Stan: Under the Bonnet

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, Mitzi Morris, Rob Trangucci, Rob Goedman, Jonah Sol Gabry, Robert L. Grant, Brian Lau, Krzysztof Sakrejda, Aki Vehtari, Rayleigh Lei, Sebastian Weber, Charles Margossian, Vincent Picaud, Imad Ali, Sean Talts, Ben Bales, Ari Hartikainen, Matthijs Vákár, Andrew Johnson, Dan Simpson, Yi Zhang, Paul Bürkner, Steve Bronder, Rok Cesnovar, Erik Strumbelj, Edward Roualdes

34 active devs, ≈ 10 *full-time equivalents*

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

Linear Regression with Prediction

```
data {
  int<lower = 0> K;          int<lower = 0> N;
  matrix[N, K] x;            vector[N] y;
  int<lower = 0> N_p;        matrix[N_tilde, K] x_p;
}
parameters {
  vector[K] beta;            real<lower = 0> sigma;
}
model {
  y ~ normal(x * beta, sigma);
}
generated quantities {
  vector[N_p] y_p = normal_rng(x_p * beta, sigma);
}
```

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, $\log \left| \frac{d}{dy} \exp(y) \right| = y$
- 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

"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

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
  - **discrete sample** along path prop to density

(Hoffman and Gelman 2011, 2014)

Adaptation During Warmup

- (I) initial fast interval to find typical set
  (adapt step size, default 75 iterations)
- (II) expanding memoryless windows to estimate metric
  (adapt step size & metric, initial 25 iterations)
- (III) final fast interval for final step size
  (adapt step size, default 50 iterations)

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 approximate or exact leave-one-out 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)
- Standard errors **on constrained scale**
  (sample unconstrained approximation and inverse transform)
- From Bayesian perspective, Laplace approximation to posterior

“Black Box” Variational Inference

- **Black box** so can fit any Stan model
- Multivariate **normal approx to unconstrained** posterior
  - covariance: diagonal (aka mean-field) or full rank
  - like Laplace approx, but around posterior mean, not mode
- **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**:

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

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:

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

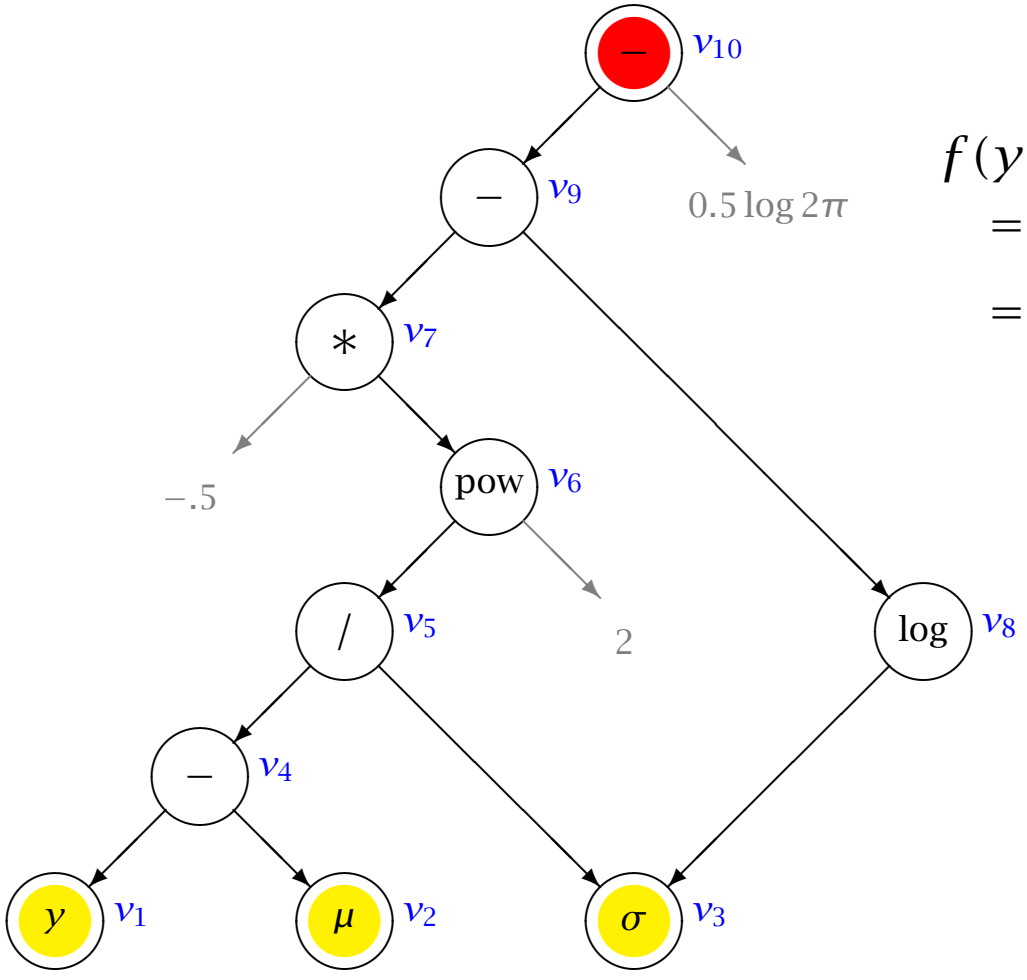
$$q \leftarrow q + \epsilon M^{-1} p \qquad \text{[full step in position]}$$

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

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



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

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

Autodiff Partials

var	*value*	*partials*
v_1	y	
v_2	μ	
v_3	σ	
v_4	$v_1 - v_2$	$\partial v_4 / \partial v_1 = 1$ $\quad$ $\partial v_4 / \partial v_2 = -1$
v_5	v_4 / v_3	$\partial v_5 / \partial v_4 = 1 / v_3$ $\quad$ $\partial v_5 / \partial v_3 = -v_4 v_3^{-2}$
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	$v_7 - v_8$	$\partial v_9 / \partial v_7 = 1$ $\quad$ $\partial v_9 / \partial v_8 = -1$
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

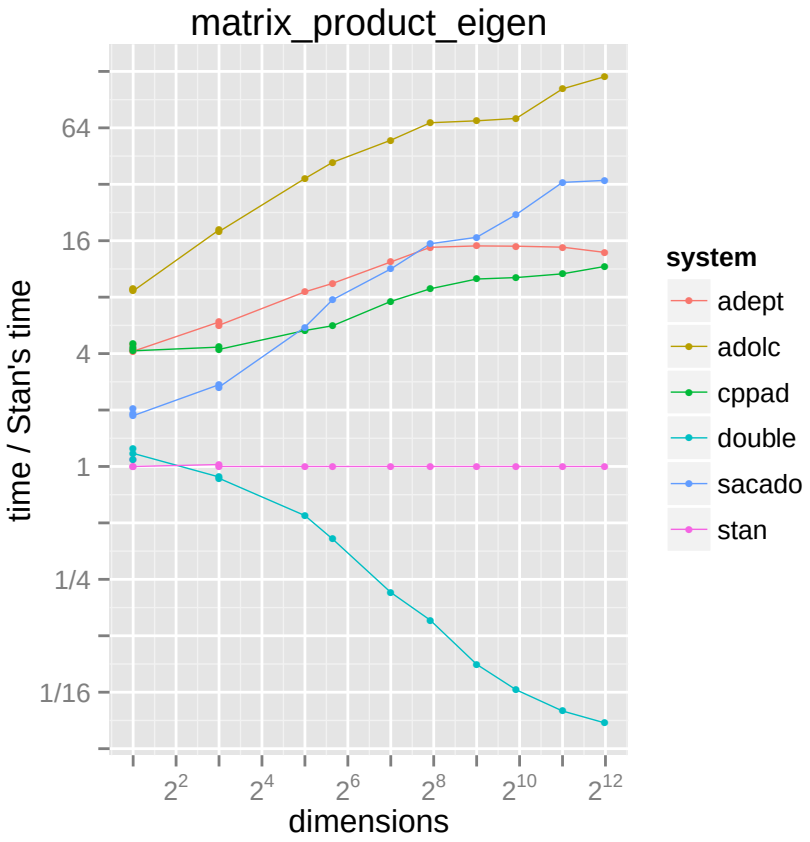
- Easily extensible **object-oriented** design
- **Code nodes** in expression graph for primitive functions
  - requires **partial derivatives**
  - built-in flexible abstract base classes
  - **lazy evaluation** of chain rule saves memory
- Autodiff through templated C++ functions
  - templating each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

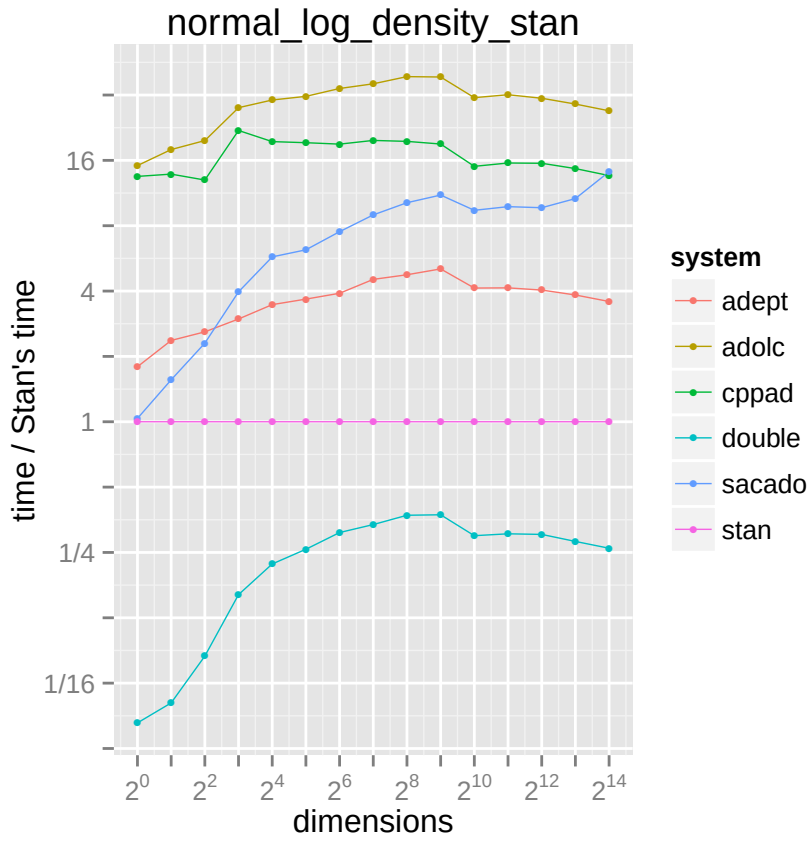
Stan's Reverse-Mode (cont.)

- 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**
- Adjoint-vector product formulation for **multivariates**
  - avoids N^2 memory cost of storing Jacobian
  - minimizes autodiff nodes and virtual function calls

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 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)

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

- Based on CVODES implementation of BDF (Sundials)
- CVODES builds-in efficient structure for sensitivity
- More nested autodiff required for system Jacobian
  - algebraic reductions save a lot of work

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
- Moving to **OCaml**—nearly complete
  - much cleaner and easier to manage than the C++
  - optimize by tranforming intermediate representations
- **Next**: tuples, ragged arrays, lambdas (closures)

Coding Probability Functions

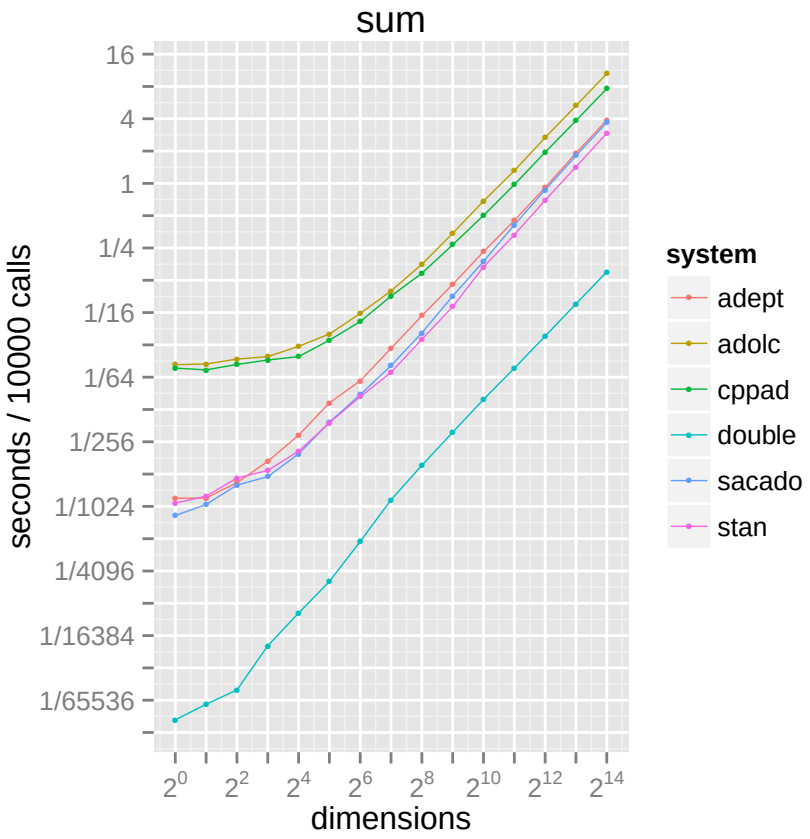
- **Vectorized** to allow scalar or container arguments
  (containers all same shape; scalars broadcast as necessary)
- Avoid **repeated computations**, e.g. $\log \sigma$ in

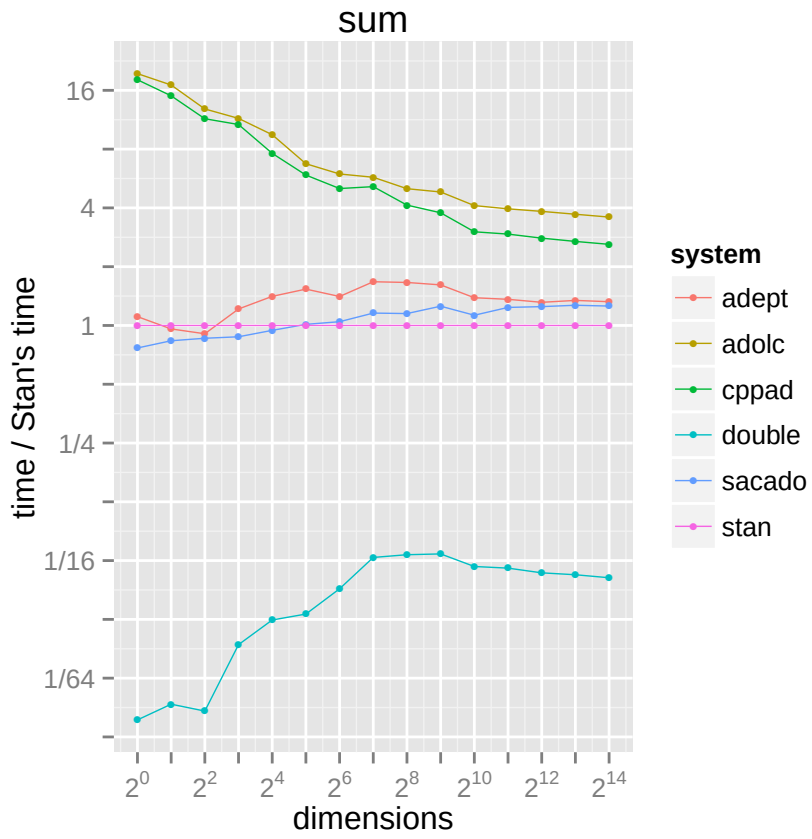
$$\begin{aligned} \log \text{Normal}(y|\mu,\sigma) &= \sum_{n=1}^{N} \log \text{Normal}(y_n|\mu,\sigma) \\ &= \sum_{n=1}^{N} -\log\sqrt{2\pi} \; - \log\sigma \; - \frac{y_n - \mu}{2\sigma^2} \end{aligned}$$

- 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 Autodiff vs. Alternatives

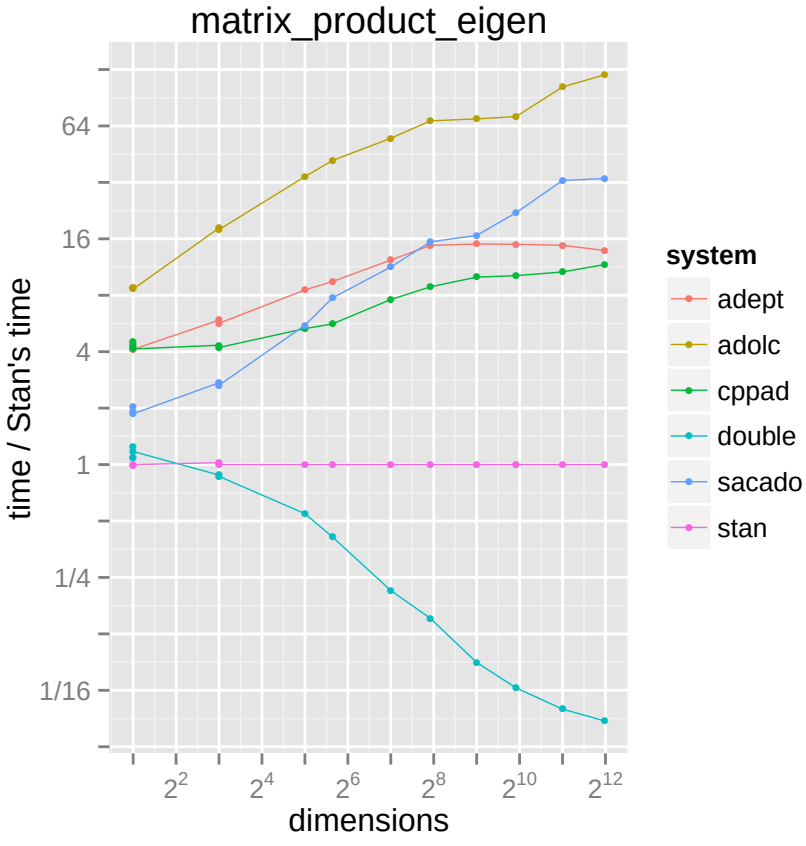
- Stan is **fastest** and uses least memory
  - among open-source C++ alternatives we managed to install

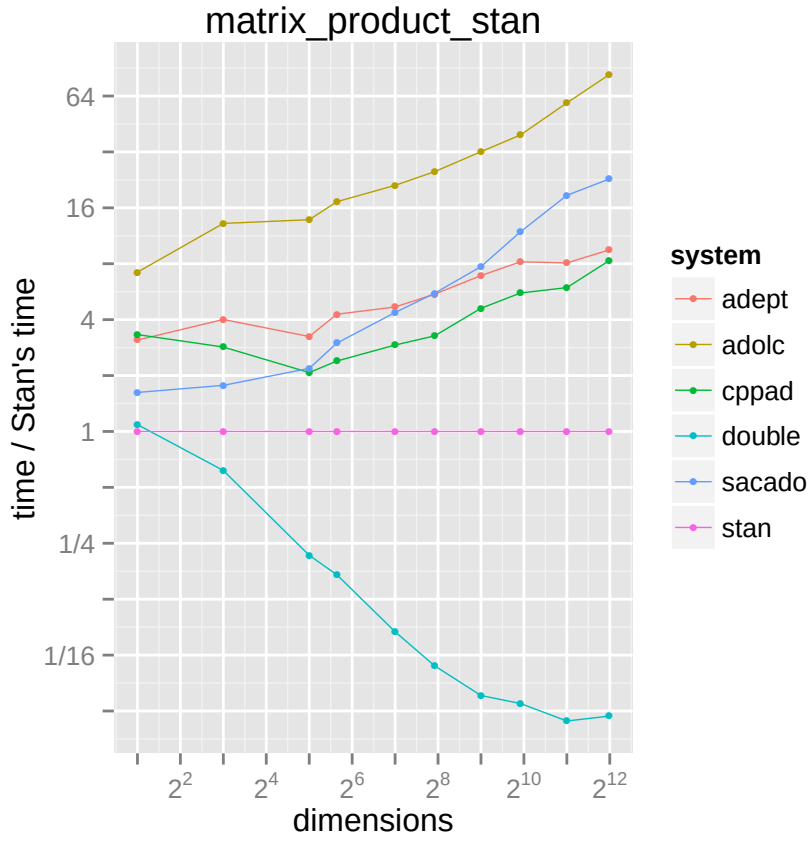




Stan's Matrix Calculations

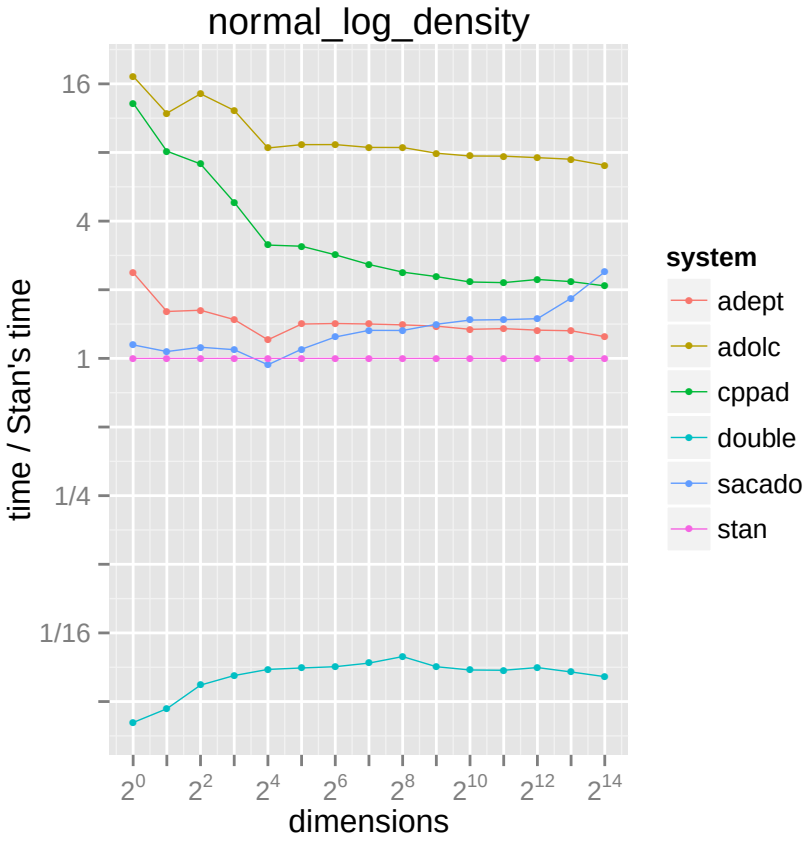
- Faster in Eigen, but takes more memory
- Best of both worlds coming soon

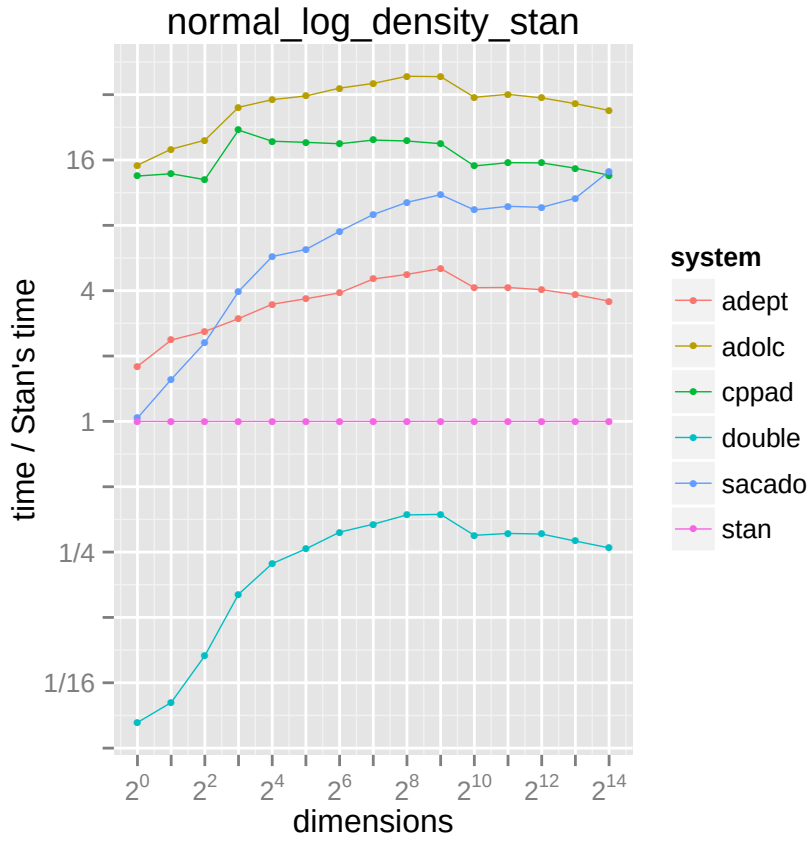




Stan's Density Calculations

- Vectorization a huge win





Hard Models, Big Data

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

Laplace Approximation

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

$$\theta^* = \arg\max_\theta \, p(\theta|y)$$

- 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 θ^*
- 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

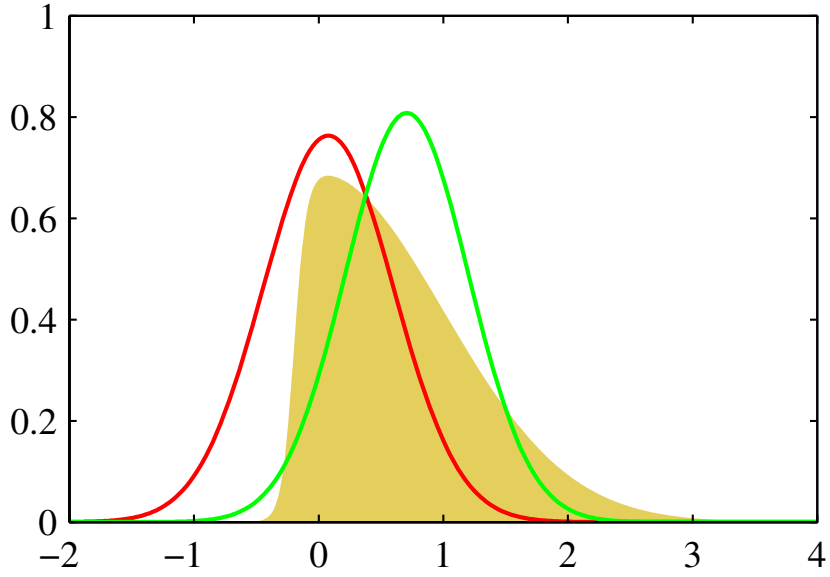
- **Black box** so can fit any Stan model
- Multivariate **normal approx to unconstrained** posterior
  - covariance: diagonal mean-field or full rank
  - not Laplace approx — around posterior mean, not mode
  - transformed back to constrained space (built-in Jacobians)
- Stochastic **gradient-descent**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

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

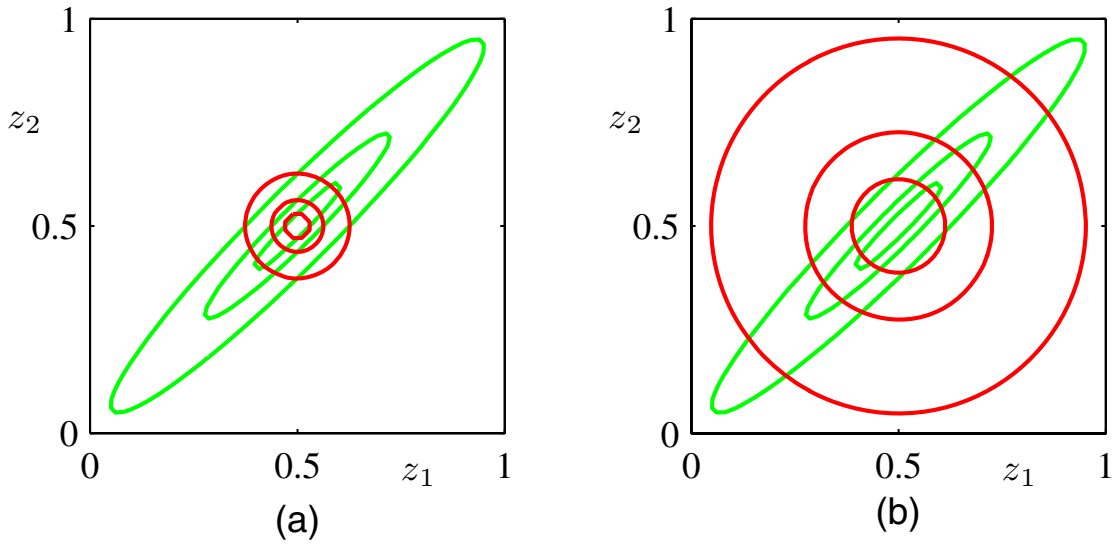
VB vs. Laplace



- *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 **underestimates 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

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

  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

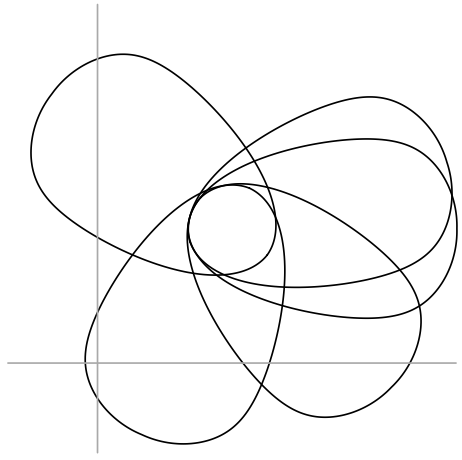
Near Future: Stochastic VB

- 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 minimize KL-divergence
- Prototype code complete

“Black Box” EP

- Fast, approximate inference (like VB)
  - VB and EP minimize divergence in opposite directions
  - especially useful for Gaussian processes
- Asynchronous, data-parallel **expectation propagation** (EP)
- Cavity distributions control subsample variance
- Prototype stage
- collaborating with Seth Flaxman, Aki Vehtari, Pasi Jylänki, John Cunningham, Nicholas Chopin, Christian Robert

The Cavity Distribution



- Two parameters, with data split into $y_1, \ldots, y_5$
- Contours of likelihood $p(y_k|\theta)$ for $k \in 1{:}5$
- $g_{-k}(\theta)$ is **cavity distribution** (current approx. without y_k)
- Separately computing for y_k reqs each partition to cover its area
- Combining likelihood with cavity **focuses on overlap**

Challenges

Discrete Parameters

- e.g., simple mixture models, survival models, HMMs, discrete measurement error models, missing data
- **Marginalize out** discrete parameters
- Efficient sampling due to **Rao-Blackwellization**
- Inference straightforward with expectations

- Too **difficult** for many of our users
  (exploring encapsulation options)

Models with Missing Data

- In principle, missing data just **additional parameters**
- In practice, how to declare?
  - **observed** data as data variables
  - **missing** data as parameters
  - combine into single vector

    (in transformed parameters or local in model)

Position-Dependent Curvature

- Mass matrix does **global** adaptation for
  - parameter scale (diagonal) and rotation (dense)
- Dense mass matrices hard to estimate ($\mathcal{O}(N^2)$ estimands)
- **Problem**: Position-dependent curvature
  - Example: banana-shaped densities
    - arise when parameter is product of other parameters
  - Example: hierarchical models
    - hierarchical variance controls lower-level parameters
- Mitigate by reducing stepsize
  - initial (`stepsize`) and target acceptance (`adapt_delta`)

The End