The Stan Math Library: Reverse-Mode Automatic Differentiation in C++

Bob Carpenter
Columbia University

Matthew D. Hoffman
Adobe Research

Marcus Brubaker
University of Toronto, Scarborough

Daniel Lee
Columbia University

Peter Li
Columbia University

Michael Betancourt
University of Warwick

September 25, 2015

Abstract

As computational challenges in optimization and statistical inference grow ever harder, algorithms that utilize derivatives are becoming increasingly more important. The implementation of the derivatives that make these algorithms so powerful, however, is a substantial user burden and the practicality of these algorithms depends critically on tools like automatic differentiation that remove the implementation burden entirely. The Stan Math Library is a C++, reverse-mode automatic differentiation library designed to be usable, extensive and extensible, efficient, scalable, stable, portable, and redistributable in order to facilitate the construction and utilization of such algorithms.

Usability is achieved through a simple direct interface and a cleanly abstracted functional interface. The extensive built-in library includes functions for matrix operations, linear algebra, differential equation solving, and most common probability functions. Extensibility derives from a straightforward object-oriented framework for expressions, allowing users to easily create custom functions. Efficiency is achieved through a combination of custom memory management, subexpression caching, traits-based metaprogramming, and expression templates. Partial derivatives for compound functions are evaluated lazily for improved scalability. Stability is achieved by taking care with arithmetic precision in algebraic expressions and providing stable, compound functions where possible. For portability, the library is standards-compliant

C++ (03) and has been tested for all major compilers for Windows, Mac OS X, and Linux. It is distributed under the new BSD license.

This paper provides an overview of the Stan Math Library's application programming interface (API), examples of its use, and a thorough explanation of how it is implemented. It also demonstrates the efficiency and scalability of the Stan Math Library by comparing its speed and memory usage of gradient calculations to that of several popular open-source C++ automatic differentiation systems (Adept, Adol-C, CppAD, and Sacado), with results varying dramatically according to the type of expression being differentiated.

1. Reverse-Mode Automatic Differentiation

Many contemporary algorithms require the evaluation of a derivative of a given differentiable function, f, at a given input value, $(x_1, \ldots, x_N)$, for example a gradient,

$$\left(\frac{\partial f}{\partial x_1}(x_1, \ldots, x_N), \cdots, \frac{\partial f}{\partial x_N}(x_1, \ldots, x_N) \right),$$

or a directional derivative,[1]

$$\vec{v}(f)(x_1, \ldots, x_N) = \sum_{n=1}^{N} v_n \frac{\partial f}{\partial x_n}(x_1, \ldots, x_N).$$

Automatic differentiation computes these values automatically, using only a representation of f as a computer program. For example, automatic differentiation can take a simple C++ expression such as `x * y / 2` with inputs `x` $= 6$ and `y` $= 4$ and produce both the output value, 12, and the gradient, $(2, 3)$.

Automatic differentiation is implemented in practice by transforming the subexpressions in the given computer program into nodes of an *expression graph* (see Figure 1, below, for an example), and then propagating chain rule evaluations along these nodes (Griewank and Walther, 2008; Giles, 2008). In *forward-mode automatic differentiation*, each node k in the graph contains both a value x_k and a *tangent*, t_k, which represents the directional derivative of x_k with respect to the input variables. The tangent values for the input values are initialized with values $\vec{v}$, because that represents the appropriate directional derivative of each input variable. The complete set of tangent values is calculated by propagating tangents forward from the inputs to the outputs with the rule

$$t_i = \sum_{j \in \text{children}[i]} \frac{\partial x_i}{\partial x_j} t_j.$$

[1] A special case of a directional derivative computes derivatives with respect to a single variable by setting $\vec{v}$ to a vector with a value of 1 for the single distinguished variable and 0 for all other variables.

In one pass over the expression graph, forward-mode computes a directional derivative of any number of output (dependent) variables with respect to a vector of input (independent) variables and direction $\vec{v}$; the special case of a derivative with respect to a single independent variable computes a column of the Jacobian of a multivariate function. The proof follows a simple inductive argument starting from the inputs and working forward to the output(s).

In *reverse-mode automatic differentiation*, each node k in the expression graph contains a value x_k and an *adjoint* a_k, representing the derivative of a single output node with respect to x_k. The distinguished output node's adjoint is initialized to 1, because its derivative with respect to itself is 1. The full set of adjoint values is calculated by propagating backwards from the outputs to the inputs via

$$a_j = \sum_{i \in \text{parents}[j]} \frac{\partial x_i}{\partial x_j} a_i.$$

This enables the derivative of a single output variable to be done with respect to multiple input variables in a single pass over the expression graph. A proof of correctness follows by induction starting from the base case of the single output variable and working back to the input variables. This computes a gradient with respect to a single output function or one row of the Jacobian of a multi-output function.

Both forward- and reverse-mode require partial derivatives of each node x_i in the expression graph with respect to its daughters x_j.

Because gradients are more prevalent in contemporary algorithms, reverse-mode automatic differentiation tends to be the most efficient approach in practice. In this section we take a more detailed look at reverse-mode automatic differentiation and compare it to other differential algorithms.

1.1. Mechanics of Reverse-Mode Automatic Differentiation

As an example, consider the log of the normal probability density function for a variable y with a normal distribution with mean μ and standard deviation σ,

*input variables

*objective function

$$f(y, \mu, \sigma) = \log\left(\mathsf{Normal}(y|\mu, \sigma)\right) = -\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2 - \log\sigma - \frac{1}{2}\log(2\pi) \quad (1)$$

and its gradient,

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

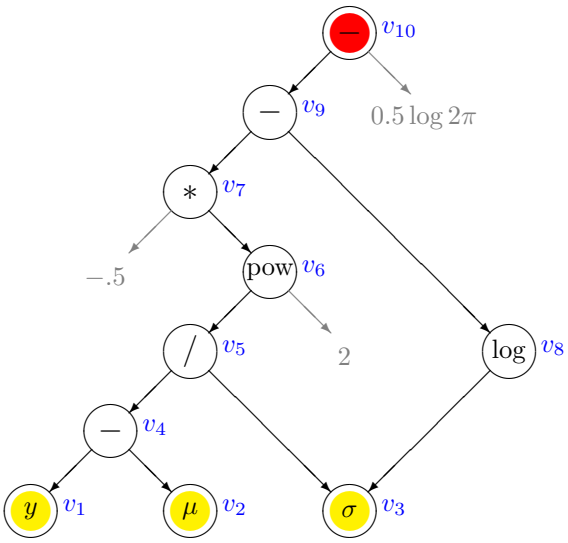

Figure 1: *Expression graph for the normal log density function given in (1). Each circle corresponds to an automatic differentiation variable, with the variable name given to the right in blue. The independent variables are highlighted in yellow on the bottom row, with the dependent variable highlighted in red on the top of the graph. The function producing each node is displayed inside the circle, with operands denoted by arrows. Constants are shown in gray with gray arrows leading to them because derivatives need not be propagated to constant operands.*

The mathematical formula for the normal log density corresponds to the expression graph in Figure 1. Each subexpression corresponds to a node in the graph, and each edge connects the node representing a function evaluation to its operands. Becuase σ is used twice in the formula, it has two parents in the graph.

Figure 2 illustrates the forward pass used by reverse-mode automatic differentiation to construct the expression graph for a program. The expression graph is constructed in the ordinary evaluation order, with each subexpression being numbered and placed on a stack. The stack is initialized here with the dependent variables, but this is not required. Each operand to an expression is evaluated before the expression node is created and placed on the stack. As a result, the stack provides a topological sort of the nodes in the graph (i.e., a sort in which each node representing an expression occurs above its subexpression nodes in the stack—see (Knuth, 1997, Section 2.2.3)). Figure 2 lists in the right column for each node, the partial derivative of the function represented by the node with respect to its operands. In the Stan Math Library, most of these partials are evaluated lazily during the reverse pass based on function's value and its operands' values.

Figure 3 shows the processing for reverse mode, which involves an adjoint value for each node. The adjoints for all nodes other than the root are initialized to 0; the root's adjoint is initialized to 1, because $\partial x / \partial x = 1$. The

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

Figure 2: *Example of gradient calculation for the the log density function of a normally distributed variable, as defined in (1). In the forward pass to construct the expression graph, a stack is constructed from the first input v_1 up to the final output v_{10}, and the values of the variables and partials are computed numerically according to the formulas given in the value and partials columns of the table.*

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	$+=$	$a_4 \times (-1)$	$a_2 = v_5/v_3$

Figure 3: *The variables v_i represent values and a_i represent corresponding adjoints. In the reverse pass, the stack is traversed from the final output down to the inputs, and as each variable is visited, each of its operands is updated with the variable's adjoint times the partial with respect to the operand. After the reverse pass finishes, (a_1, a_2, a_3) is the gradient of the density function evaluated at (y, μ, σ), which matches the correct result given in (2) after substitution for v_4 and v_5.*

backward sweep walks down the expression stack, and for each node, propagates derivatives from it down to its operands using the chain rule. Because the nodes are in topological order, by the time a node is visited, its adjoint will represent the partial derivative of the root of the overall expression with respect to the expression represented by the node. Each node then propagates its derivatives to its operands by incrementing its operands' adjoints by the product of the expression node's adjoint times the partial with respect to the operand. Thus when the reverse pass is completed, the adjoints of the independent variables hold the gradient of the dependent variable (function value) with respect to the independent variables (inputs).

1.2. Comparison to Alternative Methods of Computing Gradients

Finite Differencing

Finite differencing a numerical approximation to gradient evaluations. Given a positive difference $\epsilon > 0$, an approximate derivative can be calculated as

$$\frac{\partial f}{\partial x_n}(x) \approx \frac{f(x_1, \ldots, x_n + \epsilon, \ldots, x_N) - f(x_1, \ldots, x_N)}{\epsilon}$$

or a bit more accurately with a centered interval as

$$\frac{\partial f}{\partial x_n}(x) \approx \frac{f(x_1, \ldots, x_n + \epsilon/2, \ldots, x_N) - f(x_1, \ldots, x_n - \epsilon/2, \ldots, x_N)}{\epsilon}.$$

Although straightforward and general, calculating gradients using finite differences is slow and imprecise. Finite differencing is slow because it requires $N + 1$ function evaluations ($2N + 1$ in the centered case) to compute the gradient of an N-ary function. The numerical issues with finite differencing arise because a small ϵ is required for a good approximation of the derivative, but a large ϵ is required for floating-point precision. Small ϵ values are problematic for accuracy because because subtraction of differently scaled numbers loses a degree of precision equal to their difference in scales; for example, subtracting a number on the scale of 10^{-6} from a number on the scale of 10^0 loses 6 orders of precision in the 10^{-6}-scaled term (Higham, 2002). Thus in practice, the arithmetic precision of finite differencing is usually at best 10^{-7} rather than the maximum of roughly 10^{-14} possible with double-precision floating point calculations. There are more accurate ways to calculate derivatives with multiple differences on either side of the point being evaluated and appropriate weighting; see Fornberg (1988) for a general construction and error analysis.

Symbolic Differentiation

Whereas automatic differentiation simply evaluates a derivative at a given input, symbolic differentiation, such as Mathematica (Wolfram Research,

Inc., 2014) or SymPy (SymPy Development Team, 2014), calculate the entire derivative function analytically by applying the chain rule directly to the original function's subexpressions.

An advantage of symbolic differentation is that algebraic manipulations may be easily performed on either the input formulas or output formulas to generate more efficient or numerically stable code. For example, $\log(1 + x)$ can be rendered using the more stable `log1p` function than literally adding 1 to x (which either loses precision or underflows if x is close to 0) and taking the logarithm. While this can be done with reverse-mode automatic differentiation, it is more challenging and rarely undertaken (though an expression template approach similar to that of Adept (Hogan, 2014) could be improved upon to reduce local expressions statically); Vandevoorde and Josuttis (2002) provide an excellent overview of expression templates.

Symbolic differentiation is less expressive than reverse-mode automatic differentiation. Reverse-mode automatic differentiation can apply to any computer program involving differentiable functions and operators. This includes programs with conditionals, loops for iterative algorithms, or (recursive) function calls. A second difficulty for symbolic differentiation is that most symbolic math libraries (e.g., SymPy) are univariate and do not neatly reduce multivariate expressions involving matrices. A third difficulty for symbolic differentiation is in dealing with repeated subexpressions. The adjoint method on which reverse-mode automatic differentiation is based automatically applies an efficient dynamic programming approach that removes the kind of repeated calculations that would be involved in an unfolded expression tree. Yet another difficulty is that symbolic differentiation requires multiple passes which either require interpretation of the derivatives (what most implementations of reverse-mode automatic differentiation do, including Stan's), or a pass to compile the output of the symbolic differentiation.

The disadvantages of limited expressiveness and requiring a further compilation stage can be combined into an advantage; symbolic differentiation can generate very efficient code in cases where the same expression must be evaluated and differentiated multiple times.

A hybrid method is to use reverse-mode automatic differentiation to generate code for derivatives. This approach has the generality of reverse-mode automatic differentiation and can be advantageous in the circumstance where the same expression is evaluated multiple times. The difficulty is in parsing and dealing with all of the constructions in a complex language such as C++; existing code-generating tools such as Tapenade (Hascoët and Pascual, 2013) have only been developed for relatively simple languages like C and Fortran.

2. Calculating Gradients and Jacobians

Reverse-mode automatic differentiation in the Stan Math Library can be used to evaluate gradients of functions from $\mathbb{R}^N$ to $\mathbb{R}$ or Jacobians of differentiable functions from $\mathbb{R}^N$ to $\mathbb{R}^M$, returning values for a specified input point $x \in \mathbb{R}^N$.

2.1. Direct Calculation of Gradients of Programs

The following complete C++ program calculates derivatives of the normal log density function (1) with respect to its mean (μ) and standard deviation (σ) parameters for a constant outcome (y). The first block assigns the constant and independent variables, the second block computes the dependent variable and prints the value, and the final block computes and prints the gradients.

```
#include <cmath>
#include <stan/math.hpp>

int main() {
  using std::pow;
  double y = 1.3;
  stan::math::var mu = 0.5, sigma = 1.2;

  stan::math::var lp = 0;
  lp -= 0.5 * log(2 * stan::math::pi());
  lp -= log(sigma);
  lp -= 0.5 * pow((y - mu) / sigma, 2);
  std::cout << "f(mu, sigma) = " << lp.val() << std::endl;

  lp.grad();
  std::cout << " d.f / d.mu = " << mu.adj()
            << " d.f / d.sigma = " << sigma.adj() << std::endl;
  return 0;
}
```

*operators overloading

Constants like `y` are assigned to type `double` variables and independent variables like `mu` and `sigma` are assigned to type `stan::math::var`. The result `lp` is assigned type `stan::math::var` and calculated using ordinary C++ operations involving operators (e.g., `*`, `/`), compound assignments (e.g., `-=`), and library functions (e.g., `pow`, `log`, `pi`). The value is available through the method (member function) `val()` as soon as operations have been applied. The call to the method `grad()` propagates derivatives from the dependent variable `lp` down through the expression graph to the independent variables. The derivatives of the dependent variable with respect to the independent variables may then be extracted from the independent variables with the method `adj()`.

The gradient evaluated at the input can also be extracted as a standard vector using the method `grad()` of `stan::math::var`. Included libraries are not duplicated from the previous code.

```
#include <vector>

int main() {
  double y = 1.3;
  stan::math::var mu = 0.5, sigma = 1.2;

  stan::math::var lp = 0;
  lp -= 0.5 * log(2 * stan::math::pi());
  lp -= log(sigma);
  lp -= 0.5 * pow((y - mu) / sigma, 2);

  std::vector<stan::math::var> theta;
  theta.push_back(mu);   theta.push_back(sigma);
  std::vector<double> g;
  lp.grad(theta, g);   *no need for adj
  std::cout << " d.f / d.mu = " << g[0]
            << " d.f / d.sigma = " << g[1] << std::endl;
  return 0;
}
```

The standard vector `theta` holds the dependent variables and the standard vector `g` is used to hold the result. The function `grad()` function is called on the dependent variable to propagate the derivatives and fill `g` with the gradient. The method `grad()` will resize the vector `g` if it is not the right size.

2.2. Coding Template Functions for Automatic Differentiation

The previous example was implemented directly in the main function block using primitive operations. The Stan Math Library implements all of the built-in C++ boolean and arithmetic operators as well as the assignment and compound assignment operators. It also implements all of the library functions. Examples of operator, assignment, and function implementations are given later.

* From a user perspective, a function can be automatically differentiated with respect to some input parameters if the arguments to be differentiated can all be instantiated to `stan::math::var` types. For the most flexibility, functions should be separately templated in all of their arguments so as to support any combination of primitive (e.g., `double`, `int`) and autodiff (`stan::math::var`) instantiations.

For example, the following templated C++ function computes the log normal density as defined in (1).

```
#include <boost/math/tools/promotion.hpp>

template <typename T1, typename T2, typename T3>
inline
typename boost::math::tools::promote_args<T1, T2, T3>::type
normal_log(const T1& y, const T2& mu, const T3& sigma) {
  using std::pow;  using std::log;
  return -0.5 * pow((y - mu) / sigma, 2.0)
    - log(sigma)
    - 0.5 * log(2 * stan::math::pi());
}
```

Argument-Dependent Lookup for Function Resolution

In order to allow built-in functions such as `log()` and `pow()` to be instantiated with arguments of type `stan::math::var` and primitive C++ types, the primitive version of the function is brought in with a `using` statement. For example, `using std::pow` brings the version of `pow()` that applies to `double` arguments into scope. The definition of `pow()` for autodiff variables `stan::math` is brought in through argument-dependent lookup (International Standardization Organization, 2003, Section 3.4), which brings the namespace of any argument variables into scope for the purposes of resolving function applications.

Traits Metaprogram for Computing Return Types

Boost's traits-based metaprogram `promote_args` (Schäling, 2011) is designed to calculate return types for highly templated functions like `normal_log()`; Vandevoorde and Josuttis (2002) provide an excellent overview of trait-based template metaprogramming.

In general, for an automatically differentiated function, the return type should be `double` if all the input arguments are primitive integers or double-precision floating point values, and `stan::math::var` if any of the arguments is of type `stan::math::var`.

Given a sequence of types `T1`, ..., `TN`, the template structure `promote_args<T1, ..., TN>` defines a typedef named `type`, which is defined to be `double` if all the inputs are `int` or `double`, and `stan::math::var` if any of the input types is `stan::math::var`. The keyword `typename` is required to let the C++ parser know that the member variable is a typedef rather than a regular value. The Boost promotion mechanism is used throughout the Stan Math Library to define both return types and types of variables for intermediate results. For instance, `y - mu` would be assigned to an intermediate variable of type `promote_args<T1, T2>::type`.

The fully templated definition allows the template parameters `T1`, `T2`, or `T3` to be instantiated as independent variables (`stan::math::var`) or constants

(`double`, `int`). For example, it can be used in place of the direct definitions in the previous programs as

```
...
double y = 1.3;
stan::math::var mu = 0.5, sigma = 1.2;

stan::math::var lp = normal_log(y, mu, sigma);
...
```

No explicit template specifications are necessary on the function—C++ infers the template type parameters from the types of the arguments.

2.3. Calculating the Derivatives of Functors with Functionals

The Stan Math Library provides a fully abstracted approach to automatic differentiation that uses a C++ functor to represent a function to be differentiated and a functional for the gradient operator.

Eigen C++ Library for Matrices and Linear Algebra

The Eigen C++ library for matrix operations and linear algebra (Guennebaud et al., 2010) is used throughout the Stan Math Library. The type `Matrix<T, R, C>` is the Eigen type for matrices containing elements of type `T`, with row type `R` and column type `C`; the automatic differentiation type `stan::math::var` can be used for the element type `T`. The Stan Math Library uses three possible instantiations for the row and column type, `Matrix<T, Dynamic, 1>` for (column) vectors, `Matrix<T, 1, Dynamic>` for row vectors, and `Matrix<T, Dynamic, Dynamic>` for matrices. These three instances are all specialized with their own operators in the Eigen library. Like the standard template library's `std::vector` class, these all allocate memory for elements dynamically on the C++ heap following the resource allocation is instantiation (RAII) pattern (Stroustrup, 1994, p. 389). The RAII pattern is used to encapsulate memory allocation and freeing by allocating memory when an object is instantiated and freeing it when an object goes out of scope.

Definining Functors in C++

A functor in C++ is a function that defines `operator()` and can hence behaves syntactically like a function. For example, the normal log likelihood function for a sequence of observations may be defined directly as a functor as follows.

```
#include <Eigen/Dense>

using Eigen::Matrix;
using Eigen::Dynamic;
```

```
struct normal_ll {
  const Matrix<double, Dynamic, 1> y_;

  normal_ll(const Matrix<double, Dynamic, 1>& y) : y_(y) { }

  template <typename T>
  T operator()(const Matrix<T, Dynamic, 1>& theta) const {
    T mu = theta[0];
    T sigma = theta[1];
    T lp = 0;
    for (int n = 0; n < y_.size(); ++n)
      lp += normal_log(y_[n], mu, sigma);
    return lp;
  }
};
```

The variable `y_` is used to store the data vector in the structure. The `operator()` defines a function whose argument argument type is the Eigen type for vectors with elements of type `T`; the result is also defined to be of type `T`.

Calculating Gradients with Functionals

A functional in C++ is a function that applies to a functor. The Stan Math Library provides a functional `stan::math::gradient()` that differentiates functors implementing a constant operator over Eigen vectors with automatic differentiation scalars, i.e.,

```
var operator()(const Matrix<var, Dynamic, 1>& x) const;
```

The example functor `normal_ll` defined in the prvious section provides a templated operator, the template parameter `T` of which can be instantiated to `stan::math::var` for differentiation and to `double` for testing. The following code calculates the gradient of the functor at a specified input.

```
Matrix<double, Dynamic, 1> y(3);
y << 1.3, 2.7, -1.9;
normal_ll f(y);

Matrix<double, Dynamic, 1> x(2);
x << 1.3, 2.9;

double fx;
Matrix<double, Dynamic, 1> grad_fx;
stan::math::gradient(f, x, fx, grad_fx);
```

The argument `f` is the functor, which is the log likelihood function instantiated with a vector of data, `x`. The argument vector `x` is filled with the

parameter values. The scalar `fx` is defined to hold the function value and an Eigen vector `grad_fx` is defined to hold the gradient. The functional `stan::math::gradient` is then called to calculate the value and the gradient from the function `f` and input `x`, setting `fx` to the calculated function value and setting `grad_fx` as the gradient. See Section 6 for information on how the `stan::math::gradient` functional is implemented.

2.4. Calculating Jacobians

With reverse-mode automatic differentiation, a single forward pass can be used to construct the complete expression graph, then a reverse pass can be carried out for each output dimension.

Direct Jacobian Calculation

Suppose that `f` is a functor that accepts an dimensional Eigen N-vector of autodiff variables as input (typically through templating) and produces an Eigen M-vector as output. The following code calculates the Jacobian of `f` evaluated at the input `x`.

```
Matrix<double, Dynamic, 1> x = ...;   // inputs

Matrix<var, Dynamic, 1> x_var(x.size());
for (int i = 0; i < x.size(); ++i) x_var(i) = x(i);

Matrix<var, Dynamic, 1> f_x_var = f(x_var);

Matrix<double, Dynamic, 1> f_x(f_x_var.size());
for (int i = 0; i < f_x.size(); ++i) f_x(i) = f_x_var(i).val();

Matrix<double, Dynamic, Dynamic> J(f_x_var.size(), x_var.size());
for (int i = 0; i < f_x_var.size(); ++i) {
  if (i > 0) stan::math::set_zero_all_adjoints();
  f_x_var(i).grad();
  for (int j = 0; j < x_var.size(); ++j)
    J(i,j) = x_var(j).adj();
}
```

First, the arguments are used to create autodiff variables. Next, the function is applied and the result is converted back to a vector of doubles. Then for each output dimension, automatic differentiation calculates the gradient of that dimension and uses it to populate a row of the Jacobian. After the first gradient is calculated, all subsequent gradient calculations begin by setting the adjoints to zero; this is not required for the first gradient because the adjoints are initialized to zero.

Functional Jacobian Calculation

Alternatively, the Jacobian functional can be applied directly to the function `f`, as follows.

```
...
Matrix<double, Dynamic, Dynamic> J;
Matrix<double, Dynamic, 1> f_x;
stan::math::jacobian(f, x, f_x, J);
```

Like the `grad()` functional, the `jacobian()` functional will automatically resize `J` and `f_x` if necessary.

3. Automatic Differentiation Variable Base Classes

As demonstrated in the previous section, the client-facing data type of the Stan Math Library is the type `stan::math::var`, which is used in place of primitive `double` values in functions that are to be automatically differentiated.

3.1. Pointers to Implementations

The `var` class is implemented following the pointer to implementation (Pimpl) pattern (Sutter, 1998, 2001). The implementation class is type `vari`, and is covered in the next subsection. Like the factory pattern, the Pimpl pattern encapsulates the details of the implementation class(es), which the application programmer interface (API) need not expose to clients. For example, none of the example programs in the last section involved the `vari` type explicitly. As described in the next section, Pimpl classes often encapsulate the messy and error-prone pointer manipulations required for allocating and freeing memory.

3.2. Resource Allocation is Initialization

Like many Pimpl classes, `var` manages memory using the resource allocation is initialization (RAII) pattern (Stroustrup, 1994). With RAII, class instances are managed on the stack and passed to functions via constant references. Memory management is handled behind the scenes of the client-facing API. Memory is allocated when an object is constructed or resized. Memory is freed when the object goes out of scope.

Other classes implemented using Pimpl and RAII include `std::vector` and `Eigen::Matrix`, both of which allocate memory directly on the C++ heap and deallocate it when their variables go out of scope and their destructors are called. As explained below, memory is managed in a custom arena for `stan::math::var` instances rather than being allocated on the general-purpose C++ heap.

3.3. The `var` Class

The core of the `stan::math::var` class is a pointer to a `stan::math::vari` implementation.

```
class var {
public:
  var() : vi_(static_cast<vari*>(0U)) { }
  var(double v) : vi_(new vari(v)) { }

  double val() const { return vi_->val_; }
  double adj() const { return vi_->adj_; }

private:
  vari* vi_;
};
```

The default constructor `var()` uses a null value for its `vari` to avoid memory-allocation that would just be overwritten. When the constructor is given a `double` value, a new `vari` instance is constructed for the value. The memory management for `vari` is handled through a specialization of `operator new` as described below.

The `var` class additionally defines a copy constructor, constructors for the other primitive types, and the full set of assignment and compound assignment operators. The destructor is implicit because there is nothing to do to clean up instances; all of the work is done with the memory management for `vari`. The class is defined in the `stan::math` namespace, but the namespace declarations are not shown to save space.

3.4. The `chainable` Base Class

The design of Stan's variable implementation classes is object-oriented, in that subclasses extend `chainable` and implement the virtual method `chain()` in a function-specific way to propagate derivatives with the chain rule. The class `vari` extends the base class `chainable` and holds a value and an adjoint. Extensisons of `vari` will hold pointers to operands for propagating derivatives and sometimes store partial derivatives directly.

```
struct chainable {
  chainable() { }
  virtual ~chainable() { }

  virtual void chain() { }
  virtual void init_dependent() { }
  virtual void set_zero_adjoint() { }

  static inline void* operator new(size_t nbytes) {
```

```
    return ChainableStack::memalloc_.alloc(nbytes);
  }
};
```

The virtual method `chain()` has no body, but allows extensions of `chainable` to implement derivative propagation. The static specialization of `operator new` provides arena-based memory management for extensions of `chainable` (see Section 5). The initialization of dependents and set-zero methods are also virtual; they are used to initialize the adjoints during the derivative propagation (reverse) pass.

3.5. The `vari` Class

The `stan::math::vari` class extends the base class `stan::math::chainable` and holds a value and adjoint for a variable instantiation.

```
class vari : public chainable {
public:
  const double val_;
  double adj_;

  vari(double v) : val_(v), adj_(0) {
    ChainableStack::var_stack_.push_back(this);
  }

  virtual ~vari() { }

  virtual void init_dependent() { adj_ = 1; }
  virtual void set_zero_adjoint() { adj_ = 0; }
};
```

The initialization method for dependent (output) variables sets the adjoint to 1; it is called on the output variable whose derivative is being calculated before propagating derivatives down to input (independent) variables. The set-zero method sets the adjoint to 0, which is the value required for all other variables in the expression graph before derivative propagation; because adjoints are initialized to zero, it is only called in situations where multiple derivatives are required, as in Jacobian calculations.

4. Calculating Gradients

There is a static method in the `stan::math::chainable` class which performs the derivative propagation.

```
static void grad(chainable* vi) {
  typedef std::vector<chainable*>::reverse_iterator it_t;
```

```
    vi->init_dependent();
    it_t begin = ChainableStack::var_stack_.rbegin();
    it_t end = ChainableStack::var_stack_.rend();
    for (it_t it = begin; it < end; ++it)
      (*it)->chain();
  }
```

The function initializes the dependent variable to 1 using the `init_dependent` method on `chainable`. Then a reverse iterator is used to iterate over the variable stack from the top (`rbegin`) down to the bottom (`rend`), executing each `chainable` instance's `chain()` method. Because the stack of `chainable` instances is sorted in topological order, each node's parents (superexpressions in which it directly appears) will have all been visited by the time the node's chain-rule propagation method is called.

4.1. Jacobians and Zeroing Adjoints

To calculate Jacobians efficiently, the expression graph is constructed once and then reused to calculate the gradient required for each row of the Jacobian. Before each gradient calculation after the first, the static method `stan::math::set_zero_all_adjoints()` is called. This function walks over the variable stack `stan::math::ChainableStack::var_stack_`, resetting each value to zero using the virtual method `set_zero_adjoint()`.

```
  static void set_zero_all_adjoints() {
    for (size_t i = 0; i < ChainableStack::var_stack_.size(); ++i)
      ChainableStack::var_stack_[i]->set_zero_adjoint();
  }
```

As in all of the client-facing API methods, pointer and global variable manipulation in encapsulated.

5. Memory Management

Instances of `vari` and its extensions are allocated using the specialized static `operator new` definition in `chainable`. The specialization references the `alloc(size_t)` method of the global variable `ChainableStack::memalloc_`. This global variable holds the custom arena-based memory used for `vari` instances. Using an arena allows the memory for a gradient calculation to be efficiently allocated elemtentwise and then freed all at once after derivative calculations have been performed (see, e.g., Gay and Aiken (2001); Gay (2005)).

The core of the `stan::math::ChainableStack` template class definition, which holds the memory arena for reverse-mode autodiff, is as follows.

```
  template<typename T, typename AllocT>
  struct AutodiffStackStorage {
```

```
  static std::vector<T*> var_stack_;
  static std::vector<AllocT*> var_alloc_stack_;
  static stack_alloc memalloc_;
};

struct chainable_alloc {
  chainable_alloc() {
    ChainableStack::var_alloc_stack_.push_back(this);
  }
  virtual ~chainable_alloc() { };
};

typedef AutodiffStackStorage<chainable,chainable_alloc>
  ChainableStack;
```

The variables are all declared static, so they serve as global variables in the program. The variable `var_stack_` holds the stack of nodes in the expression graph, each represented as a pointer to a `chainable`. This stack is traversed in reverse order of construction to propagate derivatives from expressions to their operands. The variable `memalloc_` holds the byte-level memory allocator, which is described in the next section.

The typedef `ChainableStack` specifies the template parameters for the template class `AutodiffStackStorage`. Rather than creating instances, the global variables are accessed through the typedef instantiation (e.g., `ChainableStack::var_stack_`).

The `var_alloc_stack` variable holds objects allocated for autodiff that need to have their `delete()` methods called. Such variables are required to specialize `chainable_alloc`, which pushes them onto the `var_alloc_stack_` during construction. The destructor is virtual for `chainable_alloc` because extensions with customized destructors will be placed on the `var_alloc_stack_`.

5.1. Byte-Level Memory Manager: `stack_alloc`

The byte-level memory manager uses an arena-based strategy where blocks of memory are allocated and used to hold the `vari` instances created during the forward expression-graph building pass of automatic differentiation. After the partials are propagated and final gradients calculated in the reverse pass, the memory for the expression graph is reclaimed all at once.

The memory used for `chainable` instances is managed as a standard vector of `char*` blocks. New variables are filled into the blocks until they will no longer fit, at which point a new block is allocated. The data structures for memory management are defined as member variables in the `stan::math::stack_alloc` class.

```
class stack_alloc {
```

```
private:
  std::vector<char*> blocks_;
  std::vector<size_t> sizes_;
  size_t cur_block_;
  char* cur_block_end_;
  char* next_loc_;
  ...
```

The variable `blocks_` holds blocks of memory allocated on the heap. The parallel vector `sizes_` stores the size of each block. The remaining variables indicate the index of the current block, the end of the current block's available memory, and the next location within the current block to place a new variable implementation.

Given a request for a given block of memory, if it fits, it will be allocated starting at the next location in the current block. Otherwise, a new block is allocated that is twice the size of the current block and the request is attempted again.

Because memory allocation is such a low-level operation in the algorithm and because CPU branch misprediction is so expensive, the ability of the GNU compiler family to accept guidance in terms of how to sequence generated assembly code is used through the following macros.

```
#ifdef __GNUC__
#define likely(x)      __builtin_expect(!!(x), 1)
#define unlikely(x)    __builtin_expect(!!(x), 0)
#else
#define likely(x)     (x)
#define unlikely(x)   (x)
#endif
```

Then a condition is wrapped in `likely(...)` or `unlikely(...)` to give the compiler a hint as to which branch to predict by default. This improved throughput by a factor of nearly 10% compared to the unadorned code. The allocation method in `stack_alloc` is defined as

```
inline void* alloc(size_t len) {
  char* result = next_loc_;
  next_loc_ += len;
  if (unlikely(next_loc_ >= cur_block_end_))
    result = move_to_next_block(len);
  return (void*)result;
}
```

Thus running out of memory in a block is marked as unlikely.

The `move_to_next_block` method, not shown here, allocates a new block of memory and updates the underlying memory stacks. No movement of existing data is required and thus no additional memory overhead is required to store the original and copy. All of the memory allocation is also done to ensure 8-byte alignment for maximum access speed of pointers in 64-bit architectures.

5.2. Memory Lifecycle

An instance of `vari` is constructed for each subexpression evaluated in the program that involves at least one `var` argument. For each `vari` constructed, memory is allocated in the arena and a pointer to the memory allocated is pushed onto the variable stack. Both the bytes and the stack of pointers must persist until the call to `grad` is used to compute derivatives, at which point it may be recovered.

The `AutodiffStackStorage` class holds the variable stacks and an instance of `stack_alloc` for the bytes used to create `chainable` instances. The `AutodiffStackStorage` class also implements a no-argument `recover_memory()` method which frees the stack of pointers making up the variable stack, and resets the byte-level `stack_alloc` instance. The underlying blocks of memory allocated within the `stack_alloc` instance are intentionally *not* freed by the `recover_memory()` method. Rather, the underlying memory pool is saved and reused when the next call to automatic differentiation is made. A method is available to clients in the `stack_alloc` class to completely free the underlying memory pools.

The functionals for automatic differentiation all provide exception-safe memory recovery that ensures whenever a derivative functional is called memory cannot leak even if an operation within the functor throws an exception. The functionals do not themselves free the underlying memory pool, but clients can do that manually after functionals are called if desired.

6. Gradient Functional

The functional `gradient()` encapsulates the gradient calculations shown in the direct implementation as follows (the namespace qualifications of functions and classes are omitted for readability).

```
template <typename F>
void gradient(const F& f,
              const Matrix<double, Dynamic, 1>& x,
              double& fx,
              Matrix<double, Dynamic, 1>& grad_fx) {
  try {
    Matrix<var, Dynamic, 1> x_var(x.size());
    for (int i = 0; i < x.size(); ++i)
      x_var(i) = x(i);
    var fx_var = f(x_var);
    fx = fx_var.val();
    grad(fx_var.vi_);
    grad_fx.resize(x.size());
    for (int i = 0; i < x.size(); ++i)
      grad_fx(i) = x_var(i).adj();
```

```
  } catch (const std::exception& /*e*/) {
    recover_memory();
    throw;
  }
  recover_memory();
}
```

Client code, as our earlier example shows, does not need to specify the template parameter `F` explicitly, because it can be automatically inferred from the static type of the argument functor `f`. Because the functor argument reference is declared `const` here, its `operator()` implementation must also be marked `const`. The final two arguments, `fx` and `grad_fx`, will be set to the value of the function and the value of the gradient respectively.

All of the code is wrapped in a `try` block with a corresponding `catch` that recovers memory and rethrows whatever standard exception was caught. If no exception is thrown in the block of the `try`, then the function terminates with a call to recover memory (which only resets memory—it does not free any of the underlying blocks that have been allocated, saving them for reuse).

The main body of the algorithm first copies the input vector of `double` values, `x`, into a vector of `var` values, `x_var`. Then the functor `f` is applied to this newly created vector `x_var`, resulting in a type `var` result. This usage requires `f` to implement a constant `operator()` which takes a vector of `var` as its single argument and returns a `var`. Any other information required to compute the function, such as data vectors (`double` or `int` values) or external callbacks such as a logging or error reporting mechanism, must be accessible to the `operator()` definition for the functor `f`.

The evaluation of `f` may throw an exception, which will result in the automatic differentiation memory being recovered. If no exception is thrown, the function returns a `var`, the value of which is then assigned to the argument reference `fx`.

Next, the static `grad` function is called directly on the variable implementation of the result. The derivatives are then extracted from the dependent variables in the argument vector and assigned to the argument reference vector `grad_fx`, first resizing `grad_fx` if necessary.

7. Constants

There are two equivalent ways to create constant autodiff variables from constant `double` values. The first approach uses the unary constructor for `var` instances, which creates a new `var` instance `sigma` on the C++ function-call stack.

```
var sigma(0.0);
```

As shown in Section 3, an underlying `vari` instance will be constructed in the memory arena.

The second approach to constructing a constant autodiff variable uses assignment. Because the unary constructor of `var` is implicit (i.e., not declared `explicit`), assignment of a `double` to a `var` will construct a new `var` instance.

```
var tau = 0.0;
```

The behavior of `sigma` and `tau` are the same because they have exactly the same effect on memory and the autodiff stack.

It is also possible to construct instances using integers or other base types such as `unsigned long` or `float`. The additional constructors are not shown. Because the copy constructor is declared explicit, there is no ambiguity in the constructors.

A `vari` inherits the no-op `chain()` method implementation from its superclass `chainable`. Therefore, a constant will not propagate any derivative information. Where possible, it is best to use `double` values directly as arguments to functions and operators, because it will cut down on the size of the variable stack and thus speed up gradient calculations.

8. Functions and Operators

This section describes how functions are implemented in Stan for reverse-mode automatic differentiation.

8.1. Unary Operand Storage Class

Most of the unary functions in the Stan Math Library are implemented as extensions of a simple helper class, `op_v_vari`, which stores the value of a unary function and its operand.

```
struct op_v_vari : public vari {
  vari* avi_;

  op_v_vari(double f, vari* avi) : vari(f), avi_(avi) { }
};
```

Constructing a `op_v_vari` instance passes the function value `f` to the superclass constructor, which stores it in member variable `val_`. The second argument to the constructor is a `vari` pointer, which is stored here in the member variable `avi_`. The `vari` will be set to The `op_v_vari` inherits the no-op `chain()` method from `chainable`.

8.2. Unary Function Implementation

As a running example, consider the natural logarithm function, the derivative of which is

$$\log'(x) = \frac{1}{x}.$$

The implementation of the `log` function for `var` constructs a specialized `vari` instance as follows.

```
inline var log(const var& a) {
  return var(new log_vari(a.vi_));
}
```

The argument `a.vi_` supplied to the constructor is a pointer to the `vari` implementing `a`; this pointer is stored as the operand for the `log` function.

The class `log_vari` is defined as follows.

```
struct log_vari : public op_v_vari {
  log_vari(vari* avi) :
    op_v_vari(std::log(avi->val_), avi) { }

  void chain() {
    avi_->adj_ += adj_ / avi_->val_;
  }
};
```

The constructor takes a pointer, `avi`, to the operand's implementation. The value of the operand is given by `avi->val_`. The log of the operand's value is computed using `std::log()` and passed to the `op_v_vari` constructor, which stores it, along with the operand implementation itself, `avi`.

The virtual `chain()` method of the superclass `chainable` is overriden to divide the variable's adjoint, `adj_`, by the operand's value, `avi_->val_`, and use the result to increment the operand's adjoint, `avi_->adj_`. For comparison, the operation shown in Figure 3 for a log is $a_3 \mathrel{+}= a_8 \times (1/v_3)$, in which a_3 and v_3 are the adjoint and value of the operand and a_8 the adjoint of the result.

The `chain()` method is not called until the reverse pass of automatic differentiation is executed, so the derivative $1/x$ times the adjoint is evaluated lazily as `adj_ / avi_->val_`, which requires only a single division. The eager approach (as used by other packages such as Sacado and Adept) store $1/x$ as the partial in the forward pass and then multiply that stored value by the adjoint in the reverse pass. The eager approach requires additional storage for the partial $1/x$, which must be set and later accessed. The eager approach also carries out an extra operation in multiplying by the inverse rather than just dividing as in the lazy approach, effectively evaluating `adj_ * (1 / avi_->val_)` rather than `adj_ / avi_->val_`.

Most other Stan functions are implemented in the same way as `log`, with their own specialized `chain()` implementation and storage. The savings for operations like addition and subtraction are even larger because there is no need to do any multiplication at all.

8.3. Binary Functions

As an example, consider the `pow()` function, defined by

$$\text{pow}(x, y) = x^y,$$

with derivatives

$$\frac{\partial}{\partial x} x^y = y\, x^{y-1}$$

and

$$\frac{\partial}{\partial y} x^y = x^y \log x.$$

Binary Operands Storage Class

Binary functions of two autodiff variables are implemented using the following specialized `vari` implementation.

```
struct op_vv_vari : public vari {
  vari* avi_;   vari* bvi_;

  op_vv_vari(double f, vari* avi, vari* bvi)
  : vari(f), avi_(avi), bvi_(bvi) { }
};
```

Like its unary counterpart, this class stores the value (`f`) and pointers to the operands (`avi_`, `bvi_`). The number of operands need not be stored explicitly because it is implicit in the identity of the class.

For mixed operations of `var` and `double` variables, the following base `vari` is provided.

```
struct op_vd_vari : public vari {
  vari* avi_;
  double bd_;

  op_vd_vari(double f, vari* avi, double b)
  : vari(f), avi_(avi), bd_(b) {  }
};
```

Although not strictly necessary given the above, the following symmetric class is provided for naming convenience.

```
struct op_dv_vari : public vari {
  double ad_;
  vari* bvi_;

  op_dv_vari(double f, double a, vari* bvi)
  : vari(f), ad_(a), bvi_(bvi) { }
};
```

Binary Function Implementation

There are three function implementations for `pow()` based on the type of the arguments, and each has its own specialized implementation, the first of of which follows the same pattern as `log`, only with two arguments.

```
inline var pow(const var& base, const var& exponent) {
  return var(new pow_vv_vari(base.vi_,exponent.vi_));
}
```

The second specialization is similar, only the `double` argument is passed by value to the specialized `vari` constructor.

```
inline var pow(double base, const var& exponent) {
  return var(new pow_dv_vari(base,exponent.vi_));
}
```

The last implementation with a `double` exponent first checks if there is a built-in special case that can be used, and if not, constructs a specialized `vari` for power.

```
inline var pow(const var& base, double exponent) {
  if (exponent == 0.5) return sqrt(base);
  if (exponent == 1.0) return base;
  if (exponent == 2.0) return square(base);
  return var(new pow_vd_vari(base.vi_,exponent));
}
```

By using `sqrt`, `square`, or a no-op, less memory is allocated and fewer arithmetic operations are needed during the evaluation of `chain()` for the remaining expressions, thus saving both time and space.

Variable Implementations

The `pow_vv_vari` class for `pow(var,var)` is defined as follows.

```
struct pow_vv_vari : public op_vv_vari {
  pow_vv_vari(vari* avi, vari* bvi)
  : op_vv_vari(std::pow(avi->val_,bvi->val_),avi,bvi) { }

  void chain() {
    if (avi_->val_ == 0.0) return;
    avi_->adj_ += adj_ * bvi_->val_ * val_ / avi_->val_;
    bvi_->adj_ += adj_ * std::log(avi_->val_) * val_;
  }
};
```

The constructor calls the `op_vv_vari` constructor to store the value and the two operand pointers. The `chain()` implementation first tests if the base is 0,

and returns because the derivatives are both zero, and thus there is nothing to do for the chain rule. This test is not for efficiency, but rather because it avoids evaluating the logarithm of zero, which evaluates to special not-a-number value in IEEE floating-point arithmetic.[2] In the normal case of execution when the base is not zero, the `chain()` method increments the operands' adjoints based on the derivatives given above. The expression for the derivative for the power operand (`bvi_`) conveniently involves the value of the function itself, just as `exp` did.

The implementations for mixed inputs only need to compute a single derivative propagation.

```
struct pow_vd_vari : public op_vd_vari {
  pow_vd_vari(vari* avi, double b) :
    op_vd_vari(std::pow(avi->val_,b),avi,b) {
  }

  void chain() {
    avi_->adj_ += adj_ * bd_ * val_ / avi_->val_;
  }
};
```

The base argument (`avi`) is a variable implementation pointer that is stored and accessed in `chain()` as `avi_`. The exponent argument (`b`) is stored and accessed as `bd_`.

8.4. Arithmetic Operators

Basic arithmetic operators are implemented in exactly the same way as functions. For example, addition of two variables is implemented using a support class `add_vv_vari`.

```
inline var operator+(const var& a, const var& b) {
  return var(new add_vv_vari(a.vi_,b.vi_));
}
```

The implementation of `add_vv_vari` follows that of `exp`, with the same naming conventions. The derivatives are

$$\frac{\partial}{\partial x}(x+y) = 1$$

and

$$\frac{\partial}{\partial y}(x+y) = 1.$$

As a result, the `chain()` method can be reduced to just adding the expression's adjoints to that of its operands.

[2] An even more efficient alternative would be to evaluate this condition ahead of time and just return a constant zero `var` at the top level.

```
void chain() {
  avi_->adj_ += adj_;
  bvi_->adj_ += adj_;
}
```

This is particularly efficient because the derivative values of 1 need not be stored or mutliplied.

The mixed input operators are specialized in the same way as the mixed input functions. The other built-in operators for unary subtraction, multiplication, and division are implemented similarly.

8.5. Boolean Operators

The boolean operators are implemented directly without creating new variable implementation instances. For example, equality of two variables is implemented as follows.

```
inline bool operator==(const var& a, const var& b) {
  return a.val() == b.val();
}
```

Mixed input types are handled similarly.

```
inline bool operator==(const var& a, double b) {
  return a.val() == b;
}
```

The other boolean operators are defined the same way.

8.6. Memory Usage

The amount of memory required for automatic differentiation is the sum of the memory required for each expression in the expression graph. These expressions are represented with data structures like `log_vari`. The storage cost in the arena is as follows for a function with K operands.

Description	*Type*	*Size (bytes)*
value	`double`	8
operand pointers	`size_t`	$K \times 8$
vtable pointer	`size_t`	8
`var_stack_` ptr	`size_t`	8
TOTAL		$24 + K \times 8$

Each expression stores its value using double-precision floating point, requiring 8 bytes. Pointers are stored to the operand(s), requiring 8 bytes per pointer. The number of operands is not stored, but is rather implicit in the identity of the class used for the `vari` and how it computes `chain()`.

Wrapping the `vari*` in a `var` does not increase memory usage. The `var` itself is allocated on the stack and because it has no virtual functions, occupies the same amount of memory as a `vari*` pointer.

In addition to the memory required for the `vari` instances, each expression gets a pointer on the variable stack, `var_stack_`. The price for object orientation in the virtual `chain()` method is 8 bytes for the vtable pointer, which most implementations of C++ includes in order to dynamically resolve the implementation of virtual methods. The cost for storing a heterogeneous collection of expressions on the variable stack is either a virtual function lookup using the vtable or pointer chasing to implementations with function pointers.

9. Assignment and Compound Assignment

9.1. Assignment Operator

The assignment operator, `operator=`, is defined implicitly for `var` types. Default assignment of `var` instances to `var` instances is handled through the C++ defaults, which perform a shallow copy of all member variables, in this case copying the `vari*` pointer.

For assigning `double` values to `var` instances, the implicit constructor `var(double)` is used to construct a new variable instance and its `vari*` pointer is copied into the left-hand side `var`.

Assigning `var` typed expressions to `double` variables is forbidden.

9.2. Compound Assignment Operators

The compound assignment operators such as `+=` are used as follows.

```
var a = 3;
var b = 4;
a += b;
```

The above is intended to have the same effect as the following.

```
a = a + b;
```

The compound operators are declared as member functions in `var`.

```
struct var {
...
  inline var& operator*=(const var& b);
  inline var& operator*=(double b);

};
```

The left-hand side is implicitly the variable in which the operators are declared.

The implementations are defined outside the class declaration. For variable arguments, the definition is as follows; the implementation for `double` arguments is similar.

```
inline var& var::operator*=(const var& b) {
  vi_ = new multiply_vv_vari(vi_,b.vi_);
  return *this;
}
```

A new `multiply_vv_vari` instance is created and the pointer to it assigned to the member variable `vi_` of the left-hand side variable in the compound assignment. The expression `*this` is the value of the left-hand side variable in the assignment, which will be returned by reference according to the declared return type `var&`.

The effect of `a += b;` is to modify `a`, assigning it to `a + b`. This is a destructive operation on `var` instances such as `a`, but it is not a destructive operation on variable implementations. A sequence such as

```
var a = 0;
a + = b * b;
a + = c * c;
```

winds up creating the same expression graph as

```
var a = b * b + c * c;
```

with the same value pointed to by `a.vi_` in the end.

10. Variadic Functions

Some functions, such as sum, can take vectors of arbitrary size of arguments. For such variadic functions, `op_vector_vari` provides a base class with member variables for the array of operands and the array's size.

```
struct op_vector_vari : public vari {
  const size_t size_;
  vari** vis_;

  op_vector_vari(double f,
             const std::vector<stan::math::var>& vs)
  : vari(f), size_(vs.size()),
    vis_(static_cast<vari**>(operator new(sizeof(vari*)
                                          * vs.size()))) {
    for (size_t i = 0; i < vs.size(); ++i)
      vis_[i] = vs[i].vi_;
  }
};
```

Because it is used in an extension of `vari`, the specialized memory arena `operator new` is used to allocate the memory. Rather than use a `std::vector`, which allocates memory on the standard C++ heap, using the memory arena avoids fragmentation and also avoids the overhead of calling destructors.

10.1. Log Sum of Exponentials

One example that is useful in statistical models to prevent overflow and underflow is the log sum of exponentials operation, defined for an N-vector x by

$$\text{log\_sum\_exp}(x) = \log \sum_{n=1}^{N} \exp(x_n).$$

The function is symmetric in its arguments and the derivative with respect to x_n is

$$\frac{\partial}{\partial x_n} \text{log\_sum\_exp}(x) = \frac{\exp(x_n)}{\sum_{n=1}^{N} \exp(x_n)}.$$

The log sum of exponential operator has the folloiwng implementation class.

```
struct log_sum_exp_vector_vari : public op_vector_vari {
  log_sum_exp_vector_vari(const std::vector<var>& x)
  : op_vector_vari(log_sum_exp_as_double(x), x) { }

  void chain() {
    for (size_t i = 0; i < size_; ++i)
      vis_[i]->adj_ += adj_ * std::exp(vis_[i]->val_ - val_);
  }
};
```

The `chain()` method here loops over the operand pointers stored in the array `vis_`, and for each one, increments its adjoint with the adjoint of the result times the derivative with respect to the operand, as given by

$$\exp(x_n - \text{log\_sum\_exp}(x)) \;=\; \frac{\exp(x_n)}{\exp(\text{log\_sum\_exp}(x))} \;=\; \frac{\exp(x_n)}{\sum_{n=1}^{N} \exp(x_n)}.$$

The function to compute the value, `log_sum_exp_as_double` is not shown here; it uses the usual algebraic trick to preserve the most significant part of the result and avoid overflow on the remaining calculations, by

$$\log \sum_{n=1}^{N} \exp(x_n) = \max(x) + \log \sum_{n=1}^{N} \exp(x_n - \max(x)).$$

The data structure only requires storage for the operands and an additional size pointer. There is also only a single virtual function call to `chain` that propagates derivatives for each argument.

10.2. Sum Accumulation

The Stan Math Library was developed to calculate gradients of log density functions, which are typically composed of sums of simpler log probability

functions. A typical structure used in a simple linear regression would be the following (though see Section 15 for the vectorized implementation of `normal_log`).

```
var lp = 0;
for (int n = 0; n < N; ++n)
  lp += normal_log(y[n], x[n] * beta, sigma);
```

This results in an expression tree structured as

$$((\cdots((p_1 + p_2) + p_3) + p_4)\cdots + p_{N-1}) + p_N).$$

This tree has $N - 1$ nodes representing addition, each with two operands, for a total of $N - 1$ virtual function calls to `chain()` requiring two addition operations each, for a total of $2(N - 1)$ addition operations.

It is a much more efficient use of resources (time and memory) to produce a single node with N operands, requiring only a single virtual function call involving a total of N addition operations. This can be achieved with the following variable implementation class.

```
struct sum_v_vari : public vari {
  vari** v_;
  size_t length_;

  sum_v_vari(const vector<var>& v)
  : vari(var_sum(v)),
    v_(ChainableStack::memalloc_.alloc_array<vari*>(v.size())),
    length_(v.size())
  {
    for (size_t i = 0; i < length_; i++)
      v_[i] = v(i).val();
  }

  void chain() {
    for (size_t i = 0; i < N_; i++)
      v_[i]->adj_ += adj_;
  }
};
```

For simplicitly, only the constructor for a standard vector of autodiff variables is shown. This class provides more general constructors and is used throughout the Stan Math Library.

The `var_sum` function sums the values of the variables in the vector.

```
inline static double
var_sum(const vector<var>& v) {
  double result = 0;
  for (size_t i = 0; i < v.size(); ++i)
```

```
    result += v[i].val();
  return result;
}
```

This sum is then passed to the superclass constructor, which stores the value (and the adjoint, which it initializes to zero).

The following member template function of `stack_alloc` allows arrays of any type to be allocated within the memory arena for automatic differentiation.

```
template <typename T>
inline T* alloc_array(size_t n) {
  return static_cast<T*>(alloc(n * sizeof(T)));
}
```

The `alloc()` method does the allocation of the bytes in the arena, taking care to align them on an 8-byte boundary.

With the implementation in place, the sum function for vectors is trivial.

```
inline var sum(const vector<var>& x) {
  return var(new sum_v_vari(x));
}
```

More efficient special cases can be added for inputs of size zero, one and two, if these are common.

```
  if (x.size() == 0) return 0.0;
  if (x.size() == 1) return x[0];
  if (x.size() == 2) return x[0] + x[1];
```

To deal with running sums in the context of log density definitions, the intermediate values are pushed onto a `vector<var>` and then the `sum()` function is used to return their value. The linear regression density would then be implemented as follows.

```
vector<var> acc;
for (int n = 0; n < y.size(); ++n)
  acc.push_back(normal_log(y[n], x[n] * beta, sigma));
var lp = sum(acc);
```

11. Matrix Data Structures and Arithmetic

The Stan Math Library uses the Eigen C++ Library for storing basic matrix data structures, performing matrix arithmetic, and carrying out linear algebra operations such as factoring and calculating determinants.

11.1. Matrix Data Structures

Matrix and vector types are containers of univariate autodiff variables, i.e., instances of `var`. That is, the design is intrinsically organized around univariate automatic differentiation variables. This is not required by the `chainable` base class, but only `vari` instances are used.

The Stan Math Library uses the following three Eigen types.

Type	*Description*
`Matrix<T, Dynamic, Dynamic>`	matrix
`Matrix<T, Dynamic, 1>`	column vector
`Matrix<T, 1, Dynamic>`	row vector

The template parameter `T` indicates the type of elements in the matrix or vector. Here, the template parameter will be instantiated to `double` for constants or `stan::math::var` for automatic differentiation through the matrix or vector.

11.2. Type Inference

The arithmetic operations are typed, which allows inference of the result type. For example, multiplying two matrices produces a matrix, multiplying a matrix by a column vector produces a column vector, and multiplying a row matrix by a matrix produces a row matrix. As another example, multiplying a row vector by a column vector produces a scalar. Two matrices can be added, as can two column vectors or two row vectors. Adding a scalar to a matrix (vector) is defined to add the scalar to each element of the matrix (vector).

11.3. Dot Products

For example, multiplying a row vector by a column vector of the same size produces a scalar; if they are not the same size, an exception is raised. If both arguments are `double`, Eigen matrix multiplication is used to produce the result.

The naive approach to dot products is to just evaluate the product and sums. With two N-vectors x and y, the resulting expression introduces $2N-1$ nodes (N multiplication, $N-1$ addition), one for each operator in

$$(x_1 \times y_1) + (x_2 \times y_2) + \cdots + (x_N \times y_N)\,.$$

Matrix arithmetic functions such as multiplication essentially follow the design of the variadic functions as discussed in Section 10. A specialized `vari` extension is used to cut the number of nodes allocated from $2N$ to 1 and cut the number of edges traversed during derivative propagation from $4N$ to $2N$.

The following function definition applies to multiplying a row vector by a column vector; there is also a `dot_product` operation that applies to row or column vectors in either position.

```
var multiply(const Matrix<var, 1, Dynamic>& v1,
             const Matrix<var, Dynamic, 1>& v2) {
  return var(new dot_product_vari(v1, v2));
}
```

The following is a simplified version of the actual `dot_product_vari` class; the library implementation includes extra methods for other uses and is highly templated for flexibility).

```
struct dot_product_vari : public vari {
  vari** v1_;    vari** v2_;
  size_t size_;

  dot_product_vari(vari** v1, vari** v2, size_t size)
  : vari(dot(v1,v2,size)), v1_(v1), v2_(v2), size_(size) { }

  dot_product_vari(const Matrix<var, 1, Dynamic>& v1,
                   const Matrix<var, Dynamic, 1>& v2)
  : vari(dot(v1,v2)),
    v1_(static_cast<vari**>(operator new(sizeof(vari*) * size))),
    v2_(static_cast<vari**>(operator new(sizeof(vari*) * size))),
    size_(v1.size()) {
      for (size_t n = 0; n < size_; ++n) {
        v1_[n] = v1[n].vi_;
        v2_[n] = v2[n].vi_;
      }
  }

  void chain() {
    for (size_t n = 0; n < size_; ++n) {
      v1_[n]->adj_ += adj_ * v2_[n]->val_;
      v2_[n]->adj_ += adj_ * v1_[n]->val_;
    }
  }
};
```

The first constructor, for arrays of `vari*`, will be used later for matrix multiplication. The second constructor, for vectors of `var`, is what is used in the function definition above. Both the constructors compute the result using a call to an overloaded function, neither definition of which is shown. The arena-based `operator new` is used to allocate the member arrays `v1_` and `v2_`, then the body of the constructor populates them with the implementation pointers of the input operands. The chain rule is also straightforward because the partial with respect to each operand component is the corresponding component of the other operand vector.

Mixed Type Operations

Taking the dot product of a vector with `double` entries and one with `var` entries cannot be done directly using the `Eigen` multiplication operator because it only applies to inputs with identical entry types.

A very naive approach would be to promote the `double` entries to `var` and then multiply two `var` matrices. This introduces N unnecessary nodes into the expression graphs and involves N unnecessary gradient propagation steps to constants. A slightly less naive approach would be to write the loop directly, but that is problematic for the same reason as using a loop directly for two vectors of `var` entries. Instead, the Stan Math Library introduces a custom `var` vector times `double` vector dot product function that introduces a single node into the expression graph and only requires N propagations for the dot product of N-vectors.

11.4. Matrix Multiplication

To multiply two matrices, a matrix is created for the result and each entry is populated with the appropriate product of a row vector and column vector. A naive implementation for two matrices both composed of `var` entries could just use Eigen's built-in matrix multiplication, but this has the same problem as dot products, only N^2-fold.

To avoid introducing unnecessary nodes into the expression graph, an array of `vari*` is allocated in the autodiff memory arena and populated with the corresponding implementation pointers from the operand matrix rows or columns. For the result matrix, an instance of `dot_product_vari` is constructed using the corresponding row and column variable implementation pointer arrays. The resulting storage is much more economical than if a separate dot product were created for each entry; for instance, in multiplying a pair of $N \times N$ matrices, separate operand storage for each result entry would require a total of $\mathcal{O}(N^3)$ memory, whereas the current scheme requires only $\mathcal{O}(N^2)$ memory.

11.5. Specialized Matrix Multiplication

The Stan Math Library provides a custom `multiply_self_transpose` function because the memory required can be cut in half and the speed improved compared to applying the transposition and then the general matrix multiplication function. The same vectors make up the original matrix's rows and the transposed version's columns, so the same array of `vari**` can be reused.

Reducing memory usage often has the pleasant side effect of increasing speed. In the case of multiplying a matrix by its own transpose, fewer copy operations need to be performed. In general, memory locality of algorithms will also be improved because of less need to bring copies into memory.

If the matrix is lower triangular, only the non-zero portions of the matrix need to be stored and traversed and saved in the expression graph. Multiplying

a lower-triangular matrix by its own transposition is common in multivariate statistics, where it is used to reconstruct a positive definite matrix Ω, such as a correlation, covariance, or precision matrix, from its Cholesky factor, $\Omega = L\,L^\top$. For efficiency in both time and memory, the Stan Math Library provides a built-in `multiply_self_transpose_lower_tri` function.

12. Linear Algebra

Matrix arithmetic is just repeated standard arithmetic. Linear algebra operations such as calculating determinants or matrix divisions are much more complicated in their internal structure. In both cases, specialized derivatives can be much more efficient than just automatically differentiating the library functions.

Giles (2008) provides a rigorous definition of matrix automatic differentiation and a translation of some of the key matrix derivative results (Petersen and Pedersen, 2012; Magnus and Neudecker, 2007) to automatic differentiation terms.

12.1. Log Determinants

Many probability formulas require the logarithm of the absolute value of the determinant of a matrix; it shows up in change of variables problems and as the normalizing term of the multivariate normal density. If x is an $N \times N$ matrix, then the partials of the log absolute determinant are given by

$$\frac{\partial}{\partial x} \log |\det(x)| = \left(x^{-1}\right)^\top.$$

On an element by element basis, this reduces to

$$\frac{\partial}{\partial x_{m,n}} \log |\det(x)| = \left(\left(x^{-1}\right)^\top\right)_{m,n}.$$

Precomputed Gradients Implementation

In some cases, such as log determinants, it is easier to compute partial eagerly at the same time as the function value. The following utility variable implementation will be used to store vector and matrix derivatives with precomputed gradients.

```
struct precomputed_gradients_vari : public vari {
  const size_t size_;
  vari** varis_;
  double* gradients_;

  precomputed_gradients_vari(double val, size_t size,
```

```
                                vari** varis, double* gradients)
        : vari(val), size_(size),
          varis_(varis), gradients_(gradients) { }

  void chain() {
    for (size_t i = 0; i < size_; ++i)
      varis_[i]->adj_ += adj_ * gradients_[i];
  }
};
```

In addition to the value and adjoint stored by the parent class (`vari`), this class adds a parallel array of operands and gradients along with their size. The `chain()` method adds the adjoint times the stored partial to the operands' adjoint.

Log Determinant Implementation

The implementation of derivatives for the log determinant involves performing a Householder QR decomposition of the double values of the matrix, at which point the log absolute determinant and inverse can easily be extracted.

```
template <int R, int C>
inline var
log_determinant(const Eigen::Matrix<var,R,C>& m) {
  Matrix<double,R,C> m_d(m.rows(),m.cols());
  for (int i = 0; i < m.size(); ++i)
    m_d(i) = m(i).val();

  Eigen::FullPivHouseholderQR<Matrix<double,R,C> > hh
    = m_d.fullPivHouseholderQr();

  double val = hh.logAbsDeterminant();

  vari** operands = ChainableStack::memalloc_
                  .alloc_array<vari*>(m.size());
  for (int i = 0; i < m.size(); ++i)
    operands[i] = m(i).vi_;

  Matrix<double,R,C> m_inv_transpose
    = hh.inverse().transpose();
  double* gradients = ChainableStack::memalloc_
                        .alloc_array<double>(m.size());
  for (int i = 0; i < m.size(); ++i)
    gradients[i] = m_inv_transpose(i);

  return var(new precomputed_gradients_vari(
                  val,m.size(),operands,gradients));
}
```

First, the values of the `var` matrix are extracted and used to set the values in the `double` matrix `m_d`. Then the decomposition is performed using Eigen's `FullPivHouseHholderQR` class, which according to its documentation provides very good numerical stability (Golub and Van Loan, 1996, Section 5.1). Next, the value of the log absolute determinant is extracted from the decomposition class. Then the operands are set to the argument matrix's variable implementations to be used in the reverse pass of automatic differentiation. Next, the inverse is transposed and used to populate the double array `gradients`. The final step is to allocate (on the arena) a new precomputed gradients variable implementation (see the previous section for definitions); the precomputed-gradients implementation is then wrapped in a `var` for return.

The memory for the gradients and the operands is allocated using the arena-based allocator `memalloc_`. The precomputed gradients structure cannot store standard vectors or Eigen vectors directly because they manage their own memory and would not be properly deleted.[3]

The forward pass requires the value of the log determinant. It would be possible to be lazy and reconstruct the input matrix from the operands and then calculate its inverse in the reverse pass. This would save a matrix of double values in storage, but would require an additional QR decomposition.

Critically for speed, all matrix calculations are carried out directly on `double` values rather than on `var` values. The primary motivation is to reduce the size of the expression graph and subsequent time spent propagating derivatives in the reverse pass. A pleasant side effect is that the matrix operations are carried out with all of Eigen's optimizations in effect; many of these are only possible with the memory locality provided by `double` values.[4]

Specialized Log Determinant Implementation

The Stan Math Library provides a specialized log determinant implementation for symmetric, positive-definite matrices, which are commonly used as metrics for geometrical applications or as correlation, covariance or precision matrices for statistical applications. Symmetric, positive-definite matrices can be Cholesky factored using the $\text{LDL}^\top$ algorithm (Golub and Van Loan, 1996, Chapter 4), which Eigen provides through its `LDLT` class. Like the QR decomposition, after the decomposition is performed, extracting the log determinant and inverse is efficient.

[3] The Stan Math Library memory manager provides an additional stack to store pointers to `chainable` implementations that need to have their destructors called.

[4] Fog (2014) provides an excellent overview of memory locality, parallelization, and branch prediction, which is very useful for those wishing to optimize C++ code.

13. Differential Equation Solver

The Stan Math Library provides a differential equation solver and allows derivatives of solutions to be calculated with respect to parameters and/or initial values. The system state is coded as a `std::vector` and the system of equations coded as a functor.

13.1. Systems of Ordinary Differential Equations

Systems of differential equations describe the evolution of a multidimensional system over time from a given starting point.

The state of an N-dimensional system at time t will be represented by a vector $y(t) \in \mathbb{R}^N$. The change in the system is determined by a function $f(y, \theta, t)$ that returns the change in state given current position y, parameter vector $\theta \in \mathbb{R}^K$, and time $t \in \mathbb{R}$,

$$\frac{\mathrm{d}}{\mathrm{d}t} y = f(y, \theta, t).$$

The notation $f_n(y, \theta, t)$ will be used for the n-th component of $f(y, \theta, t)$, so that the change in y_n can be given by

$$\frac{\mathrm{d}}{\mathrm{d}t} y_n = f_n(y, \theta, t).$$

Given an initial position

$$\xi = y(0) \in \mathbb{R}^N$$

at initial time $t = 0$, parameter values θ, and system function f, it is possible to solve for state position $y(t)$ at other times t using numerical methods.[5] The Stan Math Library also supports initial positions given at times other than 0, which can be reduced to systems with initial time 0 with offsets. (Press et al., 2007, Chapter 17) provides an overview of ODE solvers.

Example: Simple Harmonic Oscillator

The simple harmonic osillator in two dimensions ($N = 2$) with a single parameter θ is given by the state equations

$$\frac{\mathrm{d}}{\mathrm{d}t} y_1 = y_2 \quad \text{and} \quad \frac{\mathrm{d}}{\mathrm{d}t} y_2 = -\theta y_1.$$

which can be written as a state function as

$$f(y, \theta, t) = [y_2 \quad -\theta y_1].$$

[5] It is technically possible to solve systems backward in time for $t < t_0$, but the Stan Math Library is currently restricted to forward solutions; signs can be reversed to code backward time evolution.

13.2. Sensitivities of Solution to Inputs

Solutions of differential equations are often evaluated for their sensitivity to variation in parameters, i.e.,

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

where θ is a vector of parameters, initial state, or both. This is useful for applications in optimization or parameter estimation and in reporting on (co)variation with respect to parameters.

13.3. Differentiating the Integrator

One way to compute sensitivities is to automatically differentiate the numerical integrator. This can be done with `var` instances with a suitably templated integrator, such as Boost's odeint (Ahnert and Mulansky, 2014). For example, (Weber et al., 2014) used the Dormand-Prince integrator from odeint, a fifth-order Runge-Kutta method. Because odeint provides only a single template parameter, all input variables (time, initial state, system parameters) were promoted to `var`.

Although possible, differentiating the integrator is inadvisable for two reasons. First, it generates a large expression graph which consumes both memory and time during automatic differentiation. This problem is exacerbated by the overpromotion required by limited templating. Second, there is no way to control the error in the sensitivity calculations.

13.4. Coupled System

Both of the problems faced when attempting to differentiate the integrator can be solved by creating a coupled system that adds state variables for the sensitivities to the original system. The partial derivatives are then given by the solutions for the sensitivities.

For each state variable n and each parameter or initial state α_m for which sensitivities are required, a newly defined state variable

$$z_{n,m} = \frac{\partial}{\partial \alpha_m} y_m$$

is added to the system. This produces coupled systems of the following sizes, based on whether derivatives are required with respect to the initial state, system parameters, or both.

Sensitivities		
Initial State	*Parameters*	*Coupled Size*
+	-	$N \times (N + 1)$
-	+	$N \times (K + 1)$
+	+	$N \times (N + K + 1)$

With the coupled system, sensitivities are calculated by the integrator step by step and thus can be controlled for error. The resulting sensitivities at solution times can then be used to construct an autodiff variable implementation of the output expression with respect to its inputs (initial state and/or parameters).

The Stan Math Library currently uses the Dormand-Prince integrator implementation from Boost's odeint library (Ahnert and Mulansky, 2014), but the design is modular, with dependencies only on solving integration problems using `double` values. More robust integrators, in particular the CVODE integrator (Cohen and Hindmarsh, 1996) from the SUNDIALS package (Hindmarsh et al., 2005), will be useful for their ability to automatically deal with stiff systems of equations.

13.5. Differentiating an ODE Solution with Respect to a Parameter

The sensitivity matrix for states (y) by parameters (θ) is

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

Componentwise, that is

$$z_{n,m} = \frac{\partial}{\partial \theta_m} y_n.$$

The time derivatives of z needed to complete the coupled system can be given via a function h, defined and derived componentwise as

$$\begin{aligned}
h_{n,m}(y, z, \theta, t) &= \frac{\mathrm{d}}{\mathrm{d}t} z_{n,m}. \\
&= \frac{\mathrm{d}}{\mathrm{d}t} \frac{\mathrm{d}}{\mathrm{d}\theta_m} y_n \\
&= \frac{\mathrm{d}}{\mathrm{d}\theta_m} \frac{\mathrm{d}}{\mathrm{d}t} y_n \\
&= \frac{\mathrm{d}}{\mathrm{d}\theta_m} f_n(y, \theta) \\
&= \frac{\partial}{\partial \theta_m} f_n(y, \theta) + \sum_{j=1}^{N} \left(\frac{\partial}{\partial \theta_m} y_j \right) \frac{\partial}{\partial y_j} f_n(y, \alpha) \\
&= \frac{\partial}{\partial \theta_k} f_n(y, \theta) + \sum_{j=1}^{N} z_{j,m} \frac{\partial}{\partial y_j} f_n(y, \theta).
\end{aligned}$$

The sensitivity of a parameter is the derivative of the state of the system with respect to that parameter, with all other parameters held constant (but not the states). Thus the sensitivities act as partial derivatives with respect to the parameters but total derivatives with respect to the states, because of the need to take into account the change in solution as parameters change.

The coupled system will also need new initial states $\chi_{n,m}$ for the sensitivities $z_{n,m}$, all of which work out to be zero, because

$$\chi_{n,m} = \frac{\partial}{\partial \theta_m} \xi_n = 0.$$

The final system that couples the original state with sensitivities with respect to parameters has state (y, z), initial conditions ξ, χ, and system function (f, h).

13.6. Differentiating an ODE Solution with Respect to the Initial State

The next form of coupling will be of initial states, with new state variables

$$w = \frac{\partial}{\partial \xi} y,$$

which works out componentwise to

$$w_{n,k} = \frac{\partial}{\partial \xi_k} y_n.$$

Sensitivities can be worked out in this case by defining a new system with state variables offset by the initial condition,

$$u = y - \xi.$$

This defines a new system function g with the original parameters θ and original initial state ξ now both treated as parameters,

$$g(u, (\theta, \xi)) = f(u + \xi, \theta).$$

The new initial state is a zero vector by construction. The derivatives are now with respect to the parameters of the revised system with state u, system

function g, and parameters θ, ξ, and work out to

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t} w_{n,k} &= \frac{\mathrm{d}}{\mathrm{d}t} \frac{\mathrm{d}}{\mathrm{d}\xi_k} y_n \\
&= \frac{\mathrm{d}}{\mathrm{d}\xi_k} \frac{\mathrm{d}}{\mathrm{d}t} y_n \\
&= \frac{\mathrm{d}}{\mathrm{d}\xi_k} f_n(y, \theta) \\
&= \frac{\mathrm{d}}{\mathrm{d}\xi_k} f_n(u + \xi, \theta) \\
&= \frac{\partial}{\partial \xi_k} f_n(u + \xi, \theta) \\
&\quad + \sum_{j=1}^{N} \left(\frac{\partial}{\partial \xi_k} u_j \right) \frac{\partial}{\partial u_j} f_n(u + \xi, \theta) \\
&\quad + \sum_{j=1}^{N} \left(\frac{\partial}{\partial \xi_k} \xi_j \right) \frac{\partial}{\partial \xi_j} f_n(u + \xi, \theta) \\
&= \frac{\partial}{\partial \xi_k} f_n(u + \xi, \theta) \\
&\quad + \sum_{j=1}^{N} \left(u_j + \frac{\partial}{\partial \xi_k} \xi_j \right) \frac{\partial}{\partial y_j} f_n(y, \theta).
\end{aligned}
$$

The derivative $\partial \xi_j / \partial \xi_k$ on the last line is equal to 1 if $j = k$ and equal to 0 otherwise.

13.7. Computing Sensitivities with Nested Automatic Differentiation

In order to add sensitivities with respect to parameters and/or initial states to the system, Jacobians of the differential equation system function f are required with respect to the parameters (where the parameters may include the initial states, as shown in the last section).

To allow nested evaluation of system Jacobians, the Stan Math Library allows nested derivative evaluations. When the system derivatives are required to solve the system of ODEs, the current stack location is recorded, autodiff of the system takes place on the top of the stack, and then derivative propagation stops at the recorded stack location. This allows arbitrary reverse-mode automatic differentiation to be nested inside other reverse-mode calculations. The top of the stack can even be reused for Jacobian calculations without rebuilding an expression graph for the outer system being differentiated.

13.8. Putting Results Back Together

When the numerical integrator solves the coupled system, the solution is for the original state variables y along with sensitivities $\frac{\mathrm{d}}{\mathrm{d}\theta}y$ and/or $\frac{\mathrm{d}}{\mathrm{d}\xi}y$.

Given a number of solution times requested, $t_1, \ldots, t_J$, the state solutions and sensitivity solutions are used to create a `vari` instance for each $y_n(t_j)$. Each of these variables is then connected via its sensitivity to each of the input parameters and/or initial states. This requires storing the sensitivities as part of the result variables in a general (not ODE specific) precomputed-gradients `vari` structure.

13.9. System Function

Given an initial condition and a set of requested solution times, the Stan Math Library can integrate an ODE defined in terms of a system function. The system function must be able to be instantiated by the following signature (where `vector` is `std::vector`)

```
vector<var>
operator()(double t,
           const vector<double>& y,
           const vector<var>& theta) const;
```

The return value is the vector of time derivatives evaluated at time `t` and system state `y`, given parameters `theta`, continuous and integer data `x` and `x_int`, and an output stream `o` for messages. The function must be constant.

The simple harmonic oscillator could be implemented as the following functor.

```
struct sho {
  template <typename T>
  vector<T> operator()(double t,
                       const vector<double>& y
                       const vector<var>& theta) const
    vector<T> dy_dt(2);
    dy_dt[0] = y[1];
    dy_dt[1] = -theta[0] * y[2];
    return dy_dt;
  }
};
```

13.10. Integration Function

The function provided by the Stan Math Library to compute solutions to ODEs and support sensitivity calculations through autodiff has the following signature.

```
template <typename F, typename T1, typename T2>
vector<vector<typename promote_args<T1, T2>::type> >
integrate_ode(const F& f,
              const std::vector<T1> y0,
              const double t0,
              const std::vector<double>& ts,
              const std::vector<T2>& theta
              const std::vector<double>& x,
              const std::vector<int>& x_int,
              std::ostream* msgs);
```

The argument `f` is the system function, and it must be implemented with enough generality to allow autodiff with respect to the parameters and/or initial state. The variable `y0` is the initial state and `t0` is the initial time (which may be different than 0); its scalar type parameter `T1` may be either `double` or `var`, with `var` being used to autodiff with respect to the initial state. The vector `ts` is a set of solution times requested and must have elements greater than `t0` arranged in strictly ascending order. The vector `theta` is for system parameters; its scalar type parameter `T2` can be either `double` or `var`, with `var` used to autodiff with respect to parameters. There are two additional arguments, `x` and `x_int`, used for real and double-valued data respectively. These are provided so that data does not need to be either hard coded in the system or promoted to `var` (which would add unnecessary components to the expression graph, increasing time and memory used). Finally, an output stream pointer can be provided for messages printed by the integrator.

Given the harmonic oscillator class `sho`, and initial values $(-1, 0)$, the solutions with sensitivities for the harmonic oscillator at times 1:10 with parameter $\theta = 0.35$, initial state $y(0) = (-1, 0)$, can be obtained as follows.

```
double t0 = 0.0;
vector<double> ts;
for (int t = 1; t <= 10; ++t) ts.push_back(t);
var theta = 0.35;
vector<var> y0(2);  y[0] = 0;  y[1] = -1.0;
vector<var> ys
  = integrate_ode(sho(), y0, t0, ts, theta);
for (int i = 0; i < 2; ++i) {
  if (i > 0) set_zero_all_adjoints();
  y.grad();
  for (int n = 0; n <= ys.size(); ++n)
    printf("y(%2d,%2d) = %5.2f\n", ts[n], i, ts[n][i])
    printf("sens: theta=%5.2f y0(0)=%5.2f y0(1)=%5.2f\n",
           theta.adj(), y0[0].adj(), y0[1].adj());
  }
}
```

14. Probability Functions and Traits

The primary application for which the Stan Math Library was developed is fitting statistical models, with optimization for maximum likelihood estimates and Markov chain Monte Carlo (MCMC) for Bayesian posterior sampling (Gelman et al., 2013). For optimization, gradient descent and quasi-Newton methods both depend on being able to calculate derivatives of the objective function (in the primary application, a (penalized) log likelihood or Bayesian posterior). For MCMC, Hamiltonian Monte Carlo (HMC) (Duane et al., 1987; Neal, 2011; Betancourt et al., 2014) requires gradients in the leapfrog integrator it uses to solve the Hamiltonian equations and thus simulate an efficient flow through the posterior.

In both of these applications, probability functions need only be calculated up to a constant term. In order to do this efficiently and automatically, the Stan Math Library uses traits-based metaprogramming and template parameters to configure its log probability functions. For example, consider the log normal density again

$$\log \mathsf{Normal}(y|\mu,\sigma) = -\log(2\pi) - \log\sigma - \frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2.$$

In all cases, the first term, $-\log(2\pi)$ can be dropped because it's constant. If σ is a constant, then the second term can be dropped as well. The final term must be preserved unless all of the arguments, y, μ, and σ, are constants.

For maximum generality, each log probability function has a leading boolean template argument `propto`, which will have a true value if the calculation is being done up to a proportion and a false value otherwise. Calculations up to a proportion drop constant terms. Each of the remaining arguments is templated out separately, to allow all eight combinations of constants and variables as arguments. The return type is the promotion of the argument types. The fully specified signature is

```
template <bool propto, typename T1, typename T2, typename T3>
typename return_type<T1, T2, T3>::type
normal_log(const T1& y, const T2& mu, const T3& sigma) {
  ...
```

For scalar arguments, the `return_type` traits metaprogram has the same behavior as Boost's `promote_args`; generalizations to container arguments such as vectors and arrays will be discussed in Section 15.

Without going into the details of error handling, the body of the function uses a further metaprogram to decide which terms to include in the result. Not considering generalizations to containers and analytic gradients, the body of the function behaves as follows:

```
include stan::math::square;
include std::log;
```

```
  typename return_type<T1, T2, T3>::type lp = 0;
  if (include_summand<propto>::value)
    lp += NEGATIVE_LOG_2_PI;
  if (include_summand<propto,T3>::value)
    lp -= log(sigma);
  if (include_summand<propto, T1, T2, T3>::value)
    lp -= 0.5 * square((y - mu) / sigma);
  return lp;
}
```

The `include_summand` traits metaprogram is quite simple, defining an enum with a true value if any of the types is not primitive (integer or floating point). Although it allows up to 10 arguments and supports container types, the following implementation of `include_summand` suffices for the current example.

```
template <boolean propto,
          typename T1 = double,
          typename T2 = double,
          typename T3 = double>
struct include_summand {
  enum { value  = !propto
                  || !is_constant<T1>::value
                  || !is_constant<T2>::value
                  || !is_constant<T3>::value;  }
};
```

The template specification requires a boolean template parameter `propto`, which is true if calculations are to be done up to a constant multiplier. The remaining template typename parameters have default values of `double`, which are the base return value type. The structure defines a single enum `value`, the value of which will be true if the `propto` is false or if any of the remaining template parameters are not constant values. For scalar types, `is_constant` behaves like Boost's `is_arithmetic` trait metaprogram (from the TypeTraits library); it is extended for container types as described in Section 15.

15. Vectorization of Reductions

The Stan Math Library was developed to facilitate probabilistic modeling and inference by making it easy to define density functions on the log scale with derivative support. A common use case is modeling a sequence of independent observations $y_1, \ldots, y_N$ from some parametric likelihood function $p(y_n|\theta)$ with shared parameters θ. In this case, the joint likelihood function for all the observations is

$$p(y_1, \ldots, y_N|\theta) = \prod_{n=1}^{N} p(y_n|\theta),$$

or on the log scale,

$$\log p(y_1, \ldots, y_N|\theta) = \sum_{n=1}^{N} \log p(y_n|\theta).$$

Thus the log joint likelihood is the sum of the log likelihoods for the observations y_n.

15.1. Argument Broadcasting

The normal log density function takes three arguments, the variate or outcome along with a location (mean) and scale (standard deviation) parameter. Without vectorization, each of these arguments could be a scalar of type `int`, `double`, or `stan::math::var`.

With vectorization, each argument may also be a standard library vector, `vector<T>`, an Eigen matrix, `Matrix<T, Dynamic, 1>`, or an Eigen row matrix, `Matrix<T, 1, Dynamic>`, where `T` is any of the scalar types. All container arguments must be the same size, and any scalar arguments are broadcast to behave as if they were containers with the scalar in every position.

15.2. Vector Views

Rather than implement 27 different versions of the normal log density, the Stan Math Library introduces an expression template allowing any of the types to be treated as a vector-like container holding its contents. The view provides a scalar typedef, a size, and an operator to access elements using bracket notation. Fundamentally, it stores a pointer, which will in practice be to a single element or to an array. Simplifying a bit, the code is as follows.

```
template <typename T,
          bool is_array = stan::is_vector_like<T>::value>
class VectorView {
public:
  typedef typename scalar_type<T>::type scalar_t;

  VectorView(scalar_t& c) : x_(&c) { }

  VectorView(std::vector<scalar_t>& v) : x_(&v[0]) { }

  template <int R, int C>
  VectorView(Eigen::Matrix<scalar_t,R,C>& m) : x_(&m(0)) { }

  VectorView(scalar_t* x) : x_(x) { }

  scalar_t& operator[](int i) {
    if (is_array) return x_[i];
```

```
    else return x_[0];
  }
private:
  scalar_t* x_;
};
```

There are two template parameters, `T` being the type of object being viewed and `is_array` being true if the object being viewed is to be treated like a vector when accessing members. The second argument can always be set by default; it acts as a typedef to enable a simpler definition of `operator[]`. The traits metaprogram `is_vector_like<T>::value` determines if the viewed type `T` acts like a container.

The typedef `scalar_t` uses the metaprogram `scalar_type<T>::type` to calculate the underlying scalar type for the viewed type `T`.

The constructor takes either a scalar reference, a standard vector, or an Eigen matrix and stores either a pointer to the scalar or the first element of an array in private member variable `x_`.

There is a single member function, `operator[]`, which defines how the view returns elements for a given index `i`. If the viewed type is a container as defined by template parameter `is_array`, then it returns the element at index `i`; if it is a scalar type, then the value is returned (`x_[0]` is equivalent to `*x_`).

Once a view is constructed, its `operator[]` can be used just like any other container. This pattern would be easy to extend to further containers, such as Boost sequences. After this code is executed,

```
vector<var> y(3); ...
VectorView<vector<var> > y_vw(y);
```

the value of `y_vw[i]` tracks `y[i]`, and the typedef `y_vw::scalar_t` is `var`.

For a scalar, the construction is similar. After this code is executed,

```
double z;
VectorView<double> z_vw(z);
```

the value of `z_vw[i]` tracks the value of `z` no matter what `i` is—the index is simply ignored.

No copy is made of the contents of `y` and it is up to the constructor of `VectorView` to ensure that the contents of `y` does not change, for instance by resizing. Furthermore, a non-constant reference is returned by the `operator[]`, meaning that clients of the view can change the contained object that is being viewed. This is all standard in such a view pattern, examples of which are the `block`, `head`, and `tail` functions in Eigen, which provide mutable views of matrices or vectors.

15.3. Value Extractor

The helper function `value_of` is defined to extract the `double`-based value of either an autodiff variable or a primitive value. The definition for primitive types is in the `stan::math` namespace.

```
template <typename T>
inline double value_of(const T x) {
  return static_cast<double>(x);
}
```

The function is overloaded for autodiff variables.

```
inline double value_of(const var& v) {
  return v.vi_->val_;
}
```

The definition for autodiff variables is in the `stan::math` namespace to allow argument-dependent lookup.

It is crucial for many of the templated definitions to be able to pull out `double` values no matter what the type of argument is. The function `value_of` allows

```
T x;
double x_d = value_of(x);
```

with `T` instantiated to `stan::math::var`, `double`, or `int`.

15.4. Vector Builder Generic Container

By itself, the vector view does not provide a way to generate intermediate quantities needed in calculations and store them in an efficient way. For the normal density, if σ is a single value, then $\log \sigma$ and σ^{-1} can be computed once and reused.

The class `VectorBuilder` is used as a container for intermediate values. In the following code, from the normal density, it is used as the type of intermediate containers for σ^{-1} and $\log \sigma$.[6]

```
VectorBuilder<true> inv_sigma(length(sigma));

VectorBuilder<include_summand<propto,T_scale>::value>
  log_sigma(length(sigma));
```

[6]Stan's version of these functions is slightly more complicated in that they also support forward-mode autodiff. The details of return type manipulation for forward-mode is sidestepped here by dropping template parameters and their helper template metaprograms and fixing types as `double`.

The template parameter indicates whether the variable needs to be stored or not. If the parameter is false, no storage will be allocated. The intermediate value `inv_sigma` is always computed because it will be needed for the normal density even if no derivatives are taken. The intermediate value `log_sigma`, on the other hand, is only computed and memory is only allocated for it if a term only involving a type `T_scale` is included; see Section 14 for the definition of `include_summand`.

The code for `VectorBuilder` itself is relatively straightforward

```
template <bool used, typename T1,
          typename T2=double, typename T3=double>
struct VectorBuilder {
  VectorBuilderHelper<used, contains_vector<T1, T2, T3>::value> a_;

  VectorBuilder(size_t n) : a_(n) { }

  T1& operator[](size_t i) { return a_[i]; }
};
```

The template struct `VectorBuilderHelper` provides the type of the value `a`; it is passed a boolean template parameter indicating whether any of the types is a vector and thus requires a vector rather than scalar to be allocated for storage. The constructor for `VectorBuilder` just passes the requested size to the helper's constructor. This will either construct a container of the requested size or return a dummy, depending on the value of the template parameter `used`. The definition of `operator[]` just returns the value at index `i` produced by the helper.

There are three use cases for efficiency for the vector. The simplest is the dummy, which is employed when `used` is false. It also defines the primary template structure.

```
template <bool used, bool is_vec>
struct VectorBuilderHelper {
  VectorBuilderHelper(size_t /* n */) { }

  double& operator[](size_t /* i */) {
    throw std::logic_error("used is false.");
  }
};
```

The two template parameters are booleans indicating whether the storage is used, and if it is used, whether it is a vector or not. For the base case, there is no allocation. Because there are no virtual functions declared, the size of `VectorBuilderHelper` is zero bytes and it can be optimized out of the definition of `VectorBuilder`. The definition of the operator raises a standard library logic error because it should never be called; it is only defined because it is required to be so that it can be used in alternation with helper implementations that do return values.

For the case where `used` is true but `is_vec` is false, the following specialization stores a `double` value.

```
template <>
struct VectorBuilderHelper<true,false> {
  double x_;

  VectorBuilderHelper(size_t /* n */) : x_(0.0) { }

  T1& operator[](size_t /* i */) {
    return x_;
  }
};
```

Note that like `VectorView`, it returns the value `x_` no matter what index is provided.

For the case where `used` is true and `is_vec` is true, a standard library vector is used to store values, allocating the memory in the constructor.

```
template <>
struct VectorBuilderHelper<true,true> {
  std::vector<T1> x_;

  VectorBuilderHelper(size_t n) : x_(n) { }

  T1& operator[](size_t i) {
    return x_[i];
  }
};
```

Instances only live on the C++ stack, so the memory is managed implicitly when instances of `VectorBuilder` go out of scope.

15.5. Operands and Partials

The last piece of the puzzle in defining flexible functions for automatic differentiation is a general purpose structure `operands_and_partials`. The key to this structure is that it stores the operands to a function or operator as an array of `vari` and stores the partial of the result with respect to the operand in a parallel array of `double` values. That is, it allows general eager programming of partials, which allows straightforward metaprograms in many contexts such as vectorized density functions. A simplified definition is as follows.[7]

[7] As with other aspects of the probability functions, the implementation in Stan is more general, allowing for forward-mode autodiff variables of varying order as well as reverse-mode autodiff variables and primitives. It also allows more or fewer template parameters, giving all but the first default `double` values.

```
template <typename T1, typename T2, typename T3>
struct OperandsAndPartials {
  size_t n_;
  vari** ops_;
  double* partials_;
  VectorView<is_vector<T1>::value,
             is_constant_struct<T1>::value> d_x1_;
  VectorView<is_vector<T2>::value,
             is_constant_struct<T2>::value> d_x2_;
  VectorView<is_vector<T3>::value,
             is_constant_struct<T3>::value> d_x3_;
  ...
};
```

The vector views are views into the `partial_` array. The constructor is as follows.

```
OperandsAndPartials(const T1& x1, const T2& x2, const T3& x3)
: n_(!is_constant_struct<T1>::value * length(x1)
     + !is_constant_struct<T2>::value * length(x2)
     + !is_constant_struct<T3>::value * length(x3)),
  ops_(memalloc_.array_alloc<vari*>(n_)),
  partials_(memalloc_.array_alloc<double>(n_)),
  d_x1_(partials_),
  d_x2_(partials_
        + !is_constant_struct<T1>::value * length(x1)),
  d_x3_(partials_
        + !is_constant_struct<T1>::value * length(x1)),
        + !is_constant_struct<T2>::value * length(x2))
{
  size_t base = 0;
  if (!is_constant_struct<T1>::value)
    base += set_varis<T1>::set(&ops_[base], x1);
  if (!is_constant_struct<T2>::value)
    base += set_varis<T2>::set(&ops_[base], x2);
  if (!is_constant_struct<T3>::value)
    base += set_varis<T3>::set(&ops_[base], x3);

  std::fill(partials_, partials_ + n, 0);
}
```

The class template parameters `T1`, `T2`, and `T3` define the argument types for the constructor; these will be the arguments to a probability function and may be vectors or scalars of either primitive or autodiff type. The size `n_` of the operands and partials arrays is computed by summing the non-constant argument sizes; the function `length` is trivial and not shown. The operands `ops_` and partials `partials_` are allocated in the arena using a template function

that allocates the right size memory and casts it to an array with elements of the specified template type. The first partials view, `d_x1_`, starts at the start of `partials_` itself. Each subsequent view is started by incrementing the start of the last view; this could perhaps be made more efficient by extracting the pointer from the previous view.

The body of the constructor sets the operator values in `ops_` and fills the partials array with zero values. The operands are set using the static function `set` in the template class `set_varis`, which extracts the `vari*` from the arguments.

The member function `to_var` returns a `var` with an implementation of the following class, which stores the operands and partials and uses them for the chain rule.

```
struct partials_vari : public vari {
  const size_t N_;
  vari** operands_;
  double* partials_;

  partials_vari(double value, size_t N,
                vari** operands, double* partials)
  : vari(value), N_(N),
    operands_(operands), partials_(partials) { }

  void chain() {
    for (size_t n = 0; n < N_; ++n)
      operands_[n]->adj_ += adj_ * partials_[n];
  }
};
```

The constructor passes the value to the superclass's constructor, `vari(double)`, then stores the size, operands, and partials. The `chain()` implementation increments each operand's adjoint by the result's adjoint times the partial derivative of the result relative to the operand.

15.6. Example: Vectorization of the Normal Log Density

Continuing with the example of the normal probability density function introduced in Section 14, the pieces are now all in place to see how `normal_log` is defined for reverse-mode autodiff with vectorization.

The function signature is as follows

```
template <bool propto,
          typename T_y, typename T_loc, typename T_scale>
typename return_type<T_y,T_loc,T_scale>::type
normal_log(const T_y& y, const T_loc& mu, const T_scale& sigma);
...
```

The first boolean template parameter will be true if calculations are allowed to drop constant terms. The remaining template parameters are the types of the arguments for the variable and the location and scale parameters. The result is the calculated return type, which will be `var` if any of the arguments is a (container of) `var` and `double` otherwise.

The function begins by validating all of the inputs.

```
static const char* function("stan::prob::normal_log");
check_not_nan(function, "Random variable", y);
check_finite(function, "Location parameter", mu);
check_positive(function, "Scale parameter", sigma);
check_consistent_sizes(function, "Random variable", y,
    "Location parameter", mu, "Scale parameter", sigma);
```

The function name itself is provided as a static constant. These functions raise exceptions with warning messages indicating where the problem arose. The consistent size check ensures that all of the container types (if any) are of the same size. For example, if both `y` and `mu` are vectors and `sigma` is a scalar, then `y` and `mu` must be of the same size.

Next, the function returns zero if any of the container sizes is zero, or if none of the argument types is an autodiff variable and `propto` is true. In either case, the function returns zero.

```
if (!(stan::length(y) && stan::length(mu)
      && stan::length(sigma)))
  return 0.0;

if (!include_summand<propto,T_y,T_loc,T_scale>::value)
  return 0.0;
```

Only then is the accumulator `logp` for the result initialized.

```
double logp = 0;
```

Next, the operands and partials accumulator is initialized along with vector views of each of the arguments. The size `N` is set to the maximum of the argument sizes (scalars are treated as size 1).

```
OperandsAndPartials<T_y, T_loc, T_scale>
  operands_and_partials(y, mu, sigma);

VectorView<const T_y> y_vec(y);
VectorView<const T_loc> mu_vec(mu);
VectorView<const T_scale> sigma_vec(sigma);

size_t N = max_size(y, mu, sigma);
```

Next, the vector builders for σ^{-1} and $\log \sigma$ are constructed and filled.

```
  VectorBuilder<true, T_partials_return, T_scale>
    inv_sigma(length(sigma));
  for (size_t i = 0; i < length(sigma); i++)
    inv_sigma[i] = 1.0 / value_of(sigma_vec[i]);
```

Although `inv_sigma` is always filled, `log_sigma` is not filled if the calculation is being done up to a proportion and the scale parameter is a constant.

```
  VectorBuilder<include_summand<propto,T_scale>::value,
                T_partials_return, T_scale>
    log_sigma(length(sigma));
  if (include_summand<propto,T_scale>::value)
    for (size_t i = 0; i < length(sigma); i++)
      log_sigma[i] = log(value_of(sigma_vec[i]));
```

The value of `length(sigma)` will be 1 if `sigma` is a scalar and the size of the container otherwise. Because the `include_summand` traits metaprogram is evaluated statically, the compiler is smart enough to simply drop this whole loop if the summand should not be included. As a result, exactly as many logarithms and inversions are calculated as necessary. These values are then used in a loop over `N`, the size of the largest argument (1 if they are all scalars).

```
  for (size_t n = 0; n < N; n++) {
    double y_dbl = value_of(y_vec[n]);
    double mu_dbl = value_of(mu_vec[n]);

    double y_minus_mu_over_sigma
      = (y_dbl - mu_dbl) * inv_sigma[n];
    double y_minus_mu_over_sigma_squared
      = y_minus_mu_over_sigma * y_minus_mu_over_sigma;

    if (include_summand<propto>::value)
      logp += NEG_LOG_SQRT_TWO_PI;
    if (include_summand<propto,T_scale>::value)
      logp -= log_sigma[n];
    if (include_summand<propto,T_y,T_loc,T_scale>::value)
      logp += NEGATIVE_HALF * y_minus_mu_over_sigma_squared;

    double scaled_diff = inv_sigma[n] * y_minus_mu_over_sigma;
    if (!is_constant_struct<T_y>::value)
      operands_and_partials.d_x1_[n] -= scaled_diff;
    if (!is_constant_struct<T_loc>::value)
      operands_and_partials.d_x2_[n] += scaled_diff;
    if (!is_constant_struct<T_scale>::value)
      operands_and_partials.d_x3_[n]
        += inv_sigma[n] * (y_minus_mu_over_sigma_squared - 1);
  }
  return operands_and_partials.to_var(logp,y,mu,sigma);
}
```

In each iteration, the value of `y[n]` and `mu[n]` is extracted as a `double`. Then the intermediate terms are calculated. These are needed for every iteration (as long as not all arguments are constants, which was checked earlier in the function). Then depending on the argument types, various terms are added or subtracted from the log density accumulator `logp`. The normalizing term is only included if the `propto` template parameter is false. The $\log \sigma$ term is only included if σ is not a constant. The remaining conditional should always succeed, but is written this way for consistency; the compiler will remove it as its condition is evaluated statically.

After the result is calculated, the derivatives are calculalted. These are added to the `OperandsAndPartials` data structure using the views `d_x1_`, `d_x2_`, and `d_x3_`. These are all analytical partial derivatives of the normal density with respect to its arguments. The operands and partials structure initialized all derivatives to zero. Here, they are incremented (or decremented). If any of the arguments is a scalar, then the view is always of the same element and this effectively increments the single derivative. Thus the same code works for a scalar or vector `sigma`, either breaking the partial across each argument, or reusing `sigma` and incrementing the partial.

The final line just converts the result, with value and arguments, to a `var` for return.

16. Evaluation

In this section, Stan's reverse-mode automatic differentation is evaluated for speed and memory usage. The evaluated version of Stan is version 2.6.3.

16.1. Systems and Versions

In addition to Stan, the following operator overloading, reverse-mode automatic differentiation systems are evaluated.

- Adept, version 1.0: `http://www.met.reading.ac.uk/clouds/adept`
  see (Hogan, 2014)
- Adol-C, version 2.5.2: `https://projects.coin-or.org/ADOL-C`
  see (Griewank and Walther, 2008)
- CppAD, version 1.5: `http://www.coin-or.org/CppAD`
  see (Bell, 2012)
- Sacado (Trilinos), version 11.14: `http://trilinos.org`
  see (Gay, 2005)
- Stan, version 2.6.3: `http://mc-stan.org`
  see this paper

Like Stan, CppAD is purely header only. Although Sacado is distributed as part of the enormous (150MB compressed) Trilinos library, which comes with a complex-to-configure CMake file, Sacado itself is header only and can be run as such by including the proper header files. Adept requires a library archive to be built and linked with client code. Adol-C requires a straightforward CMake configuration step to generate a makefile, which then compiles object files from C and C++ code; the object files are then linked with the client code. Detailed instructions for building all of these libraries from source are included with the evaluation code for this paper and the source versions used for the evaluations are included in the Git repository.

Systems Excluded

Other C++ systems for computing derivatives were excluded from the evaluation for various reasons:

- lack of reverse-mode automatic differentation
  - ADEL: https://github.com/eleks/ADEL
  - CeresSolver: http://ceres-solver.org
  - CTaylor: https://ctaylor.codeplex.com
  - FAD: http://pierre.aubert.free.fr/software/software.php3
- lack of operator overloading to allow differentiation of existing C++ programs
  - ADIC2: http://www.mcs.anl.gov/adic
  - ADNumber: https://code.google.com/p/adnumber
  - AutoDiff_Library: https://github.com/fqiang/autodiff_library
  - CasADi: http://casadi.org
  - CppAdCodeGen: https://github.com/joaoleal/CppADCodeGen
  - Rapsodia: http://www.mcs.anl.gov/Rapsodia
  - TAPENADE: http://tapenade.inria.fr:8080/tapenade/index.jsp
  - Theano: http://deeplearning.net/software/theano
- require graphics processing unit (GPU)
  - AD4CL: https://github.com/msupernaw/AD4CL
- lack of open-source licensing
  - ADC: http://www.vivlabs.com

- AMPL: http://ampl.com
- COSY INFINITY: http://cosy.pa.msu.edu
- FADBAD++: http://www.imm.dtu.dk/~kajm/FADBAD

16.2. What is being Evaluated

The evaluations in this paper are based on simple gradients with retaping for each evaluation.

Gradients vs. Jacobians

All of the evaluations are for simple gradient calculations for functions $f : \mathbb{R}^N \to \mathbb{R}$ with multiple inputs and a single output. That is the primary use case for which Stan's automatic differentiation was designed.

Stan's reverse-mode automatic differentiation can be used to compute Jacobians as shown in Section 4.1.

Stan's lazy evaluation of partial derivatives in the reverse pass over expression graph is designed to save memory in the gradient case, but requires recomputations of gradients when calculating Jacobians of functions $g : \mathbb{R}^N \to \mathbb{R}^M$ with multiple inputs and multiple outputs.

Retaping

Adol-C and CppAD allow the expression graph created in a forward pass through the code (which they call a "tape") to be reused. This can speed up subsequent reverse-mode passes. The drawback to reusing the expression graph is that if there are conditionals or while loops in the program being evaluated, the expression grpah cannot be reused. The second drawback is that to evaluate the function and gradients requires what is effectively an interpreted forward pass to be rerun. A major advantage comes in Jacobian calculations, where all of the expression derivatives computed in the initial expression graph construction can be reused.

Stan is not ideally set up to exploit retaping because of the lazy evaluation of partial derivatives. For the lazy evaluation cases involving expensive functions (multiplications, transcendentals, iterative algorihtms, etc.), this would have to be carried out each time, just as in the Jacobian case.

CppAD goes even further in allowing functions with static expression graphs to be taped once and reused by gluing them together into larger functions.[8] An related approach to compiling small pieces of larger functions is provided by the expression templates of Adept.

[8] CppAD calls this checkpointing; see http://www.coin-or.org/CppAD/Doc/checkpoint.htm for details.

Memory

Stan is very conservative with memory usage compared to other systems. We do not actually evaluate memory because we don't know how to do it. We can analytically evaluate the amount of memory required. In continuous runs, Stan's underlying memory allocation is stored by default and reused, so that there should not actually be any underlying system memory thrashing other than for the Eigen and standard library vectors, which manage their own memory in the C++ heap.

Compile Time

We also have not evaluated compile time. Stan is relatively slow to compile, especially for the probability functions and matrix operations, because of its extensive use of templating. But this cost is only paid once in the system. Once the code is compiled, it can be reused. The use case for which Stan is designed typically involves tens of thousands or even millions of gradient calculations, for which compilation is only performed once.

Systems Still under Consideration

The following systems are open source and provide operator overloading, but the authors of this paper have not (yet) been able to figure out how to install them and get a simple example working.

- AUTODIF (ADMB): http://admb-project.org
- OpenAD: http://www.mcs.anl.gov/OpenAD/

16.3. Functors to Differentiate

To make evaluations easy and to ensure the same code is being evaluated for each system, a functional is provided for each system being evaluated that allows it to compute gradients of a functor.

The functors to differentiate define the following method signature.

```
template <typename T>
T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x) const;
```

Each functor also defines a static name function used in displaying results.

```
static std::string name() const;
```

Each functor further defines a static function that fills in the values to differentiate.

```
template <typename T>
static void fill(Eigen::Matrix<T, Eigen::Dynnamic, 1>& x);
```

The `fill()` function is called once for each size before timing.

For example, the following functor sums the elements of its argument vector.

```
struct sum_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    T sum_x = 0;
    for (int i = 0; i < x.size(); ++i)
      sum_x += x(i);
    return sum;
  }

  static void fill(Eigen::VectorXd& x) const {
    for (int i = 0; i < x.size(); ++i)
      x(i) = i;
  }

  static std::string name() const {
    return "sum";
  }
};
```

16.4. Functionals for Differentiation

The functionals to perform the differentiation of functors for each system are defined as follows.

Adept Gradient Functional

The functional using Adept to calculate gradients of functors is defined as follows.

```
template <typename F>
void adept_gradient(const F& f,
                    const Eigen::VectorXd& x,
                    double& fx,
                    Eigen::VectorXd& grad_fx) {
  Eigen::Matrix<adept::adouble, Eigen::Dynamic, 1> x_ad(x.size());
  for (int i = 0; i < x.size(); ++i)
    x_ad(i) = x(i);
  adept::active_stack()->new_recording();
  adept::adouble fx_ad = f(x_ad);
  fx = fx_ad.value();
  fx_ad.set_gradient(1.0);
```

```
  adept::active_stack()->compute_adjoint();
  grad_fx.resize(x.size());
  for (int i = 0; i < x.size(); ++i)
    grad_fx(i) = x_ad[i].get_gradient();
}
```

A new tape is explicitly started using `new_recording()`, the top variable's adjoint is set to 1 using `set_gradient()`, and derivatives are propagated by calling `compute_adjoint()`. At this point, the gradients can be read out of the automatic diffeentiation variables.

Adol-C Gradient Functional

The functional using Adol-C to calculate gradients of functors is defined as follows.

```
template <typename F>
void adolc_gradient(const F& f,
                    const Eigen::VectorXd& x,
                    double& fx,
                    Eigen::VectorXd& grad_fx) {
  grad_fx.resize(x.size());
  trace_on(1);
  Eigen::Matrix<adouble, Eigen::Dynamic, 1> x_ad(x.size());
  for (int n = 0; n < x.size(); ++n)
    x_ad(n) <<= x(n);
  adouble y = f(x_ad);
  y >>= fx;
  trace_off();
  gradient(1, x.size(), &x(0), &grad_fx(0));
}
```

The Adol-C library signals a new tape recording with the `trace_on()` function. The operator `<<=` is overloaded to create new dependent variables, which are then used to compute the result. The result is written into value `fx` using the `>>=` operator to signal the dependent variable and the recording is turned off using `trace_off()`. Then a function `gradient()` calculates the gradients from the recording.

CppAD Gradient Functional

The functional using CppAD to calculate gradients of functors is defined as follows.

```
template <typename F>
void cppad_gradient(const F& f,
                    const Eigen::VectorXd& x,
```

```
                    double& fx,
                    Eigen::VectorXd& grad_fx) {
  Eigen::Matrix<CppAD::AD<double>, Eigen::Dynamic, 1>
    x_ad(x.size());
  for (int n = 0; n < x.size(); ++n)
    x_ad(n) = x[n];
  Eigen::Matrix<CppAD::AD<double>, Eigen::Dynamic, 1> y(1);
  CppAD::Independent(x_ad);
  y[0] = f(x_ad);
  CppAD::ADFun<double> g = CppAD::ADFun<double>(x_ad, y);
  fx = Value(y[0]);
  Eigen::VectorXd w(1);
  w(0) = 1.0;
  grad_fx =  g.Reverse(1, w);
}
```

CppAD builds-in utilities for working directly with Eigen vectors. CppAD requires a declaration of the independent (input) variables with `Independent()`. Then a function-like object is defined by constructing a `CppAD::ADFun`. The value is extracted using `Value()` and gradients are calculated using the `Reverse` method of the `ADFun` constructed, `g`.

Sacado Gradient Functional

The functional using Sacado to calculate gradients of functors is defined as follows.

```
template <typename F>
void sacado_gradient(const F& f,
                     const Eigen::VectorXd& x,
                     double& fx,
                     Eigen::VectorXd& grad_fx) {
  Eigen::Matrix<Sacado::Rad::ADvar<double>, Eigen::Dynamic, 1>
    x_ad(x.size());
  for (int n = 0; n < x.size(); ++n)
    x_ad(n) = x[n];
  fx = f(x_ad).val();
  Sacado::Rad::ADvar<double>::Gradcomp();

  grad_fx.resize(x.size());
  for (int n = 0; n < x.size(); ++n)
    grad_fx(n) = x_ad(n).adj();
}
```

The execution and memory management of Sacado are very similar to those of Stan. Nothing is required to start recoding other than the use of automatic differentiation variables (here `ADvar<double>`). Functors are applied

as expected and values extracted using the method `val()`. Then gradients are computed with a global function call (here `Gradcomp()`). This functional was implemented without the `try-catch` logic for recovering memory shown in the Stan version, because memory is managed by Sacado inside the call to `Gradcomp()`.

Stan Gradient Functional

The functional using Stan to calculate gradients of functors was defined in Section 6.

16.5. Test Harness

A single file with the test harness code is provided. The test itself is run with the following function, which includes a template parameter for the type of the functor being differentiated.

```
template <typename F>
inline void run_test() {
  adept::Stack adept_global_stack_;
  F f;
  std::string file_name = F::name() + "_eval.csv";
  std::fstream fs(file_name, std::fstream::out);
  print_results_header(fs);
  int max = 16 * 1024;
  for (int N = 1; N <= max; N *= 2) {
    std::cout << "N = " << N << std::endl;
    Eigen::VectorXd x(N);
    F::fill(x);
    time_gradients(f, x, fs);
  }
  fs.close();
}
```

Adept requires a global stack to be allocated by calling its nullary constructor, which is done at the very top of the `run_test()` function.

The key feature here is that a `double`-based vector `x` is defined of size `N`, starting at 1 and doubling through size 2^{14} (16,384) to show how the speed varies as a function of problem size; larger sizes are not provided because 2^{14} was enough to establish the trends with larger data. For each size `N`, the functor's static `fill()` function is applied to `x`, then the gradients are timed. The routine `time_gradients`, which is called for each size of input, is defined as follows.

```
template <typename F>
inline void time_gradients(const F& f, const Eigen::VectorXd& x,
                           std::ostream& os) {
```

```
int N = x.size();
Eigen::VectorXd grad_fx(N);
double fx = 0;
clock_t start;
std::string f_name = F::name();
double z = 0;

z = 0;
start = clock();
for (int m = 0; m < NUM_CALLS; ++m) {
  adept_gradient(f, x, fx, grad_fx);
  z += fx;
}
print_result(start, F::name(), "adept", N, os);

z = 0;
start = clock();
for (int m = 0; m < NUM_CALLS; ++m) {
  adolc_gradient(f, x, fx, grad_fx);
  z += fx;
}
print_result(start, F::name(), "adolc", N, os);

z = 0;
start = clock();
for (int m = 0; m < NUM_CALLS; ++m) {
  cppad_gradient(f, x, fx, grad_fx);
  z += fx;
}
print_result(start, F::name(), "cppad", N, os);

z = 0;
start = clock();
for (int m = 0; m < NUM_CALLS; ++m) {
  sacado_gradient(f, x, fx, grad_fx);
  z += fx;
}
print_result(start, F::name(), "sacado", N, os);

z = 0;
start = clock();
for (int m = 0; m < NUM_CALLS; ++m) {
  stan::math::gradient(f, x, fx, grad_fx);
  z += fx;
}
print_result(start, F::name(), "stan", N, os);
```

```
    z = 0;
    start = clock();
    for (int m = 0; m < NUM_CALLS; ++m)
      z += f(x);
    print_result(start, F::name(), "double", N, os);
  }
```

This function defines the necessary local variables for timing and printing results to a file output stream, prints the header for the output. Then t times each system's functional call to the functor to differentiate and prints the results.

Timing is performed using the `ctime` library `clock()` function, with 100,000 (or 10,000) repeated calls to each automatic differentiation system with the only user programs executing being the Mac OS X Terminal (version 2.5.3).

A variable is defined to pull the value out to ensure that the entire function is not compiled away because results are not used. The final call applies the functor to a vector of `double` values without computing gradients to provide a baseline measurement of function execution time.[9]

The Adol-C and CppAD systems have alternative versions that do not recompute the "tape" for subsequent function calls, but this is not the use case we are evaluating. There is some savings for doing this because Adol-C in particular is relatively slow during the taping stage; see Section 16.2.

16.6. Test Hardware and Compilation

This section provides actual performance numbers for the various systems using

- Hardware: Macbook Pro computer (Retina, Mid 2012), with a 2.3 GHz Intel Core i7 with 16 GB of 1600 MHz DDR3 memory
- Compiler: clang version 3.7.0 (trunk 233481)
- Compiler Flags:
  `-O3 -DNDEBUG -DEIGEN_NO_DEBUG -DADEPT_STACK_THREAD_UNSAFE`
- Libraries: Eigen 3.2.4, Boost 1.55.0

The compiler flags turn on optimization level 3 and turn off system and Eigen-level debugging. They also put Adept into thread-unsafe mode for its stack, which matches the way Stan runs; like Adept, Stan can be run in thread safe mode at roughly a 20% penalty in performance by using thread-local instances of the stack representing the expression graph.

[9] This doesn't seem to make a difference with any compilers; each function was also run in such a way to print the result to ensure that each system was properly configured to compute gradients.

The makefile included with the code for this paper also includes the ability to test with GCC version 3.9. Results were similar enough for GCC that they are not included in this paper.

16.7. Basic Function and Operator Evaluations

This section provides evaluations of basic operators and functions defined as part of the C++ language or as part of the standard `cmath` library. The next section considers evaluations of Stan-specific functions and optimized alternatives to basic functions.

Sums and Products

The simplest functions just add or multiply a sequence of numbers. The sum functor was defined in Section 16.3. The remaining functors will be shown without their `name()` methods. For products, the following functor is used.

```
struct product_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    T product_x = 1;
    for (int i = 0; i < x.size(); ++i)
      product_x *= x(i);
    return product_x;
  }

  static void fill(Eigen::VectorXd& x) {
    for (int i = 0; i < x.size(); ++i)
      x(i) = pow(1e10, 1.0 / x.size());
  }
};
```

The fill function ensures that the total product is 10^{10}.

The evaluation results for sums and products are plotted in Figure 4, with actual time taken shown as well as time relative to Stan's time.[10] For sums and products, the time taken for `double`-based function evaluation versus evaluation with gradient calculations ranges from a factor of 100 for a handful of dimensions to roughly a factor of 10 to 15 for evaluations with 200 or more dimensions. Sacado is the fastest syste for problems with fewer than 8 (sums) or 16 (products) dimensions, Stan is faster for larger problems. It is clear from the plots thtat both CppAD and Adol-C have large constant overheads that make them relatively slow for smaller problems; they are still slower than

[10] Given the difficulty in reading the actual time plots, only relative plots will be shown going forward.

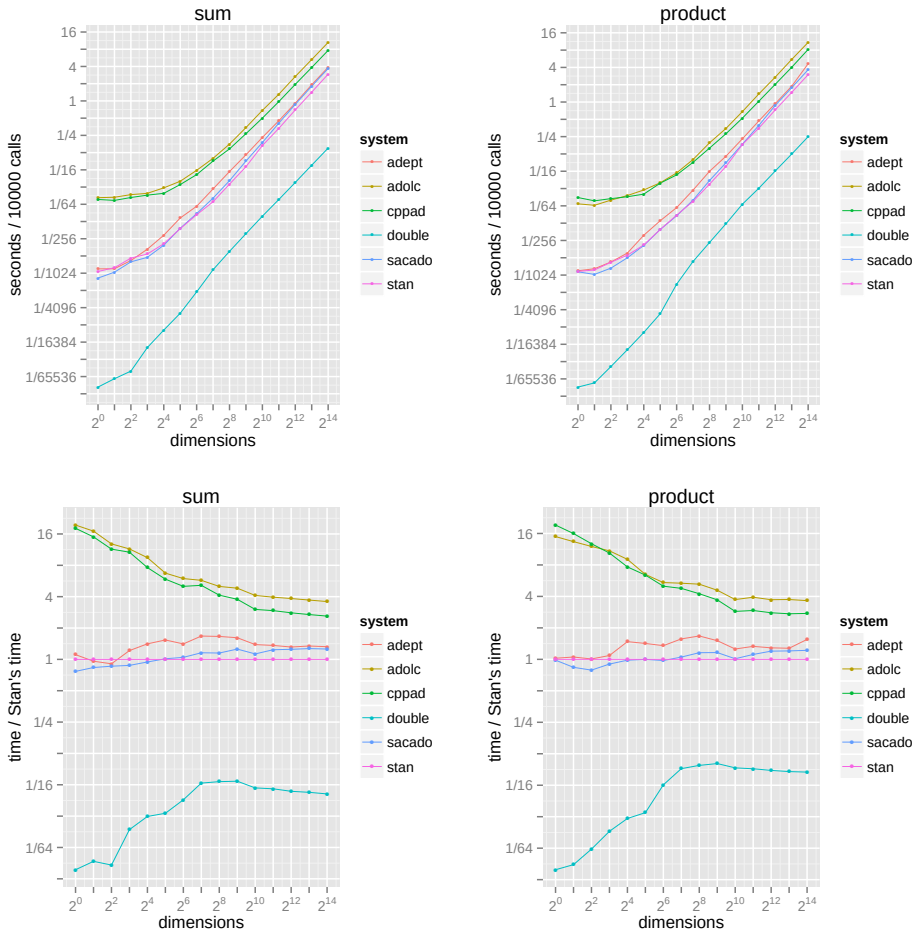

Figure 4: *Evaluation of sums (left) and products (right). The top plots provide a measurement of time per 100,000 gradient calculations. The bottom plots shows the speed of each system relative to Stan's time.*

Sacado, Adept, and Stan for problems of 1000 variables or more where their relative speed seems to stabilize. These overall results are fairly consistent through the evaluations.

Power Function

The following functor is evaluated to test the built-in `pow()` function; because of the structure, a long chain of derivatives is created and both the mantissa and exponent are differentiated.

```
struct powers_fun {
  static void fill(Eigen::VectorXd& x) {
    for (int i = 0; i < x.size(); ++i)
      x(i) = i % 2 == 0 ? 3 : 1.0 / 3;
  }
```

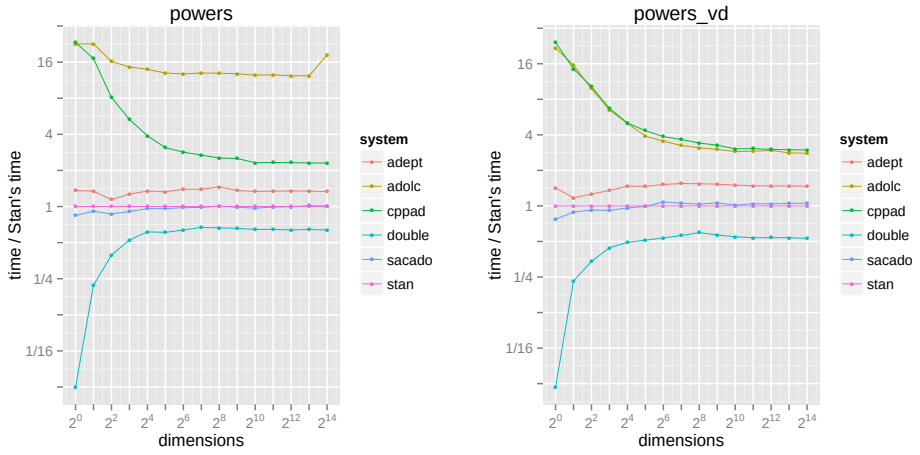

Figure 5: *Relative evaluation of power function. (Left) Plot is for a function with repeated application of* `pow` *to two variable arguments. (Right) The sum of a sequence of* `pow` *applications to a variable with a fixed exponent.*

```
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    T result = 10.0;
    for (int i = 1; i < x.size(); ++i)
      result = pow(result, x(i));
    return result;
  }
};
```

The evaluation is shown in Figure 5. Here, Sacado and Stan have almost identical performance, which is about 40% faster than Adept and much faster than CppAD or Adol-C. Adol-C seems to particularly struggle with this example, being roughly 15 times slower than Stan. Because of the time taken for the `double`-based calculation of `pow()` for fractional powers, for problems of more than 8 dimensions, the gradients are calculated in only about 50% more time than the `double`-based function itself.

The second evaluation of powers is for a fixed exponent with a summation of the results, as would be found in an iterative algorithm.

```
struct powers_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    T result = 10.0;
    for (int i = 1; i < x.size(); ++i)
      result = pow(result, x(i));
```

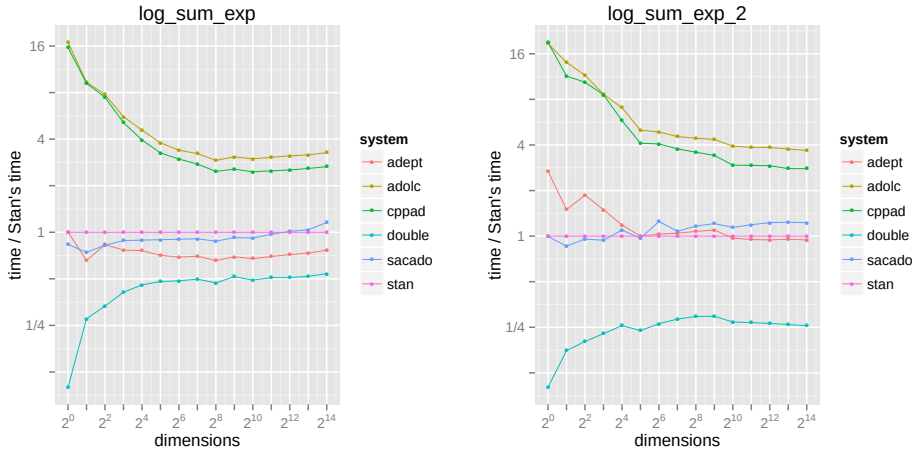

Figure 6: *Relative evaluation of the log sum of exponents function written recursively (left) and directy (right).*

```
    return result;
  }

  static void fill(Eigen::VectorXd& x) {
    for (int i = 0; i < x.size(); ++i)
      x(i) = i % 2 == 0 ? 3 : 1.0 / 3;
  }
};
```

The evaluation of this gradient calculation is shown on the right side of Figure 5. The relative speeds are similar other than for Adol-C, which is much faster in this case. In this case, the gradient calculations in Stan take about twice as long as the function itself.

Logarithm and Exponentiation Functions

The log sum of exponents function is a commonly used function in numerical computing to avoid overflow when adding two numbers on a log scale. Here, a simpler form of it is defined that does not attempt to avoid overflow, though the functor is set up so that results will not overflow.

```
struct log_sum_exp_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    T total = 0.0;
    for (int i = 0; i < x.size(); ++i)
      total = log(exp(total) + exp(x(i)));
    return total;
```

```
  }

  static void fill(Eigen::VectorXd& x) {
    for (int i = 0; i < x.size(); ++i)
      x(i) = i / static_cast<double>(x.size());
  }
};
```

Results are shown in Figure 6. For this operation, the expression templates used in Adept prove their worth and it is about 40% faster than Stan. Sacado is a bit faster than Stan, and again, Adol-C and CppAd are more than twice as slow. Because each calculation is so slow on `double` values, for problems of more than eight dimensions, the gradients are calculated in about double the time it takes to evaluate the function itself on `double` values.

To see that it's Adept's expression templates that make the difference and to illustrate how important the way a function is formulated is, consider this alternative implementation of the same log sum of exponents function.

```
struct log_sum_exp_2_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    T total = 0.0;
    for (int i = 0; i < x.size(); ++i)
      total += exp(x(i));
    return log(total);
  }
  ...
}
```

The result is shown in Figure 6; with this implementation, performance of Stan and Adept are similar. The second (direct) implementation is also much faster for `double`-based function evaluation because of fewer applications of the expensive `log` and `exp` functions.

Matrix Products

This section provides evaluations of taking gradients of matrix products. The first two evaluations are for a naive looping implementation. The first evaluation involves differentiating both matrices in the product.

```
struct matrix_product_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {
```

```
    using Eigen::Matrix;
    using Eigen::Dynamic;
    using Eigen::Map;
    int N = static_cast<int>(std::sqrt(x.size() / 2));
    Matrix<T, Dynamic, Dynamic> a(N,N);
    Matrix<T, Dynamic, Dynamic> b(N,N);
    int i = 0;
    for (int m = 0; m < N; ++m) {
      for (int n = 0; n < N; ++n) {
        a(m,n) = x(i++);
        b(m,n) = x(i++);
      }
    }
    Matrix<T, Dynamic, Dynamic> ab(N,N);
    for (int m = 0; m < N; ++m) {
      for (int n = 0; n < N; ++n) {
        ab(m,n) = 0;
        for (int k = 0; k < N; ++k)
          ab(m,n) += a(m,k) * b(k,n);
      }
    }
    T sum = 0;
    for (int m = 0; m < N; ++m)
      for (int n = 0; n < N; ++n)
        sum += ab(m,n);
    return sum;
  }

  static void fill(Eigen::VectorXd& x) {
    int N = static_cast<int>(sqrt(x.size() / 2));
    if (N < 1) N = 1;
    x.resize(N * N * 2);
    for (int i = 0; i < x.size(); ++i)
      x(i) = static_cast<double>(i + 1) / (x.size() + 1);
  }
};
```

The `fill()` implementation is different than what came before, because the evaluation requires two square matrices. Thus the vector to be filled is resized to the largest vector of variables that can fill two matrices. For a case involving 2^{12} variables, the matrices being multiplied 45×45, because $45 = \lfloor\sqrt{2^{12}/2}\rfloor$.

The implementation of matrix product itself is just straightforward looping, first to compute the matrix product and assign it to the matrix `ab`, then to reduce the matrix to a single value through summation. The resulting expression graph has much higher connectivity, with each variable being repeated N times for an $N \times N$ matrix product.

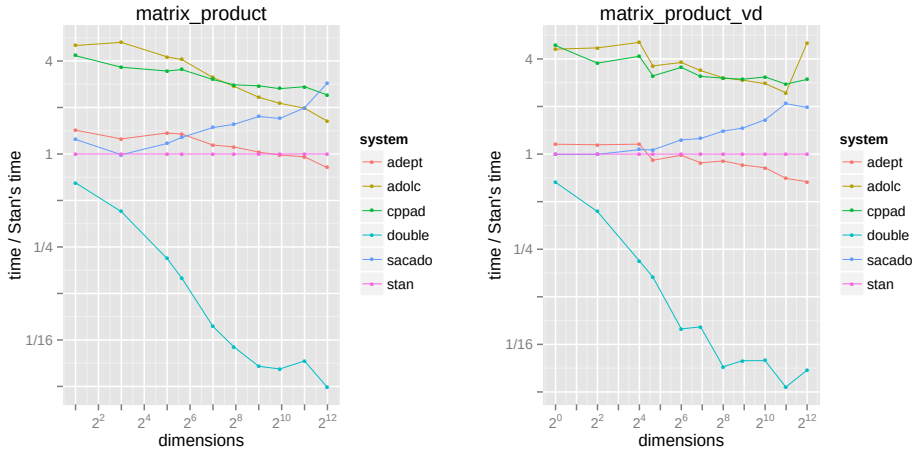

Figure 7: *Relative evaluation of naive loop-based matrix products with derivatives of both matrix (left) and just the first matrix (right). The number of dimensions on the x axis is the total number of entries in the matrix; the number of subexpressions evaluated grows proportionally to the square root of the number of entries.*

For the product of an $N \times N$ matrix, there are N^2 parameters if one matrix has `double` scalar values and $2N^2$ if they are both automatic differentiation variables. From the definition of matrix products, it is clear that there are a total of N^3 products and N^2 sums required to multiply two $N \times N$ matrices. Thus multiplying two 45×45 matrices requires over 90,000 products to be calculated, each of which involves a further multiplication during automatic differentiation.

When considering the evaluations, the number of dimensions is the number of parameters—the matrices have dimensionality equal to the square root of that size.

The relative timing results are shown in Figure 7. Because the matrices are resized to accomodate two square matrices, the points do not fall exactly on even powers of two. Also, because iterating through powers of two sizes, the largest pair of matrices is the same for successive lower orders, so there are duplicated evaluations, which show some variation due to the relatively small number of loops evaluated. It can be seen that Stan is faster for small matrices, with Adept being faster for larger matrices. Sacado fares relatively poorly compared to the less connected evaluations.

The following functor is for the evaluation of matrix products with gradients taken only of the first matrix.

```
struct matrix_product_vd_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    using Eigen::Matrix;
```

```
    using Eigen::Dynamic;
    using Eigen::Map;
    int N = static_cast<int>(std::sqrt(x.size()));
    Matrix<T, Dynamic, Dynamic> a(N,N);
    Matrix<double, Dynamic, Dynamic> b(N,N);
    int i = 0;
    for (int m = 0; m < N; ++m) {
      for (int n = 0; n < N; ++n) {
        a(m,n) = x(i++);
        b(m,n) = 1.02;
      }
    }
    Matrix<T, Dynamic, Dynamic> ab(N,N);
    for (int m = 0; m < N; ++m) {
      for (int n = 0; n < N; ++n) {
        ab(m,n) = 0;
        for (int k = 0; k < N; ++k)
          ab(m,n) += a(m,k) * b(k,n);
      }
    }
    T sum = 0;
    for (int m = 0; m < N; ++m)
      for (int n = 0; n < N; ++n)
        sum += ab(m,n);
    return sum;
  }

  static void fill(Eigen::VectorXd& x) {
    int N = static_cast<int>(sqrt(x.size()));
    if (N < 1) N = 1;
    x.resize(N * N);
    for (int i = 0; i < x.size(); ++i)
      x(i) = static_cast<double>(i + 1) / (x.size() + 1);
  }
};
```

Here, the resizing is to a single square matrix, so the case of 2^{12} variables involves two 64×64 matrices. As with differentiating both sides, Stan is faster for smaller matrices, with Adept being faster for larger matrices.

To demonstrate the utility of a less naive implementation of matrix product, consider the results of using Eigen's built-in matrix product, which is tuned to maximize memory locality. The following functor is used for the evaluation; the `fill()` function is the same as the first matrix example for taking gradients of both components.

```
struct matrix_product_eigen_fun {
  template <typename T>
```

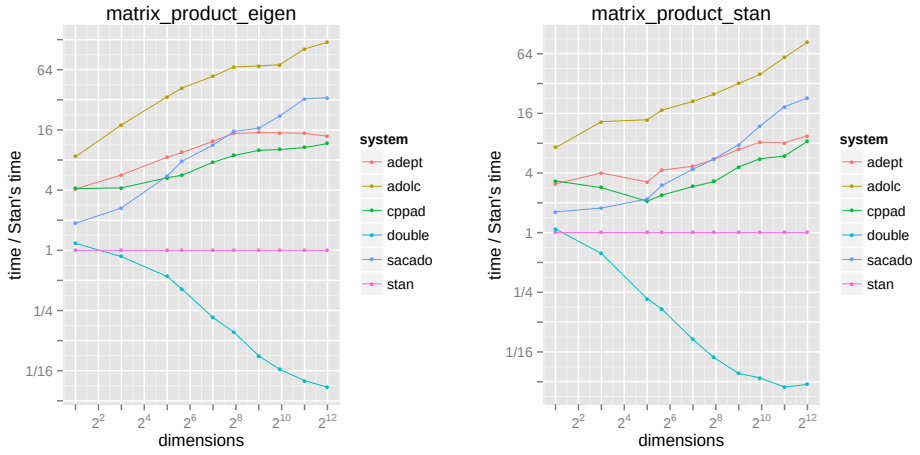

Figure 8: *Evaluation of matrix products using Eigen's built-in* `operator*()` *and* `sum()` *functions (left), and with Stan's built-in* `multiply()` *and* `sum()` *functions (right). The number of dimensions on the x axis is the total number of entries in the matrix; the number of subexpressions evaluated grows proportionally to the square root of the number of entries.*

```
T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
  const {

  using Eigen::Matrix;
  using Eigen::Dynamic;
  int N = static_cast<int>(std::sqrt(x.size() / 2));
  Matrix<T, Dynamic, Dynamic> a(N,N);
  Matrix<T, Dynamic, Dynamic> b(N,N);
  int i = 0;
  for (int m = 0; m < N; ++m) {
    for (int n = 0; n < N; ++n) {
      a(m,n) = x(i++);
      b(m,n) = x(i++);
    }
  }
  return (a * b).sum();
 }
};
```

Both CppAD and Stan specialize `std::numeric_limits`, which is used by Eigen to calculate memory sizes and optimize memory locality in matrix product calculations. The difference in relative speed is striking, as shown in Figure 8. Despite the rather large number of operations required for gradients, relative speed compared to a pure `double`-based implementation is better by a factor of 50%.

The next evaluation replaces Eigen's `operator*` and `sum` methods with customized versions in Stan. Specifically, for Stan the final implementation is

done with

```
stan::math::sum(stan::math::multiply(a, b));
```

rather than with the built-in Eigen operations, as in the previous evaluation:

```
(a * b).sum();
```

Eigen's direct implementation is about 50% faster, but will consume roughly twice as much memory as Stan's custom dot-products and summation, which rely on custom `vari` implementations.

Normal Log Density

The next function is closer to the applications for which Stan was designed, being the log of the normal density function. The functor to be evaluated is the following.

```
struct normal_log_density_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    T mu = -0.56;
    T sigma = 1.37;
    T lp = 0;
    for (int i = 0; i < x.size(); ++i)
      lp += normal_log_density(x(i), mu, sigma);
    return lp;
  }

  static void fill(Eigen::VectorXd& x) {
    for (int i = 0; i < x.size(); ++i)
      x(i) = static_cast<double>(i + 1 - x.size()/2) / (x.size() + 1);
  }
};
```

The density function itself is defined up to an additive constant ($-\log\sqrt{2\pi}$) by the following function.

```
template <typename T>
inline
T normal_log_density(const T& y, const T& mu, const T& sigma) {
  T z = (y - mu) / sigma;
  return -log(sigma) - 0.5 * z * z;
}
```

The timing results are given in Figure 9. For this function, Stan is roughly 50% faster than the next fastest system, Adept, with the advantage declining a

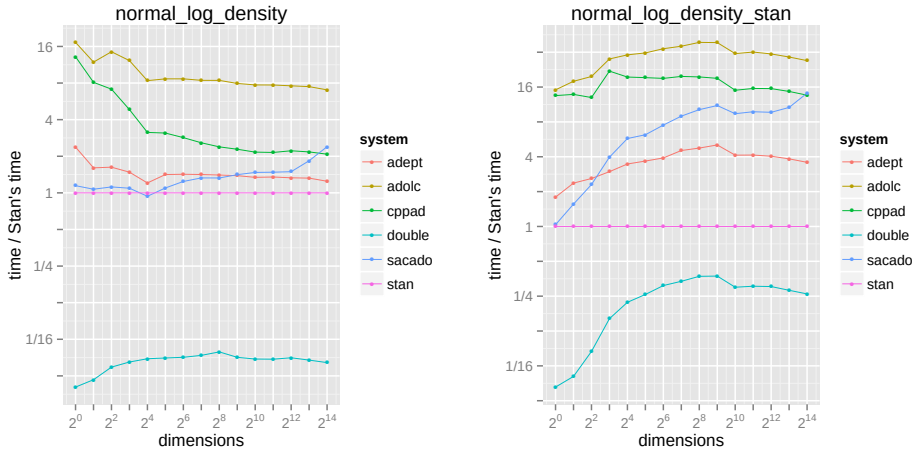

Figure 9: *Evaluation of the normal log density function implemented directly (left) and using Stan's built-in function* `normal_log()` *for the Stan version (right). Stan's approach shows increasing speed relative to the naive double evaluation because the proportion of logarithm evaluations shrinks as the dimensionality grows.*

bit as the problem size gets larger. For problems with ten or more evaluations, Stan's gradient calculation takes a bit more than twenty times as long as the `double`-based evaluation. Althoug roughly the same speed as Stan for small problems, Sacado falls further behind as the number of gradients evaluated increases, whereas CppAd starts very far behind and asymptotes at roughly half the speed of Stan.

The second plot in Figure 9 shows the relative speed of Stan's built-in, vectorized version of the normal log density function. For this evaluation, the functor's operator is the following; the `fill()` function remains the same as before.

```
struct normal_log_density_stan_fun {
  template <typename T>
  T operator()(const Eigen::Matrix<T, Eigen::Dynamic, 1>& x)
    const {

    T mu = -0.56;
    T sigma = 1.37;
    return normal_log_density(x, mu, sigma);
  }
};
```

The implementation of the log density function for Stan is the following.

```
inline stan::math::var
normal_log_density(const Eigen::Matrix<stan::math::var,
                                       Eigen::Dynamic, 1>& y,
                   const stan::math::var& mu,
```

```
                    const stan::math::var& sigma) {
  return stan::prob::normal_log<true>(y, mu, sigma);
}
```

For the other systems, a standalone `normal_log_density` provides the same definition as in the previous evaluation.

```
template <typename T>
inline T
normal_log_density(const Eigen::Matrix<T, Eigen::Dynamic, 1>& y,
                   const T& mu, const T& sigma) {
  T lp = 0;
  for (int i = 0; i < y.size(); ++i) {
    T z = (y(i) - mu) / sigma;
    lp += -log(sigma) - 0.5 * z * z;
  }
  return lp;
}
```

The comparison with the `double`-based version shows that the custom version of the normal log density is roughly a factor of six faster than the naive implementation.

17. Previous Work

Stan's basic pointer-to-implementation pattern and arena-based memory design was based on Gay's original system RAD Gay (2005); RAD is also the basis for the Sacado automatic differentiation package, which is part of the Trilinos project Heroux et al. (2005). The arena-based memory usage pattern is described in more detail in Gay and Aiken (2001). In addition to coding slightly different base classes and memory data structures and responsibilities for recovery, Stan uses `vari` specializations to allow lazy evaluation of partials and thus save memory and reduce the number of assignments. Stan also has a value-based return type for its client class `var` rather than the reference-based return approach of RAD for its function and operator overloads.

18. Summary and Future Work

This paper demonstrated the usability of Stan's automatic differentiation library through simple examples, which showed how to use the data structures directly or fully encapsulate gradient calculations of templated C++ functors using Stan's gradient functional. The efficiency of Stan was compared to other open-source C++ libraries, which showed varying relative performance drops compared to Stan in the different problems evaluated.

The memory usage of Stan's gradient calculations was described in detail, with various techniques being employed to reduce memory usage through lazy

evaluation and vectorization of operations such as log density functions. The Stan automatic differentiation library is part of the sandalone Stan Math Library, which is extensively tested for accuracy and instantiability, and has presented a stable interface through dozens of releases of the larger Stan package. The Stan Math Library is distributed under the BSD license, which is compatible with other open-source licesense such as GPL. Stan's open-source development community ensures continued growth of the library in the future.

Adept's use of expression templates to unfold derivative propagations at compile time makes it more efficient than Stan at run time in cases where there are complex expressions on the right-hand sides of assignments. The advantage grows as the right-hand side expression size grows. There is no reason in principle why the expression templates of Adept and the underlying efficiency of Stan's data structures and memory management could not be combined to improve both systems.

Although not the use case that Stan was developed for, CppAD and Adol-C's ability to re-use a "tape" is potentially useful in some applications where there are no general while loops or conditionals that can evaluate differently on different evaluations.

It is clear from the matrix evaluations that there is also more performance gains to be had for larger matrices, presumably through enhanced memory locality considerations in both expression graph construction and in derivative propagation.

The next major features planned for the stan Math Library are inverse cumulative distribution functions and stiff differential equation solver(s). In order to avoid expensive, imprecise, and unstable finite difference calculations, the latter requires second-order automatic differentiation to compute Jacobians of a differential equation system coupled with its sensitivities.

Acknowledgements

We would like to thank Brad Bell (author of CppAD), Robin Hogan (author of Adept), and Andrew Walther (author of Adol-C) for useful comments on our evaluations of their systems and comments on a draft of this paper. We never heard back from the authors of Sacado.

A. Stan Functions

The following is a comprehensive list of functions supported by Stan as of version 2.6.

A.1. Type Conventions

Scalar Functions

Where argument type `real` occurs, any of `double`, `int`, or `stan::math::var` may be used. Result types with `real` denote `var` if any of the arguments contain a `var` and `double` otherwise.

Vector or Array Functions

Where value type `reals` is used, any of the following types may be used.

- Scalars: `double`, `int`, `var`
- Standard vectors: `vector<double>`, `vector<int>`, `vector<var>`
- Eigen vectors: `Matrix<double,Dynamic,1>`, `Matrix<var,Dynamic,1>`
- Eigen row vectors:
  `Matrix<double,Dynamic,1>`, `Matrix<var,1,Dynamic>`

If any of the arguments contains a `var`, the return type `real` is a `var`, otherwise it is `double`.

A.2. C++ Built-in Arithmetic Operators

function	*arguments*	*return*
operator*	(real x, real y)	real
operator+	(real)	real
operator+	(real x, real y)	real
operator-	(real x)	real
operator-	(real x, real y)	real
operator/	(real x, real y)	real

A.3. C++ Built-in Relational Operators

function	*arguments*	*return*
operator!	(real x)	int
operator!=	(real x, real y)	int
operator&&	(real x, real y)	int
operator<	(real x, real y)	int
operator<=	(real x, real y)	int
operator==	(real x, real y)	int
operator>	(real x, real y)	int
operator>=	(real x, real y)	int
operator||	(real x, real y)	int

A.4. C++ `cmath` Library Functions

function	*arguments*	*return*
`abs`	`(real x)`	`real`
`acos`	`(real x)`	`real`
`acosh`	`(real x)`	`real`
`asin`	`(real x)`	`real`
`asinh`	`(real x)`	`real`
`atan2`	`(real x, real y)`	`real`
`atan`	`(real x)`	`real`
`atanh`	`(real x)`	`real`
`cos`	`(real x)`	`real`
`cosh`	`(real x)`	`real`
`cbrt`	`(real x)`	`real`
`ceil`	`(real x)`	`real`
`erf`	`(real x)`	`real`
`erfc`	`(real x)`	`real`
`exp2`	`(real x)`	`real`
`exp`	`(real x)`	`real`
`fdim`	`(real x, real y)`	`real`
`floor`	`(real x)`	`real`
`fma`	`(real x, real y, real z)`	`real`
`fmax`	`(real x, real y)`	`real`
`fmin`	`(real x, real y)`	`real`
`fmod`	`(real x, real y)`	`real`
`hypot`	`(real x, real y)`	`real`
`log`	`(real x)`	`real`
`log10`	`(real x)`	`real`
`log1p`	`(real x)`	`real`
`log2`	`(real x)`	`real`
`pow`	`(real x, real y)`	`real`
`round`	`(real x)`	`real`
`sin`	`(real x)`	`real`
`sinh`	`(real x)`	`real`
`sqrt`	`(real x)`	`real`
`tan`	`(real x)`	`real`
`tanh`	`(real x)`	`real`
`trunc`	`(real x)`	`real`

A.5. Special Mathematical Functions

function	*arguments*	*return*
bessel_first_kind	(int v, real x)	real
bessel_second_kind	(int v, real x)	real
binary_log_loss	(int y, real y_hat)	real
binomial_coefficient_log	(real x, real y)	real
digamma	(real x)	real
expm1	(real x)	real
fabs	(real x)	real
falling_factorial	(real x, real n)	real
gamma_p	(real a, real z)	real
gamma_q	(real a, real z)	real
inv	(real x)	real
inv_cloglog	(real y)	real
inv_logit	(real y)	real
inv_sqrt	(real x)	real
inv_Phi	(real x)	real
inv_square	(real x)	real
lbeta	(real alpha, real beta)	real
lgamma	(real x)	real
lmgamma	(int n, real x)	real
log1m	(real x)	real
log1m_exp	(real x)	real
log1m_inv_logit	(real x)	real
log1p_exp	(real x)	real
log_diff_exp	(real x, real y)	real
log_falling_factorial	(real x, real n)	real
log_inv_logit	(real x)	real
log_mix	(real theta, real lp1, real lp2)	real
log_rising_factorial	(real x, real n)	real
log_sum_exp	(real x, real y)	real
logit	(real x)	real
modified_bessel_first_kind	(int v, real z)	real
modified_bessel_second_kind	(int v, real z)	real
codemultiply_log	(real x, real y)	real
owens_t	(real h, real a)	real
Phi	(real x)	real
Phi_approx	(real x)	real
rising_factorial	(real x, real n)	real
square	(real x)	real
step	(real x)	real
tgamma	(real x)	real
trigamma	(real x)	real

A.6. Special Control and Test Functions

function	*arguments*	*return*
if_else	(int cond, real x, real y)	real
int_step	(real x)	int
is_inf	(real x)	int
is_nan	(real x)	int

A.7. Matrix Arithmetic Functions

function	*arguments*	*return*
add	(matrix x, matrix y)	matrix
add	(matrix x, real y)	matrix
add	(real x, matrix y)	matrix
add	(real x, row_vector y)	row_vector
add	(real x, vector y)	vector
add	(row_vector x, real y)	row_vector
add	(row_vector x, row_vector y)	row_vector
add	(vector x, real y)	vector
add	(vector x, vector y)	vector
columns_dot_product	(matrix x, matrix y)	row_vector
columns_dot_product	(row_vector x, row_vector y)	row_vector
columns_dot_product	(vector x, vector y)	row_vector
columns_dot_self	(matrix x)	row_vector
columns_dot_self	(row_vector x)	row_vector
columns_dot_self	(vector x)	row_vector
crossprod	(matrix x)	matrix
diag_post_multiply	(matrix m, row_vector rv)	matrix
diag_post_multiply	(matrix m, vector v)	matrix
diag_pre_multiply	(row_vector rv, matrix m)	matrix
diag_pre_multiply	(vector v, matrix m)	matrix
divide	(matrix x, real y)	matrix
divide	(row_vector x, real y)	row_vector
divide	(vector x, real y)	vector
dot_product	(row_vector x, row_vector y)	real
dot_product	(row_vector x, vector y)	real
dot_product	(vector x, row_vector y)	real
dot_product	(vector x, vector y)	real
dot_self	(row_vector x)	real
dot_self	(vector x)	real
elt_divide	(matrix x, matrix y)	matrix
elt_divide	(matrix x, real y)	matrix
elt_divide	(real x, matrix y)	matrix
elt_divide	(real x, row_vector y)	row_vector
elt_divide	(real x, vector y)	vector
elt_divide	(row_vector x, real y)	row_vector
elt_divide	(row_vector x, row_vector y)	row_vector
elt_divide	(vector x, real y)	vector
elt_divide	(vector x, vector y)	vector

function	*arguments*	*return*
elt_multiply	(matrix x, matrix y)	matrix
elt_multiply	(row_vector x, row_vector y)	row_vector
elt_multiply	(vector x, vector y)	vector
multiply_lower_tri_self_transpose	(matrix x)	matrix
multiply	(matrix x, matrix y)	matrix
multiply	(matrix x, real y)	matrix
multiply	(matrix x, vector y)	vector
multiply	(real x, matrix y)	matrix
multiply	(real x, row_vector y)	row_vector
multiply	(real x, vector y)	vector
multiply	(row_vector x, matrix y)	row_vector
multiply	(row_vector x, real y)	row_vector
multiply	(row_vector x, vector y)	real
multiply	(vector x, real y)	vector
multiply	(vector x, row_vector y)	matrix
mdivide_right	(matrix B, matrix A)	matrix
mdivide_right	(row_vector b, matrix A)	row_vector
mdivide_left	(matrix A, matrix B)	matrix
mdivide_left	(matrix A, vector b)	vector
quad_form	(matrix A, matrix B)	matrix
quad_form	(matrix A, vector B)	real
quad_form_diag	(matrix m, row_vector rv)	matrix
quad_form_diag	(matrix m, vector v)	matrix
quad_form_sym	(matrix A, matrix B)	matrix
quad_form_sym	(matrix A, vector B)	real
rows_dot_product	(matrix x, matrix y)	vector
rows_dot_product	(row_vector x, row_vector y)	vector
rows_dot_product	(vector x, vector y)	vector
rows_dot_self	(matrix x)	vector
rows_dot_self	(row_vector x)	vector
rows_dot_self	(vector x)	vector
subtract	(matrix x)	matrix
subtract	(matrix x, matrix y)	matrix
subtract	(matrix x, real y)	matrix
subtract	(real x)	real
subtract	(real x, matrix y)	matrix
subtract	(real x, row_vector y)	row_vector
subtract	(real x, vector y)	vector
subtract	(row_vector x)	row_vector
subtract	(row_vector x, real y)	row_vector
subtract	(row_vector x, row_vector y)	row_vector
subtract	(vector x)	vector
subtract	(vector x, real y)	vector
subtract	(vector x, vector y)	vector
tcrossprod	(matrix x)	matrix

A.8. Special Matrix Functions

function	*arguments*	*return*
cumulative_sum	(real[] x)	real[]
cumulative_sum	(row_vector rv)	row_vector
cumulative_sum	(vector v)	vector
dims	(T x)	int[]
distance	(row_vector x, row_vector y)	real
distance	(row_vector x, vector y)	real
distance	(vector x, row_vector y)	real
distance	(vector x, vector y)	real
exp	(matrix x)	matrix
exp	(row_vector x)	row_vector
exp	(vector x)	vector
log	(matrix x)	matrix
log	(row_vector x)	row_vector
log	(vector x)	vector
log_softmax	(vector x)	vector
log_sum_exp	(matrix x)	real
log_sum_exp	(real x[])	real
log_sum_exp	(row_vector x)	real
log_sum_exp	(vector x)	real
max	(matrix x)	real
max	(real x[])	real
max	(row_vector x)	real
max	(vector x)	real
mean	(matrix x)	real
mean	(real x[])	real
mean	(row_vector x)	real
mean	(vector x)	real
min	(int x[])	int
min	(matrix x)	real
min	(real x[])	real
min	(row_vector x)	real
min	(vector x)	real

function	*arguments*	*return*
num_elements	(T[] x)	int
num_elements	(matrix x)	int
num_elements	(row_vector x)	int
num_elements	(vector x)	int
prod	(int x[])	real
prod	(matrix x)	real
prod	(real x[])	real
prod	(row_vector x)	real
prod	(vector x)	real
sd	(matrix x)	real
sd	(real x[])	real
sd	(row_vector x)	real
sd	(vector x)	real
softmax	(vector x)	vector
squared_distance	(row_vector x, row_vector y[])	real
squared_distance	(row_vector x, vector y[])	real
squared_distance	(vector x, row_vector y[])	real
squared_distance	(vector x, vector y)	real
sum	(int x[])	int
sum	(matrix x)	real
sum	(real x[])	real
sum	(row_vector x)	real
sum	(vector x)	real
variance	(matrix x)	real
variance	(real x[])	real
variance	(row_vector x)	real
variance	(vector x)	real

A.9. Matrix and Array Manipulation Functions

function	*arguments*	*return*
append_col	(matrix x, matrix y)	matrix
append_col	(matrix x, vector y)	matrix
append_col	(row_vector x, row_vector y)	row_vector
append_col	(vector x, matrix y)	matrix
append_col	(vector x, vector y)	matrix
append_row	(matrix x, matrix y)	matrix
append_row	(matrix x, row_vector y)	matrix
append_row	(row_vector x, matrix y)	matrix
append_row	(row_vector x, row_vector y)	matrix
append_row	(vector x, vector y)	vector
block	(matrix x, int i, int j, int n_rows, int n_cols)	matrix
col	(matrix x, int n)	vector
cols	(matrix x)	int
cols	(row_vector x)	int
cols	(vector x)	int
diag_matrix	(vector x)	matrix
diagonal	(matrix x)	vector
head	(T[] sv, int n)	T[]
head	(row_vector rv, int n)	row_vector
head	(vector v, int n)	vector
transpose	(matrix x)	matrix
transpose	(row_vector x)	vector
transpose	(vector x)	row_vector
rank	(int[] v, int s)	int
rank	(real[] v, int s)	int
rank	(row_vector v, int s)	int
rank	(vector v, int s)	int
rep_array	(T x, int k, int m, int n)	T[]
rep_array	(T x, int m, int n)	T[]
rep_array	(T x, int n)	T[]
rep_matrix	(real x, int m, int n)	matrix
rep_matrix	(row_vector rv, int m)	matrix
rep_matrix	(vector v, int n)	matrix
rep_row_vector	(real x, int n)	row_vector
rep_vector	(real x, int m)	vector

function	*arguments*	*return*
row	(matrix x, int m)	row_vector
rows	(matrix x)	int
rows	(row_vector x)	int
rows	(vector x)	int
segment	(T[] sv, int i, int n)	T[]
segment	(row_vector v, int i, int n)	row_vector
segment	(vector v, int i, int n)	vector
sort_asc	(int[] v)	int[]
sort_asc	(real[] v)	real[]
sort_asc	(row_vector v)	row_vector
sort_asc	(vector v)	vector
sort_desc	(int[] v)	int[]
sort_desc	(real[] v)	real[]
sort_desc	(row_vector v)	row_vector
sort_desc	(vector v)	vector
sort_indices_asc	(int[] v)	int[]
sort_indices_asc	(real[] v)	int[]
sort_indices_asc	(row_vector v)	int[]
sort_indices_asc	(vector v)	int[]
sort_indices_desc	(int[] v)	int[]
sort_indices_desc	(real[] v)	int[]
sort_indices_desc	(row_vector v)	int[]
sort_indices_desc	(vector v)	int[]
sub_col	(matrix x, int i, int j, int n_rows)	vector
sub_row	(matrix x, int i, int j, int n_cols)	row_vector
tail	(T[] sv, int n)	T[]
tail	(row_vector rv, int n)	row_vector
tail	(vector v, int n)	vector
to_array_1d	(int[...] a)	int[]
to_array_1d	(matrix m)	real[]
to_array_1d	(real[...] a)	real[]
to_array_1d	(row_vector v)	real[]
to_array_1d	(vector v)	real[]
to_array_2d	(matrix m)	real[,]
to_matrix	(int[,] a)	matrix
to_matrix	(matrix m)	matrix
to_matrix	(real[,] a)	matrix
to_matrix	(row_vector v)	matrix
to_matrix	(vector v)	matrix
to_row_vector	(int[] a)	row_vector
to_row_vector	(matrix m)	row_vector
to_row_vector	(real[] a)	row_vector
to_row_vector	(row_vector v)	row_vector
to_row_vector	(vector v)	row_vector
to_vector	(int[] a)	vector
to_vector	(matrix m)	vector
to_vector	(real[] a)	vector
to_vector	(row_vector v)	vector
to_vector	(vector v)	vector

A.10. Linear Algebra Functions

function	*arguments*	*return*
cholesky_decompose	(matrix A)	matrix
determinant	(matrix A)	real
eigenvalues_sym	(matrix A)	vector
eigenvectors_sym	(matrix A)	matrix
inverse	(matrix A)	matrix
inverse_spd	(matrix A)	matrix
log_determinant	(matrix A)	real
mdivide_left_tri_low	(matrix A, matrix B)	matrix
mdivide_left_tri_low	(matrix A, vector B)	vector
mdivide_right_tri_low	(matrix B, matrix A)	matrix
mdivide_right_tri_low	(row_vector B, matrix A)	row_vector
qr_Q	(matrix A)	matrix
qr_R	(matrix A)	matrix
singular_values	(matrix A)	vector
trace	(matrix A)	real
trace_gen_quad_form	(matrix D,matrix A, matrix B)	real
trace_quad_form	(matrix A, matrix B)	real

A.11. Probability Functions

function	*arguments*	*return*
bernoulli_ccdf_log	(ints y, reals theta)	real
bernoulli_cdf	(ints y, reals theta)	real
bernoulli_cdf_log	(ints y, reals theta)	real
bernoulli_log	(ints y, reals theta)	real
bernoulli_logit_log	(ints y, reals alpha)	real
beta_binomial_ccdf_log	(ints n, ints N, reals alpha, reals beta)	real
beta_binomial_cdf	(ints n, ints N, reals alpha, reals beta)	real
beta_binomial_cdf_log	(ints n, ints N, reals alpha, reals beta)	real
beta_binomial_log	(ints n, ints N, reals alpha, reals beta)	real
beta_ccdf_log	(reals theta, reals alpha, reals beta)	real
beta_cdf	(reals theta, reals alpha, reals beta)	real
beta_cdf_log	(reals theta, reals alpha, reals beta)	real
beta_log	(reals theta, reals alpha, reals beta)	real
binomial_ccdf_log	(ints n, ints N, reals theta)	real
binomial_cdf	(ints n, ints N, reals theta)	real
binomial_cdf_log	(ints n, ints N, reals theta)	real
binomial_log	(ints n, ints N, reals theta)	real
binomial_logit_log	(ints n, ints N, reals alpha)	real
categorical_log	(ints y, vector theta)	real
categorical_logit_log	(ints y, vector beta)	real
cauchy_ccdf_log	(reals y, reals mu, reals sigma)	real
cauchy_cdf	(reals y, reals mu, reals sigma)	real
cauchy_cdf_log	(reals y, reals mu, reals sigma)	real
cauchy_log	(reals y, reals mu, reals sigma)	real
chi_square_ccdf_log	(reals y, reals nu)	real
chi_square_cdf	(reals y, reals nu)	real
chi_square_cdf_log	(reals y, reals nu)	real
chi_square_log	(reals y, reals nu)	real
dirichlet_log	(vector theta, vector alpha)	real
double_exponential_ccdf_log	(reals y, reals mu, reals sigma)	real
double_exponential_cdf	(reals y, reals mu, reals sigma)	real
double_exponential_cdf_log	(reals y, reals mu, reals sigma)	real
double_exponential_log	(reals y, reals mu, reals sigma)	real
exp_mod_normal_ccdf_log	(reals y, reals mu, reals sigma reals lambda)	real
exp_mod_normal_cdf	(reals y, reals mu, reals sigma reals lambda)	real
exp_mod_normal_cdf_log	(reals y, reals mu, reals sigma reals lambda)	real
exp_mod_normal_log	(reals y, reals mu, reals sigma reals lambda)	real
exponential_ccdf_log	(reals y, reals beta)	real
exponential_cdf	(reals y, reals beta)	real
exponential_cdf_log	(reals y, reals beta)	real
exponential_log	(reals y, reals beta)	real

function	*arguments*	*return*
frechet_ccdf_log	(reals y, reals alpha, reals sigma)	real
frechet_cdf	(reals y, reals alpha, reals sigma)	real
frechet_cdf_log	(reals y, reals alpha, reals sigma)	real
frechet_log	(reals y, reals alpha, reals sigma)	real
gamma_ccdf_log	(reals y, reals alpha, reals beta)	real
gamma_cdf	(reals y, reals alpha, reals beta)	real
gamma_cdf_log	(reals y, reals alpha, reals beta)	real
gamma_log	(reals y, reals alpha, reals beta)	real
gaussian_dlm_obs_log	(matrix y, matrix F, matrix G, matrix V, matrix W, vector m0, matrix C0)	real
gaussian_dlm_obs_log	(matrix y, matrix F, matrix G, vector V, matrix W, vector m0, matrix C0)	real
gumbel_ccdf_log	(reals y, reals mu, reals beta)	real
gumbel_cdf	(reals y, reals mu, reals beta)	real
gumbel_cdf_log	(reals y, reals mu, reals beta)	real
gumbel_log	(reals y, reals mu, reals beta)	real
hypergeometric_log	(int n, int N, int a, int b)	real
inv_chi_square_ccdf_log	(reals y, reals nu)	real
inv_chi_square_cdf	(reals y, reals nu)	real
inv_chi_square_cdf_log	(reals y, reals nu)	real
inv_chi_square_log	(reals y, reals nu)	real
inv_gamma_ccdf_log	(reals y, reals alpha, reals beta)	real
inv_gamma_cdf	(reals y, reals alpha, reals beta)	real
inv_gamma_cdf_log	(reals y, reals alpha, reals beta)	real
inv_gamma_log	(reals y, reals alpha, reals beta)	real
inv_wishart_log	(matrix W, real nu, matrix Sigma)	real
lkj_corr_cholesky_log	(matrix L, real eta)	real
lkj_corr_log	(matrix y, real eta)	real
logistic_ccdf_log	(reals y, reals mu, reals sigma)	real
logistic_cdf	(reals y, reals mu, reals sigma)	real
logistic_cdf_log	(reals y, reals mu, reals sigma)	real
logistic_log	(reals y, reals mu, reals sigma)	real
lognormal_ccdf_log	(reals y, reals mu, reals sigma)	real
lognormal_cdf	(reals y, reals mu, reals sigma)	real
lognormal_cdf_log	(reals y, reals mu, reals sigma)	real
lognormal_log	(reals y, reals mu, reals sigma)	real

function	*arguments*	*return*
multi_gp_cholesky_log	(matrix y, matrix L, vector w)	real
multi_gp_log	(matrix y, matrix Sigma, vector w)	real
multi_normal_cholesky_log	(row_vectors y, row_vectors mu, matrix L)	real
multi_normal_cholesky_log	(row_vectors y, vectors mu, matrix L)	real
multi_normal_cholesky_log	(vectors y, row_vectors mu, matrix L)	real
multi_normal_cholesky_log	(vectors y, vectors mu, matrix L)	real
multi_normal_log	(row_vectors y, row_vectors mu, matrix Sigma)	real
multi_normal_log	(row_vectors y, vectors mu, matrix Sigma)	real
multi_normal_log	(vectors y, row_vectors mu, matrix Sigma)	real
multi_normal_log	(vectors y, vectors mu, matrix Sigma)	real
multi_normal_prec_log	(row_vectors y, row_vectors mu, matrix Omega)	real
multi_normal_prec_log	(row_vectors y, vectors mu, matrix Omega)	real
multi_normal_prec_log	(vectors y, row_vectors mu, matrix Omega)	real
multi_normal_prec_log	(vectors y, vectors mu, matrix Omega)	real
multi_student_t_log	(row_vectors y, real nu, row_vectors mu, matrix Sigma)	real
multi_student_t_log	(row_vectors y, real nu, vectors mu, matrix Sigma)	real
multi_student_t_log	(vectors y, real nu, row_vectors mu, matrix Sigma)	real
multi_student_t_log	(vectors y, real nu, vectors mu, matrix Sigma)	real
multinomial_log	(int[] y, vector theta)	real
neg_binomial_2_ccdf_log	(ints n, reals mu, reals phi)	real
neg_binomial_2_cdf	(ints n, reals mu, reals phi)	real
neg_binomial_2_cdf_log	(ints n, reals mu, reals phi)	real
neg_binomial_2_log	(ints y, reals mu, reals phi)	real
neg_binomial_2_log_log	(ints y, reals eta, reals phi)	real
neg_binomial_ccdf_log	(ints n, reals alpha, reals beta)	real
neg_binomial_cdf	(ints n, reals alpha, reals beta)	real
neg_binomial_cdf_log	(ints n, reals alpha, reals beta)	real
neg_binomial_log	(ints n, reals alpha, reals beta)	real
normal_ccdf_log	(reals y, reals mu, reals sigma)	real
normal_cdf	(reals y, reals mu, reals sigma)	real
normal_cdf_log	(reals y, reals mu, reals sigma)	real
normal_log	(reals y, reals mu, reals sigma)	real
ordered_logistic_log	(int k, real eta, vector c)	real
pareto_ccdf_log	(reals y, reals y_min, reals alpha)	real
pareto_cdf	(reals y, reals y_min, reals alpha)	real
pareto_cdf_log	(reals y, reals y_min, reals alpha)	real
pareto_log	(reals y, reals y_min, reals alpha)	real
pareto_type_2_ccdf_log	(reals y, reals mu, reals lambda, reals alpha)	real
pareto_type_2_cdf	(reals y, reals mu, reals lambda, reals alpha)	real
pareto_type_2_cdf_log	(reals y, reals mu, reals lambda, reals alpha)	real
pareto_type_2_log	(reals y, reals mu, reals lambda, reals alpha)	real

function	*arguments*	*return*
poisson_ccdf_log	(ints n, reals lambda)	real
poisson_cdf	(ints n, reals lambda)	real
poisson_cdf_log	(ints n, reals lambda)	real
poisson_log	(ints n, reals lambda)	real
poisson_log_log	(ints n, reals alpha)	real
rayleigh_ccdf_log	(real y, real sigma)	real
rayleigh_cdf	(real y, real sigma)	real
rayleigh_cdf_log	(real y, real sigma)	real
rayleigh_log	(reals y, reals sigma)	real
scaled_inv_chi_square_ccdf_log	(reals y, reals nu, reals sigma)	real
scaled_inv_chi_square_cdf	(reals y, reals nu, reals sigma)	real
scaled_inv_chi_square_cdf_log	(reals y, reals nu, reals sigma)	real
scaled_inv_chi_square_log	(reals y, reals nu, reals sigma)	real
skew_normal_ccdf_log	(reals y, reals mu, reals sigma, reals alpha)	real
skew_normal_cdf	(reals y, reals mu, reals sigma, reals alpha)	real
skew_normal_cdf_log	(reals y, reals mu, reals sigma, reals alpha)	real
skew_normal_log	(reals y, reals mu, reals sigma, reals alpha)	real
student_t_ccdf_log	(reals y, reals nu, reals mu, reals sigma)	real
student_t_cdf	(reals y, reals nu, reals mu, reals sigma)	real
student_t_cdf_log	(reals y, reals nu, reals mu, reals sigma)	real
student_t_log	(reals y, reals nu, reals mu, reals sigma)	real
uniform_ccdf_log	(reals y, reals alpha, reals beta)	real
uniform_cdf	(reals y, reals alpha, reals beta)	real
uniform_cdf_log	(reals y, reals alpha, reals beta)	real
uniform_log	(reals y, reals alpha, reals beta)	real
von_mises_log	(reals y, reals mu, reals kappa)	real
weibull_ccdf_log	(reals y, reals alpha, reals sigma)	real
weibull_cdf	(reals y, reals alpha, reals sigma)	real
weibull_cdf_log	(reals y, reals alpha, reals sigma)	real
weibull_log	(reals y, reals alpha, reals sigma)	real
wiener_log	(reals y, reals alpha, reals tau, reals beta, reals delta)	real
wishart_log	(matrix W, real nu, matrix Sigma)	real

References

Ahnert, K. and Mulansky, M. (2014). *odeint: Solving ODEs in C++*. 40, 41

Bell, B. M. (2012). CppAD: a package for C++ algorithmic differentiation. *Computational Infrastructure for Operations Research*. 57

Betancourt, M., Byrne, S., Livingstone, S., and Girolami, M. (2014). The geometric foundations of Hamiltonian Monte Carlo. *ArXiv e-prints*, 1410.5110. 46

Cohen, S. D. and Hindmarsh, A. C. (1996). CVODE, a stiff/nonstiff ODE solver in C. *Computers in Physics*, 10(2):138–143. 41

Duane, A., Kennedy, A., Pendleton, B., and Roweth, D. (1987). Hybrid Monte Carlo. *Physics Letters B*, 195(2):216–222. 46

Fog, A. (2014). Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. Technical report, Technical University of Denmark. 38

Fornberg, B. (1988). Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706. 6

Gay, D. and Aiken, A. (2001). Language support for regions. In *Programming Language Design and Implementation (PLDI)*, volume 36, pages 70–80. ACM SIGPLAN. 17, 78

Gay, D. M. (2005). Semiautomatic differentiation for efficient gradient computations. In Bücker, H. M., Corliss, G. F., Hovland, P., Naumann, U., and Norris, B., editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 147–158. Springer, New York. 17, 57, 78

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2013). *Bayesian Data Analysis*. Chapman &Hall/CRC Press, London, third edition. 46

Giles, M. (2008). An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation. Technical Report 08/01, Oxford University. 2, 36

Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations*. Johns Hopkins University Press, 3rd edition. 38

Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics (SIAM), 2nd edition. 2, 57

Guennebaud, G., Jacob, B., et al. (2010). Eigen version 3. http://eigen.tuxfamily.org. 11

Hascoët, L. and Pascual, V. (2013). The Tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):20. 7

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. (2005). An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423. 78

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM: Society for Industrial and Applied Mathematics, 2nd edition. 6

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. (2005). SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396. 41

Hogan, R. J. (2014). Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software*, 40(4):26:1–26:16. 7, 57

International Standardization Organization (2003). *International Standard ISO/IEC 14882: Programming languages—C++*. American National Standards Institute, 2003-10-15 (2nd) edition. 10

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1, Fundamental Algorithms*. Addison-Wesley, 3rd edition. 4

Magnus, J. R. and Neudecker, H. (2007). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. John Wiley & Sons, 3rd edition. 36

Neal, R. (2011). MCMC using Hamiltonian dynamics. In Brooks, S., Gelman, A., Jones, G. L., and Meng, X.-L., editors, *Handbook of Markov Chain Monte Carlo*, pages 116–162. Chapman and Hall/CRC. 46

Petersen, K. B. and Pedersen, M. S. (2012). The matrix cookbook. *Technical University of Denmark*. 36

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes in C++*. Cambridge University Press, 3rd edition. 39

Schäling, B. (2011). The Boost C++ libraries. http://en.highscore.de/cpp/boost/. 10

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley. 11, 14

Sutter, H. (1998). Pimpls—beauty marks you can depend on. *C++ Report*, 10(5). 14

Sutter, H. (2001). *More Exceptional C++*. C++ In Depth. Addison-Wesley. 14

SymPy Development Team (2014). *SymPy: Python library for symbolic mathematics*. 7

Vandevoorde, D. and Josuttis, N. M. (2002). *C++ Templates: The Complete Guide*. Addison-Wesley. 7, 10

Weber, S., Carpenter, B., Lee, D., Bois, F. Y., Gelman, A., and Racine, A. (2014). Bayesian drug-disease model with Stan—using published longitudinal data summaries in population models. *Abstracts of the Annual Meeting of the Population Approach Group in Europe (PAGE)*. 40

Wolfram Research, Inc. (2014). *Mathematica*, 10th edition. 6