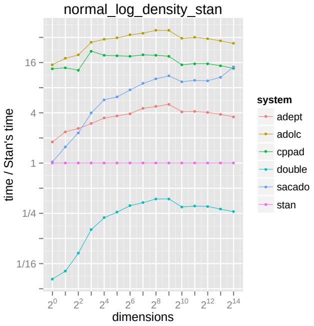
Stan’s Reverse-Mode (cont.)

- 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

Stan’s Autodiff vs. Alternatives

- Stan is **fastest** (and uses least memory)
  - among open-source C++ alternatives





Forward-Mode Auto Diff

- Evaluates expression graph forward from one independent variable to any number of dependent variables
- Function evaluation propagates **chain rule** forward
- In one pass, computes $\frac{\partial}{\partial x} f(x)$ for a function $f : \mathbb{R} \rightarrow \mathbb{R}^N$
  - derivative of N outputs with respect to a single input

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

- Required for Riemannian HMC
- Gradients of Hessians (tensor of third-order partials)

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

  - N^2 forward passes: gradient of derivative of derivative

Third-order Derivatives (cont.)

- Gradient of trace of Hessian times matrix
  - $\nabla \mathrm{tr}(H\,M)$, or
  - needed for Riemannian Hamiltonian Monte Carlo
  - computable in quadratic time for fixed M

Jacobians

- Assume function $f : \mathbb{R}^N \rightarrow \mathbb{R}^M$
- 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
  - 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: $\mathcal{O}(N^2)$
    (for SoftAbs RHMC)

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

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

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