УДК 519.683.8

Символьное вычисление производных функций многих переменных во время компиляции средствами шаблонов C++

Г. И. Рудой
Московский физико-технический институт

1 Введение

Задача вычисления производных сложной функции возникает во многих вычислительных задачах, таких как численная оптимизация, построение регрессионных моделей [1] и тому подобные. Во многих случаях функция, производные которой необходимо вычислить, известна наперед во время написания программы.

Одним из подходов к решению этой задачи, помимо автоматического дифференцирования [2], является получение аналитических выражений для производных в явном виде либо вручную, либо при помощи различных математических пакетов, и последующее программирование полученных производных в виде соответствующей последовательности операторов целевого языка программирования.

Подобный подход позволяет добиться хорошей производительности, поскольку вычисление таких выражений может быть оптимизировано на этапе компиляции, но вместе с этим он имеет и ряд минусов, как то:

- возможность допустить труднообнаружимую ошибку или опечатку при программировании выражений для производных,
- необходимость вручную обновлять все соответствующие выражения при модификации исходной функции, производные которой необходимо вычислять,
- отсутствие гарантий со стороны компилятора и системы типов, что в коде программы вычисляются производные, вообще говоря, той функции, что ожидается.

В предлагаемом решении [3] для языка C++ производные функций не указываются в явном виде, а вычисляются средствами компилятора в процессе компиляции программы. При этом у компилятора остается возможность применить все те же оптимизации, которые были бы использованы при ручном программировании выражений для производных.

В вычислительном эксперименте метод применен к вычислению производных нелинейной функции, описывающей мощность лазера как функцию прозрачности его резонатора, и производительность этого метода сравнена с вручную оптимизированным кодом вычисления соответствующих производных.

2 Постановка задачи

Пусть дана некоторая функция многих переменных $y = f(\mathbf{x})$, где $\mathbf{x} \in \mathbb{R}^n, \mathbf{x} = (x_1, \ldots, x_n)$. Требуется выразить эту функцию в виде кода на C++ таким образом, чтобы не требовалось вручную указывать выражения для $\frac{\partial f}{\partial x_i}$, сохранив при этом возможность применения оптимизаций компилятора к вычислению этих выражений.

Отметим, что требование возможности компиляторных оптимизаций заставляет отказаться от решений, представляющих функцию f в виде древообразной структуры, обход и дифференцирование которой происходит во время выполнения программы.

3 Представление функций

В предлагаемом решении функция f представляется в виде некоторого типа, который выводится компилятором для композиции заранее заданных базовых термов. Так, например, функция $f(x_1) = x_1^{x_1}$ указывается следующим образом:

```
using Formula_t = decltype (Pow (x1, x1));
```

где `x1` и `Pow` — определенные в предлагаемой библиотеке термы, представляющие соответственно свободную переменную и операцию возведения в степень.

Отметим, что использование типов для представления функций вместе со вспомогательными функциями (таких, как `Pow`), имеющими объявления, но не определения, гарантирует, что дифференцирование действительно происходит во время компиляции программы.

Тип функции f строится как рекурсивное дерево выражения. Для представления узла дерева используется следующая структура:

```
template<typename NodeClass, typename... Args>
struct Node;
```

где `NodeClass` — тип узла (например, переменная, константа, унарная функция, бинарная функция), а `Args` — параметры этого узла (например, индекс переменной, значение константы, дочерний узел унарной функции, дочерние узлы бинарной функции соответственно).

Специализации `Node` поддерживают следующую функциональность:

- взятие производной по правилу дифференцирования сложных функций;
- вычисление значения узла согласно его параметрам в данной точке;
- формирование текстового представления узла.

3.1 Целочисленные константы

Для определения узла, соответствующего целочисленной константе, сначала определим вспомогательный тип:

```
using NumberType_t = long long;

template<NumberType_t N>
struct Number {};
```

Тогда узел, соответствующий целочисленной константе, представляется следующим образом:

```
template<NumberType_t N>
struct Node<Number<N>>
{
  template<char FPrime, int IPrime>
  using Derivative_t = Node<Number<0>>;

  static std::string Print ()
```

```
  {
    return std::to_string (N);
  }

  template<typename Vec>
  static typename Vec::value_type Eval (const Vec&)
  {
    return N;
  }

  constexpr Node () {}
};
```

Действительно, производная любого числа — ноль (этому соответствует тип `Derivative_t`, параметры которого рассмотрим позже). Функция `Print` обеспечивает преобразование целочисленной константы в строку, а функция `Eval`, обеспечивающая вычисление значения узла в данной точке, возвращает значение этой константы. Параметры функции `Eval` рассмотрим далее.

3.2 Свободные переменные

Для определения переменной также введем вспомогательный тип:

```
template<char Family, int Index>
struct Variable {};
```

где `Family` и `index` — «семейство» и индекс переменной. Так, например, для w_0 они равны, соответственно, `'w'` и `0`.

Рассмотрим определение узла для свободной переменной:

```
template<char Family, int Index>
struct Node<Variable<Family, Index>>
{
  template<char FPrime, int IPrime>
  using Derivative_t = std::conditional_t<FPrime == Family && IPrime == Index,
      Node<Number<1>>,
      Node<Number<0>>>;

  static std::string Print ()
  {
    return std::string { Family, '_' } + std::to_string (Index);
  }

  template<typename Vec>
  static typename Vec::value_type Eval (const Vec& values)
  {
    return values (Node {});
  }

  constexpr Node () {}
};
```

Так, производная переменной равна единице, если производная берется по самой этой переменной, и нулю иначе. Собственно, параметры `FPrime` и `IPrime` типа `Derivative_t` соответствуют семейству и индексу переменной, по которой вычисляется производная.

Функция `Eval`, вычисляющая значение переменной, создает безымянный объект того же типа, что и узел, и вызывает переданный объект `values` с этим типом. Соответствующая перегрузка `operator()` у типа `Vec` обеспечивает выбор значения, соответствующего типу `Node`. Детальная реализация типа `Vec` рассмотрена далее.

Отметим, что известный на этапе компиляции тип `Node` и, как следствие, известная на этапе компиляции перегрузка `Vec::operator()` обеспечивают компилятору возможность

прямо подставить соответствующее значение переменной (пусть и известное лишь на этапе выполнения), как если бы оно было указано вручную.

3.3 Унарные функции

Для указания конкретной унарной функции вводится перечисление `UnaryFunction` и соответствующий тип `UnaryFunctionWrapper`, а также некоторые синонимы типов для удобства дальнейшего использования:

```
enum class UnaryFunction
{
  Sin,
  Cos,
  Ln,
  Neg
};

template<UnaryFunction UF>
struct UnaryFunctionWrapper;

using Sin = UnaryFunctionWrapper<UnaryFunction::Sin>;
using Cos = UnaryFunctionWrapper<UnaryFunction::Cos>;
using Neg = UnaryFunctionWrapper<UnaryFunction::Neg>;
using Ln = UnaryFunctionWrapper<UnaryFunction::Ln>;
```

В специализациях `UnaryFunctionWrapper` реализована логика взятия производных каждой конкретной унарной функции:

```
template<>
struct UnaryFunctionWrapper<UnaryFunction::Sin>
{
  template<typename Child>
  using Derivative_t = Node<Cos, Child>;
};

template<>
struct UnaryFunctionWrapper<UnaryFunction::Cos>
{
  template<typename Child>
  using Derivative_t = Node<Neg, Node<Sin, Child>>;
};

template<>
struct UnaryFunctionWrapper<UnaryFunction::Ln>
{
  template<typename Child>
  using Derivative_t = Node<Div, Node<Number<1>>, Child>;
};

template<>
struct UnaryFunctionWrapper<UnaryFunction::Neg>
{
  template<typename>
  using Derivative_t = Node<Number<-1>>;
};
```

При этом дальнейшее дифференцирование дочернего узла производится соответствующей специализацией `Node`:

```
template<UnaryFunction UF, typename... ChildArgs>
struct Node<UnaryFunctionWrapper<UF>, Node<ChildArgs...>>
{
  using Child_t = Node<ChildArgs...>;
```

```
  template<char FPrime, int IPrime>
  using Derivative_t = Node<Mul,
      typename UnaryFunctionWrapper<UF>::template Derivative_t<Child_t>,
      typename Node<ChildArgs...>::template Derivative_t<FPrime, IPrime>>;

  static std::string Print ()
  {
    return FunctionName (UF) + "(" + Node<ChildArgs...>::Print () + ")";
  }

  template<typename Vec>
  static typename Vec::value_type Eval (const Vec& values)
  {
    const auto child = Child_t::Eval (values);
    return EvalUnary (UnaryFunctionWrapper<UF> {}, child);
  }
};
```

Соответственно, определение типа `Derivative_t` для данной специализации `Node` вычисляет производную унарной функции согласно `UnaryFunctionWrapper`, рекурсивно вычисляет производную дочернего узла, и затем перемножает их согласно правилу взятия производых сложной функции. Подобное разделение логики взятия производных позволяет избежать дублирования кода.

Функция `Eval` вычисляет значение дочернего узла, а затем вызывает соответствующую перегрузку `EvalUnary`. Определения перегрузок последней достаточно тривиальны:

```
template<typename T>
T EvalUnary (const Sin&, T value)
{
  return std::sin (value);
}

template<typename T>
T EvalUnary (const Cos&, T value)
{
  return std::cos (value);
}

template<typename T>
T EvalUnary (const Ln&, T value)
{
  return std::log (value);
}

template<typename T>
T EvalUnary (const Neg&, T value)
{
  return -value;
}
```

Отметим, что операцию `Neg` можно и не вводить, заменяя ее на умножение на минус единицу.

3.4 Бинарные функции

Представление бинарных функций аналогично унарным. Для полноты введем используемые далее типы:

```
enum class BinaryFunction
{
  Add,
  Mul,
  Div,
```

```
  Pow
};

using Add = BinaryFunctionWrapper<BinaryFunction::Add>;
using Mul = BinaryFunctionWrapper<BinaryFunction::Mul>;
using Div = BinaryFunctionWrapper<BinaryFunction::Div>;
using Pow = BinaryFunctionWrapper<BinaryFunction::Pow>;
```

3.5 Дополнительные определения

Приведем объявления, позволяющие приблизить запись формул к обычной математической.

Некоторые часто используемые имена свободных переменных:

```
template<char Family, int Index = 0>
constexpr Node<Variable<Family, Index>> Var {};

using X0 = Node<Variable<'x', 0>>;
constexpr X0 x0;
using X1 = Node<Variable<'x', 1>>;
constexpr X1 x1;
using X2 = Node<Variable<'x', 2>>;
constexpr X2 x2;

using W0 = Node<Variable<'w', 0>>;
constexpr W0 w0;
using W1 = Node<Variable<'w', 1>>;
constexpr W1 w1;
using W2 = Node<Variable<'w', 2>>;
constexpr W2 w2;
```

Константа, соответствующая единице:

```
constexpr Node<Number<1>> _1;
```

Операторы и унарные функции для более естественной записи выражений:

```
template<typename T1, typename T2>
Node<Add, std::decay_t<T1>, std::decay_t<T2>> operator+ (T1, T2);

template<typename T1, typename T2>
Node<Mul, std::decay_t<T1>, std::decay_t<T2>> operator* (T1, T2);

template<typename T1, typename T2>
Node<Div, std::decay_t<T1>, std::decay_t<T2>> operator/ (T1, T2);

template<typename T1, typename T2>
Node<Add, std::decay_t<T1>, Node<Neg, std::decay_t<T2>>> operator- (T1, T2);

template<typename T>
Node<Sin, std::decay_t<T>> Sin (T);

template<typename T>
Node<Cos, std::decay_t<T>> Cos (T);

template<typename T>
Node<Ln, std::decay_t<T>> Ln (T);
```

Отметим, что функциям и операторам не нужно определение, достаточно объявления, так достаточно лишь указания возвращаемого типа соответствующих функций.

4 Вычисление производных

Вычисление производных также производится на уровне типов. При этом соответствующая метафункция `VarDerivative_t` лишь вызывает `Derivative_t` у переданного ей узла:

```
template<typename Node, typename Var>
struct VarDerivative;

template<typename Expr, char Family, int Index>
struct VarDerivative<Expr, Node<Variable<Family, Index>>>
{
  using Result_t = typename Expr::template Derivative_t<Family, Index>;
};

template<typename Node, typename Var>
using VarDerivative_t = typename VarDerivative<Node, std::decay_t<Var>>::Result_t;
```

5 Использование зависимых переменных

Для оптимизации скорости вычислений иногда имеет смысл кэшировать результаты некоторых сложных операций (таких, как, например, взятие логарифма) над свободными переменными. Получающиеся при этом переменные имеют зависимость от исходных свободных, и это необходимо учитывать при взятии соответствующих производных.

Для этого предлагается преобразовывать выражения, заменяя поддеревья, отвечающие за сложные операции, на зависимые переменные во время вычисления значения выражения, и выполняя обратную замену при взятии производной.

Так, например, если в формуле часто фигурирует выражение $\log \omega_0$, где ω_0 — редко изменяемый параметр, то имеет смысл ввести дополнительную переменную $\tilde{\omega_0} = \log \omega_0$ и переписать выражение в ее терминах.

Для этого определим метафункцию `ApplyDependency`, рекурсивно спускающуюся по дереву выражения и подменяющую элемент дерева, соответствующий шаблону, следующим образом:

1. Если рассматриваемый узел совпадает с шаблоном, то необходимо его заменить на данное выражение.
2. Иначе, если рассматриваемый узел — унарная функция, необходимо рекурсивно спуститься на дочерний узел.
3. Иначе, если рассматриваемый узел — бинарная функция, необходимо рекурсивно спуститься на оба дочерних узла.
4. Иначе ничего не делать.

Последнему случаю соответствует базовая реализация ApplyDependency, а оставшимся вариантам — специализации шаблона. Так как специализации шаблонов рассматриваются не по порядку, а все вместе, необходимо отключать специализации для второго и третьего случаев при совпадении шаблона с рассматриваемым узлом при помощи техники SFINAE [4]:

```
template<typename Var, typename Expr, typename Formula, typename Enable = void>
struct ApplyDependency
{
  using Result_t = Formula;
```

```
};

template<typename Var, typename Expr, typename Formula>
using ApplyDependency_t = typename ApplyDependency<std::decay_t<Var>,
        std::decay_t<Expr>, Formula>::Result_t;

template<
    typename Var,
    typename Expr,
    UnaryFunction UF,
    typename Child
  >
struct ApplyDependency<Var,
    Expr,
    Node<UnaryFunctionWrapper<UF>, Child>,
    std::enable_if_t<!std::is_same<Var,
        Node<UnaryFunctionWrapper<UF>, Child>>::value>>
{
  using Result_t = Node<
        UnaryFunctionWrapper<UF>,
        ApplyDependency_t<Var, Expr, Child>
      >;
};

template<
    typename Var,
    typename Expr,
    BinaryFunction BF,
    typename FirstNode,
    typename SecondNode
  >
struct ApplyDependency<Var,
    Expr,
    Node<BinaryFunctionWrapper<BF>, FirstNode, SecondNode>,
    std::enable_if_t<!std::is_same<Var,
        Node<BinaryFunctionWrapper<BF>, FirstNode, SecondNode>>::value>>
{
  using Result_t = Node<
        BinaryFunctionWrapper<BF>,
        ApplyDependency_t<Var, Expr, FirstNode>,
        ApplyDependency_t<Var, Expr, SecondNode>
      >;
};

template<typename Var, typename Expr>
struct ApplyDependency<Var, Expr, Var>
{
  using Result_t = Expr;
};
```

6 Передача параметров

Отметим, что каждая переменная имеет свой собственный тип, что дает возможность выбора соответствующего значения при помощи выбора соответствующей перегрузки некоторой функции из данного семейства. Для построения этого семейства предлагается следующий подход.

Каждая конкретная функция из семейства задается лямбда-выражением. Так, для построения функци, возвращающей для типа `NodeType`, известного на момент компиляции, задаваемое в процессе выполнения значение `val` типа `ValueType`, используется следующая конструкция:

```
template<typename ValueType, typename NodeType>
auto BuildFunctor (NodeType, ValueType val)
{
    return [val] (NodeType) { return val; };
}
```

Для того, чтобы скомбинировать несколько подобных функций, предлагается воспользоваться тем, что лямбда-выражения в языке C++ представляют собой сгенерированные компилятором структуры которые, тем не менее, могут выступать в роли базовых классов. Тогда для объединения нескольких лямбда-выражений можно использовать следующий вспомогательный тип:

```
template<typename F, typename S>
struct Map : F, S
{
  using F::operator();
  using S::operator();

  Map (F f, S s)
  : F { std::forward<F> (f) }
  , S { std::forward<S> (s) }
  {
  }
};
```

где `F` и `S` — прозвольные объекты, имеющие конструктор копирования и оператор вызова (`operator()`).

Отметим, что тип `Map<F, S>` также можно использовать как один из аргументов к типу `Map`. Иными словами, `Map` порождает моноид на пространстве объектов с переопределенным `operator()`, где единичным элементом является объект с оператором вида `void operator() ()` .

С учетом этого для построения объекта, передаваемого в вышеописанные фукнции `Eval`, можно использовать следующий рекурсивный механизм:

```
template<typename F>
auto Augment (F&& f)
{
  return f;
}

template<typename F, typename S>
auto Augment (F&& f, S&& s)
{
  return Map<std::decay_t<F>, std::decay_t<S>> { f, s };
}

template<typename ValueType>
auto BuildFunctor ()
{
  struct
  {
    ValueType operator() () const
    {
      return {};
    }

    using value_type = ValueType;
  } dummy;
  return dummy;
}

template<typename ValueType, typename NodeType, typename... Tail>
auto BuildFunctor (NodeType, ValueType val, Tail&&... tail)
```

```
{
  return detail::Augment ([val] (NodeType) { return val; },
      BuildFunctor<ValueType> (std::forward<Tail> (tail)...));
}
```

Данный подход, кроме того, позволяет гарантировать полноту и единственность задания аргументов на этапе компиляции: ошибкой компиляции является как отсутствие значения для какой-либо из переменных, так и несколько раз указанное значение для одной и той же переменной.

7 Вычислительный эксперимент

Предложенный метод применен для вычисления производных регрессионной модели, описывающей мощность лазера как функцию прозрачности его резонатора [1]. Функция имеет следующий вид:

$$y(R_0) = \gamma \frac{1 - R_0}{1 + R_0} \Big(\frac{g_0}{\alpha_0 - \frac{1}{2} L \ln R_0} - 1 \Big),$$

где L — некоторая константа, R_0 — свободная переменная, а α_0, g_0 и γ — параметры регрессионной модели, которые необходимо оценить, и которые в контексте данной работы также можно считать свободными переменными.

Для удобства записи этой формулы в виде кода переименуем стандартные переменные так, чтобы они соответствовали предметной области (считая, что $k = \gamma$):

```
auto g0 = w0;
auto alpha0 = w1;
auto k = w2;
auto r0 = x0;
auto logr0 = x1;
auto r0ppsq = x2;
```

где `logr0` и `r0ppsq` представляют кешированные подвыражения.

Тогда искомую формулу, регрессионный остаток и производные можно записать следующим образом:

```
using Formula_t = decltype (k * (_1 - r0) / (_1 + r0) *
          (g0 / (alpha0 - logr0 / Num<300>) - _1));

const auto& params = Params::BuildFunctor<float> (g0, someValue,
        alpha0, anotherValue,
        k, yetOneMoreValue,
        r0, independentVariable,
        logr0, logOfTheIndependentVariable);

const auto residual = Formula_t::Eval (params) - knownValue;
const auto dg0 = VarDerivative_t<Formula_t, decltype (g0)>::Eval (params);
const auto dalpha0 = VarDerivative_t<Formula_t, decltype (alpha0)>::Eval (params);
const auto dk = VarDerivative_t<Formula_t, decltype (k)>::Eval (params);
```

Производная по R_0 требует подстановки значения $\log R_0$:

```
using Unwrapped_t = ApplyDependency_t<decltype (logr0), decltype (Ln (r0)), Formula_t>;
using Derivative_t = VarDerivative_t<Unwrapped_t, decltype (r0)>;
using CacheLog_t = ApplyDependency_t<decltype (Ln (r0)), decltype (logr0), Derivative_t>;
using CachedSq_t = ApplyDependency_t<decltype ((-k * (_1 + r0) - (k * (_1 - r0))) /
          ((_1 + r0) * (_1 + r0))),
      decltype (k * r0ppsq),
      CacheLog_t>;
const auto dr0 = CachedSq_t::Eval (params);
```

Предпоследнее выражение для `CachedSq_t` обеспечивает замену соответствующего подвыражения в формуле на рассчитанное заранее значение `r0ppsq` и, формально, не является необходимым.

Сравнение производительности выполнялось при оценке эмпирического распределения параметров регрессионной модели методом Монте-Карло [5]. Вычислительная программа компилировалась и выполнялась с использованием компилятора clang 3.9 для архитектуры amd64 под ОС Linux на процессоре Intel Core i7-3930K с флагами `-march=native -O3 -ffast-math`. Время Монте-Карло-симуляции, вычислявшей приблизительно $212 \cdot 10^9$ производных за все время своей работы, и выполнявшейся в 12 потоков (с учетом виртуальных HT-ядер процессора) по результатам десяти запусков, составило 531 секунду, стандартное отклонение — 4.2 с. Аналогичная симуляция, использовавшая вручную написанный и оптимизированный код, выполнялась за 529 секунд, стандартное отклонение — 5.0 с.

Непосредственное время вычисления производных в обоих случаях составляло около 35% времени работы алгоритма. Таким образом, разница во времени работы оказалась незначимой.

8 Заключение

Предложен метод символьного вычисления производных во время компиляции средствами шаблонов C++. Метод позволяет сократить объем кода, требуемый для описания соответствующей функции и ее производных, уменьшить риск возникновения ошибок и опечаток при кодировании выражений для вычисления производных, а также гарантировать, что при изменении исходной функции соответствующим образом изменятся и все зависимые производные.

За счет выполнения дифференцирования во время компиляции производительность сгенерированного кода близка к таковой для вручную написанного и оптимизированного кода, что позволяет использовать предложенный метод в требовательных к производительности областях.

Список литературы

[1] Rudoy, G. I.: *Analysis of the stability of nonlinear regression models to errors in measured data.* Pattern Recognition and Image Analysis, 26(3):608–616, 2016.

[2] Neidinger, Richard D: *Introduction to automatic differentiation and matlab object-oriented programming.* SIAM review, 52(3):545–563, 2010.

[3] Rudoy, G. I.: *IAmMad*, 2016. `https://github.com/0xd34df00d/IAmMad`.

[4] Josuttis, Nicolai M: *C++ Templates: The Complete Guide.* Addison-Wesley Professional, 2003.

[5] Рудой, Г. И.: *О возможности применения методов Монте-Карло в анализе нелинейных регрессионных моделей.* Сибирский Журнал Вычислительной Математики, 4:425–434, 2015.