Section 4.
How Stan Works

Bob Carpenter

Columbia University

Part I
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 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 draws in sample 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

Penalized MLE

- Posterior **mode finding** via L-BFGS optimization

  (uses model gradient, efficiently approximates Hessian)

- **Disables Jacobians** for parameter inverse transforms

- **Standard errors** on unconstrained scale

  (estimated using curvature of penalized log likelihood function

- Models, data, initialization as in MCMC

- **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 + autdiff
- Returns **approximate posterior** mean / covariance
- Returns **sample** transformed to constrained space

Posterior Analysis: Estimates

- For each parameter (and `lp__`)
  - Posterior mean
  - Posterior standard deviation
  - Posterior MCMC error esimate: sd$/N_{\text{eff}}$
  - Posterior quantiles
  - Number of effective samples
  - $\hat{R}$ convergence statistic

- . . . and much much more in ShinyStan

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** (soon: Hessian)
  - data I/O and parameter initialization
  - model provides variable names and dimensionalities
  - transforms to and from constrained representation (with or without Jacobian)

Part II

How Stan Works

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)

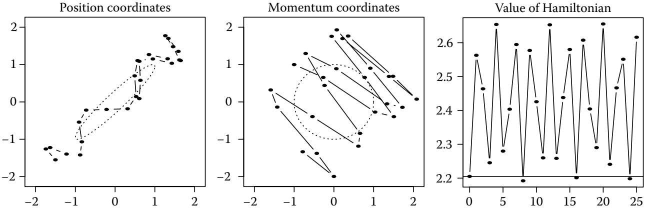
Optimize: L-BFGS

- Initialize unconstrained parameters and Hessian
  - Random values on unconstrained scale uniform in $(-2, 2)$
    - or user specified on constrained scale, transformed
  - Hessian approximation initialized to unit matrix
- While not converged
  - Move unconstrained parameters toward optimum based on Hessian approximation and step size (Newton step)
  - If diverged (arithmetic, support), reduce step size, continue
  - else if converged (parameter change, log density change, gradient value), return value
  - else update Hessian approx. based on calculated gradient

Sample: Hamiltonian Flow

- Generate random **kinetic energy**
  - random Normal$(0, 1)$ in each parameter
- Use negative log posterior as **potential energy**
- Hamiltonian is kinetic plus potential energy
- **Leapfrog Integration**: for *fixed* stepsize (time discretization), number of steps (total time), and mass matrix,
  - update momentum half-step based on potential (gradient)
  - update position full step based on momentum
  - update momentum half-step based on potential
- Numerical solution of Hamilton's first-order version of Newton's second-order diff-eqs of motion (force $=$ mass $\times$ acceleration)

Sample: Leapfrog Example



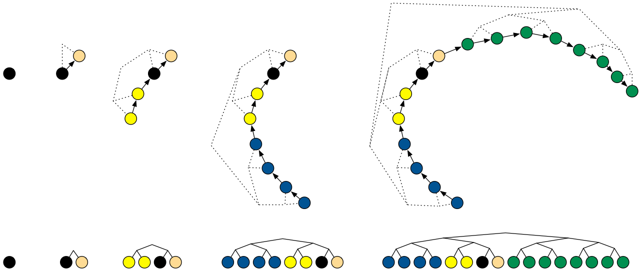
- Trajectory of 25 leapfrog steps for correlated 2D normal (ellipses at 1 sd from mean), stepsize of 0.25, initial state of $(-1, 1)$, and initial momentum of $(-1.5, -1.55)$.

Radford Neal (2013) MCMC using Hamiltonian Dynamics. In *Handbook of MCMC*. (free online at http://www.mcmchandbook.net/index.html)

Sample: No-U-Turn Sampler (NUTS)

- Adapts Hamiltonian simulation time
  - goal to maximize mixing, maintaining detailed balance
  - too short devolves to random walk
  - too long does extra work (i.e., orbits)
- For exponentially increasing number of steps up to max
  - Randomly choose to extend forward or backward in time
  - Move forward or backward in time number of steps
    - stop if any subtree (size 2, 4, 8, ...) makes U-turn
    - remove all current steps if subtree U-turns (not ends)
- Randomly select param with density above slice (or reject)

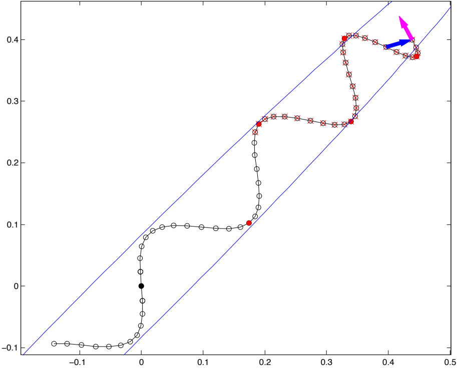
Sample: NUTS Binary Tree



- Example of repeated doubling building binary tree forward and backward in time until U-turn.

  Hoffman and Gelman. 2014. The No-U-Turn Sampler. *JMLR*. (free online at http://jmlr.org/papers/v15/hoffman14a.html)

Sample: NUTS U-Turn



- Example of trajectory from one iteration of NUTS.
- Blue ellipse is contour of 2D normal.
- Black circles are leapfrog steps.
- Solid red circles excluded below slice
- U-turn made with blue and magenta arrows
- Red crossed circles excluded for detailed balance

Sample: HMC/NUTS Warmup

- Estimate stepsize
  - too small requires too many leapfrog steps
  - too large induces numerical inaccuracy
  - need to balance
- Estimate mass matrix
  - Diagonal accounts for parameter scales
  - Dense optionally accounts for rotation

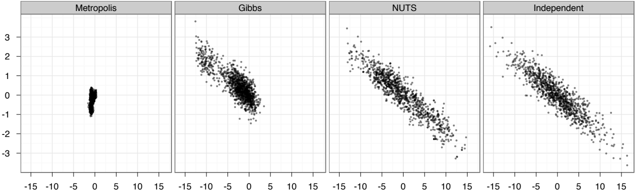
Sample: Warmup (cont.)

- Initialize unconstrained parameters as for optimization
- For exponentially increasing block sizes
  - for each iteration in block
    - generate random kinetic energy
    - simulate Hamiltonian flow (HMC fixed time, NUTS adapts)
    - choose next state (Metroplis for HMC, slice for NUTS)
  - update regularized point estimate of mass matrix
    - use parameter draws from current block
    - shrink diagonal toward unit; dense toward diagonal
  - tune stepsize (line search) for target acceptance rate

Sample: HMC/NUTS Sampling

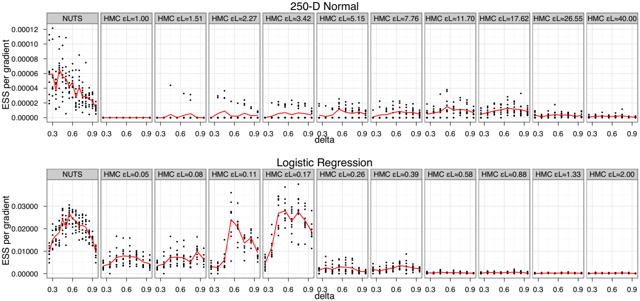
- Fix stepsize and and mass matrix
- For sampling iterations
  - generate random kinetic energy
  - simulate Hamiltonian flow
  - apply Metropolis accept/reject (HMC) or slice (NUTS)

NUTS vs. Gibbs and Metropolis



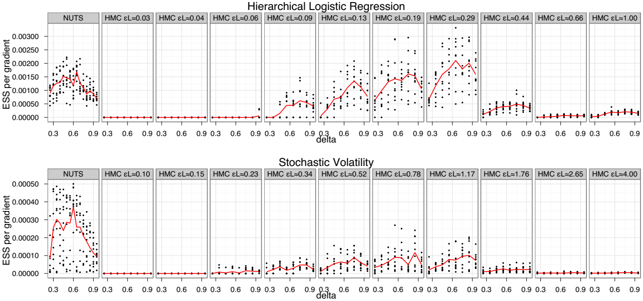
- Two dimensions of highly correlated 250-dim normal
- **1,000,000 draws** from Metropolis and Gibbs (thin to 1000)
- **1000 draws** from NUTS; 1000 independent draws

NUTS vs. Basic HMC



- 250-D normal and logistic regression models
- Vertical axis is effective sample size per sample (bigger better)
- Left) NUTS; Right) HMC with increasing $t = \epsilon L$

NUTS vs. Basic HMC II



- Hierarchical logistic regression and stochastic volatility
- Simulation time t is $\epsilon\ L$, step size (ϵ) times number of steps (L)
- NUTS can beat optimally tuned HMC (latter very expensive)

Part III

Under Stan’s Hood

Euclidean Hamiltonian

- **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:

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

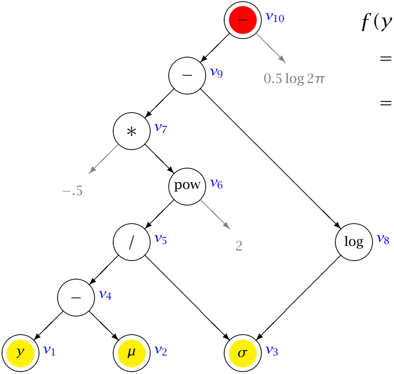
$$q \leftarrow q + \epsilon M^{-1} p \qquad \text{[full step in position]}$$

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

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



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

$$\frac{\partial}{\partial y} f(y, \mu, \sigma) = -(y-\mu)\sigma^{-2}$$

$$\frac{\partial}{\partial \mu} f(y, \mu, \sigma) = (y-\mu)\sigma^{-2}$$

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

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	$(v_5)^2$	$\partial v_6/\partial v_5 = 2v_5$	
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

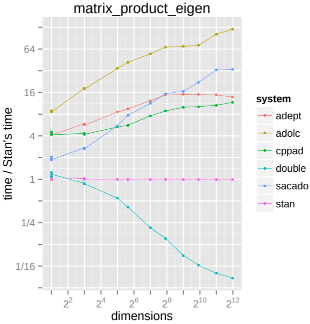
- Easily extensible **object-oriented** design
- **Code nodes** in expression graph for primitive functions
  - requires **partial derivatives**
  - built-in flexible abstract base classes
  - **lazy evaluation** of chain rule saves memory
- Autodiff through templated C++ functions
  - templating on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

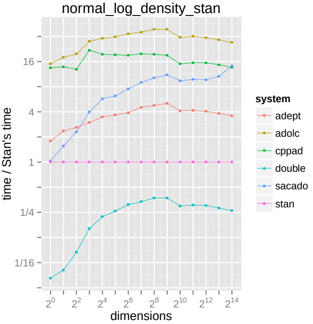
Stan’s Reverse-Mode (cont.)

- 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

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

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

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

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

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

- 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

The End (Section 4)