ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 4 за 2017 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

Краснов М.М.

Применение символьного дифференцирования для решения ряда вычислительных задач

Рекомендуемая форма библиографической ссылки: Краснов М.М. Применение символьного дифференцирования для решения ряда вычислительных задач // Препринты ИПМ им. М.В.Келдыша. 2017. № 4. 24 с. doi:10.20948/prepr-2017-4
URL: http://library.keldysh.ru/preprint.asp?id=2017-4

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

М.М. Краснов

Применение символьного дифференцирования для решения ряда вычислительных задач

Москва — 2017

Краснов М.М.

Применение символьного дифференцирования для решения ряда вычислительных задач

На примере метода Ньютона решения нелинейных уравнений и других подобных задач демонстрируется применение символьного дифференцирования на языке C++. В основе предлагаемого подхода лежит метапрограммирование языка C++, в частности шаблоны выражений. Рассматриваются решение уравнения с одной переменной, системы уравнений с несколькими переменными, задача нахождения минимума одной функции с несколькими переменными и другие задачи.

Ключевые слова: символьное дифференцирование, метапрограммирование, метод Ньютона

Mikhail Mikhailovich Krasnov

Application of symbolic differentiation for solving of some computational problems

On the example of the Newton’s method of solving of non-linear equations and other similar problems the usage of the symbolic differentiation in the C++ programming language is demonstrated. The proposed approach is based on the metaprogramming of the C++ language, in particular, on expression templates. The problem of solving of a single equation with one variable, of a system of equations with multiple variables, the problem of finding of a minimum of an equation with multiple variables and other similar problems are considered.

Key words: symbolic differentiation, metaprogramming, Newton's method

Оглавление

1. Введение

При решении вычислительных задач для нахождения значений производных разных порядков в точке могут применяться разные способы дифференцирования: численное, автоматическое, символьное. Все они сильно отличаются друг от друга и их ни в коем случае нельзя путать.

При численном дифференцировании задаётся регулярная прямоугольная сетка с некоторым шагом (постоянным или переменным) в каждом направлении пространства. В каждом узле сетки задаётся значение функции, и значение производной в узле сетки вычисляется в соответствии с некоторым шаблоном производной нужного порядка точности исходя из значений функции в данном и соседних узлах и шага сетки. Например, значение первой производной первого порядка точности вычисляется как $df_k=(f_{k+1}-f_k)/(x_{k+1}-x_k)$, а второго порядка точности – как $df_k=(f_{k+1}-f_{k-1})/(x_{k+1}-x_{k-1})$. Здесь x_k – это координаты узлов сетки, а f_k – значения функции в этих узлах. Вторая производная первого порядка точности обычно вычисляется по формуле $d2f_k=(f_{k+1}+f_{k-1}-2f_k)/\Delta x^2$, где Δx – шаг сетки (постоянный).

При автоматическом дифференцировании речь обычно идёт совсем о другом. Пусть также нам задана некоторая сетка (теперь уже неважно, регулярная или нет) и в каждом узле сетки в виде сеточной функции заданы значение некоторой функции и её производная, возможно, не одна, а по разным (вообще говоря, обобщённым) переменным. Пусть далее нам по этой сеточной функции нужно вычислить другую сеточную функцию (также хранящую значения функции и её производных в узлах сетки) в соответствии с некоторой формулой, например, $g=\sin(\pi f)$. Тогда в каждом узле сетки мы можем легко вычислить значение производной новой функции, в нашем примере по формуле $g'=\cos(\pi f)\cdot\pi f'$. Если нужно вычислять не только первую производную, но и производные более высоких порядков (например, вторую), то также

принципиальных проблем нет. В этом случае наша исходная сеточная функция должна хранить все требуемые производные нужных порядков, и требуемые производные новой сеточной функции могут быть легко по ним вычислены. Проблема состоит в том, что формула преобразования может быть произвольной, и для каждой такой формулы нужно определить формулы для нужных производных, которые в случае производных высоких порядков могут стать весьма сложными. Когда говорят об автоматическом дифференцировании, речь может идти, в частности, о том, чтобы эти формулы определялись автоматически по исходной формуле преобразования. Предлагаемые в данной работе идеи, основанные на использовании метапрограммирования языка C++ (см. [1]) для символьного дифференцирования, вполне могут быть использованы для автоматического дифференцирования при определении формул для производных.

При символьном дифференцировании обычно никакой сетки и, соответственно, никаких сеточных функций нет. Постановка задачи может быть, например, такой. Имеется исходный набор базовых переменных (например, давление, плотность, температура, время). По этим переменным в соответствии с некоторыми формулами задаётся набор основных переменных (например, энергия, энтропия). Ну и, наконец, некоторыми формулами задаётся набор конечных переменных, причём эти формулы могут содержать как значения базовых и основных переменных, так и значения частных производных разных порядков (обычно не выше второго) основных переменных по базовым переменным. Требуется по заданным значениям базовых переменных вычислить значения конечных переменных.

Решение задачи в такой постановке не представляет большой трудности и будет изложено в конце статьи. Вначале покажем решение других (похожих между собой) задач методом Ньютона. Первая – решение нелинейного уравнения с одной неизвестной. Вторая – решение системы нелинейных уравнений с несколькими неизвестными. И третья – решение задачи поиска минимума (или максимума) для одного уравнения с несколькими неизвестными.

Материал статьи излагается применительно к стандарту языка C++ 2011 года, т.н. C++0x. Он поддерживается современными версиями большинства компиляторов, в том числе gnu C++ версии 4.9 и выше, Intel C++ compiler версии 2013 года и выше, Microsoft Visual Studio 2013 и выше, nvcc версии 7.0 и выше. Можно скомпилировать код и более ранними версиями компиляторов, но тогда некоторые конструкции (в основном касающиеся метапрограммирования) придётся брать из библиотеки boost (см. [2]).

Автор выражает благодарность Феодоритовой Ольге Борисовне, сотруднице нашего отдела № 8 ИПМ им. М.В. Келдыша РАН за то, что она сподвигла меня написать данную статью и предоставила литературу, касающуюся метода Ньютона, а также Савенкову Евгению Борисовичу и Балашову Владиславу Александровичу за предоставленную модельную задачу.

2. Шаблоны выражений

Зададимся вопросом: что значит «задать формулу»? При этом мы хотим по одним формулам получать другие формулы, например, по формуле получать её производную, которая также является формулой, и от неё, в свою очередь, также можно получить производную в виде формулы. Ясно, что, например, функции для задания формул не годятся.

В языке C++ есть достаточно распространённый механизм для задания выражений под названием «шаблоны выражений» (expression templates) (см. [3]). С помощью этого механизма можно, оставаясь в рамках языка, определить язык в соответствие с некоторой грамматикой. При этом выражение на этом языке будет оставаться корректным выражением и на языке C++. Такой язык носит название «предметно-ориентированный язык» (domain specific language, DSL). Определим для формул такой язык и реализуем его с помощью механизма шаблонов выражений. Вначале определим грамматику нашего языка выражений:

«выражение»::=«сумма» | *-«терм»* | *+«терм»;*
«сумма»::=«слагаемое» |
«слагаемое»+«выражение» |
«слагаемое»-«выражение»;
«слагаемое»::=«терм» |
«терм»«слагаемое»* |
«терм»/«слагаемое»;
«терм»::=«константа» | *«переменная»* |
«имя-функции»(«выражение») | *(«выражение»);*

Данная грамматика является корректной, но сложноватой. Для наших нужд больше подойдёт менее строгая, но более простая следующая грамматика:

«выражение»::=«константа» | *«переменная»* |
«выражение»+«выражение» | *«выражение»-«выражение»* |
«выражение»«выражение»* | *«выражение»/«выражение»* |
«имя-функции»(«выражение») | *(«выражение»)* | *-«выражение»;*

Разбор выражения будет производить компилятор, в частности он в соответствии со своими правилами будет обрабатывать скобки, старшинство операций и левоассоциативность арифметических операций. Более простая грамматика будет позволять, например, такие выражения, как x+-y, которое не является корректным выражением. Но так как мы реализуем язык в рамках языка C++ (т.е. выражение, корректное в нашем языке, должно быть также корректным выражением на языке C++), то такие ошибочные формулы будет отсекать сам компилятор.

Прежде чем мы реализуем этот простой язык для наших формул, рассмотрим ещё один механизм языка C++ – шаблонный полиморфизм. Он основан на шаблоне проектирования под названием Curiously recurring template pattern, CRTP (см. [4]). Как известно, при появлении языка C++, когда в языке ещё не было шаблонов, полиморфизм реализовывался с помощью механизма виртуальных функций. С появлением шаблонов появился другой способ его реализации – т.н. шаблонный полиморфизм. Основная его идея состоит в том, что в базовый класс в качестве параметра шаблона базового класса передаётся конечный класс (в этом и состоит та самая «забавная рекурсия»). После этого ссылку на базовый класс легко преобразовать в ссылку на конечный класс и получить доступ ко всем методам и переменным конечного класса. Покажем, как это делается:

```
template<class T>
struct math_object_base {
      T& self(){
            return static_cast<T&>(*this);
      }
      const T& self() const {
            return static_cast<const T&>(*this);
      }
};
```

Этот класс (math_object_base) будем делать базовым классом для всех наших классов и передавать ему в качестве параметра шаблона конечный класс. Точнее будет сказать, что math_object_base – это не класс, а шаблон класса. Класс получается при подстановке конкретного значения параметра шаблона. Таким образом, у каждого конечного класса будет свой уникальный базовый класс. Такой полиморфизм по сути является статическим. Для каждого конечного класса компилятором генерируется свой базовый класс, а для полиморфных функций и методов – свой экземпляр функции или метода. Это может увеличить время компиляции, но при современном быстродействии процессоров это увеличение можно считать несущественным.

Метод self базового класса делает то самое упомянутое выше приведение ссылки на базовый класс к ссылке на конечный класс. Благодаря «забавной рекурсии» этот метод может быть реализован в самом базовом классе.

Следующим шагом определим класс выражения:

```
template<class E>
struct expression : math_object_base<E>{};
```

Этот класс не несёт в себе никакого функционала и является чисто маркерным. Он также принимает в качестве параметра шаблона конечный класс и передаёт его дальше в свой базовый класс. Все наши дальнейшие классы будут наследоваться непосредственно от этого класса expression.

3. Реализация выражений

В соответствии с нашей упрощённой грамматикой мы должны реализовать константы, переменные, сумму, разность, произведение и частное выражений, отрицание выражения и вызов функции от выражения. Это означает, что для каждого из перечисленных понятий мы должны реализовать свой класс, пронаследованный от класса expression, что и будет означать, что они являются выражениями. От каждого конкретного выражения мы хотим, чтобы оно могло вычислить своё значение от заданного значения переменной (или набора переменных в случае, если мы работаем с выражениями от нескольких переменных) и возвращать выражение для производной (или частной производной в случае нескольких переменных). В принципе ещё может быть интересным преобразование выражения в строку, но так как для символьного дифференцирования это не нужно, то мы не будем этого делать, чтобы не перегружать изложение материала. При необходимости добавить этот функционал не представляется сложным.

Начнём для простоты со случая выражений от одной переменной. Обобщение на несколько переменных сделаем позже. Начнём с самого простого – констант. Тут нужно сделать такое замечание. Хотелось бы как можно больше вычислений вынести на стадию компиляции. Но на стадии компиляции передавать в качестве параметров шаблонов и вычислять методами метапрограммирования можно только целочисленные константы. Поэтому целочисленные константы, значения которых известны уже на стадии компиляции, выделим в отдельный случай. Константы произвольных типов обработаем отдельно. Итак, вот класс для целочисленных констант:

```
template<int N>
struct int_constant : expression<int_constant<N> >{
      static const int value = N;
      typedef int_constant<0> diff_type;

      diff_type diff() const {
            return diff_type();
      }
      template<typename T>
      int operator()(const T &x) const {
            return value;
      }
};

template<class E>
struct is_int_constant : std::false_type{};

template<int N>
struct is_int_constant<int_constant<N> > : std::true_type{};
```

```
template<class E>
struct int_constant_value : std::integral_constant<int, 0>{};

template<int N>
struct int_constant_value<int_constant<N> > :
    std::integral_constant<int, N>{};
```

В нашем первом классе int_constant определены три вещи. Во-первых, тип diff_type. Этот тип задаёт тип выражения для производной. Так как производная любой константы равна нулю, то, естественно, это int_constant<0>. Во-вторых, метод diff, возвращающий выражение для производной, и, в-третьих, функциональный оператор, вычисляющий значение выражения от заданного значения переменной. В случае константы значение выражения от значения переменной не зависит и всегда равно значению этой константы. Во всех наших следующих классах мы также будет определять все эти три вещи.

Кроме того, мы определяем две метафункции (функции времени компиляции). Первая – is_int_constant, принимающая в качестве параметра произвольный тип и отвечающая на вопрос «является ли этот тип типом int_constant». Эта метафункция нам понадобится в дальнейшем для оптимизации метавычислений. Вторая метафункция int_constant_value также принимает в качестве параметра произвольный тип и для типа int_constant возвращает значение, которое этот тип хранит, а для остальных типов возвращает ноль.

Следующий случай – константа произвольного типа. Это может быть и целочисленный тип, например, для случая, когда значение константы становится известным только во время исполнения. Назовём класс для констант произвольного типа scalar:

```
template<typename VT>
struct scalar : expression<scalar<VT> >{
     typedef VT value_type;
     typedef int_constant<0> diff_type;

     scalar(const value_type &value) : value(value){}
     diff_type diff() const {
          return diff_type();
     }
     template<typename T>
     value_type operator()(const T &x) const {
          return value;
     }
     const value_type value;
};

template<class E>
struct is_scalar : std::false_type {};

template<typename VT>
```

```
struct is_scalar<scalar<VT> > : std::true_type {};

template<typename T>
scalar<T> _(const T &val) {
	return scalar<T>(val);
}
```

Производная константы любого типа равна (целочисленному) нулю. Остальное должно быть понятно. Из первых двух примеров выражений уже видно, как любое выражение может вернуть выражение для своей производной, причём тип выражения для производной может отличаться от типа самого выражения (как в случае типа scalar).

Так же, как и для класса int_constant, мы определяем метафункцию is_scalar, принимающую в качестве параметра произвольный тип и отвечающую на вопрос «является ли этот тип типом scalar». Кроме того, мы определяем шаблонную функцию времени исполнения с именем «_» (подчерк), принимающую значение любого типа и возвращающую скаляр этого типа с переданным значением.

Далее опишем переменную:

```
struct variable : expression<variable>
{
	typedef int_constant<1> diff_type;

	diff_type diff() const {
		return diff_type();
	}

	template<typename T>
	T operator()(const T &x) const {
		return x;
	}
};
```

Здесь тоже должно быть всё понятно. Производная переменной по самой себе равна единице. Следующий по возрастанию сложности случай – отрицание выражения. Вот текст класса negate_expression:

```
template<class E>
struct negate_expression;

template<class E>
struct negate_expression_type {
	typedef typename std::conditional<
		is_int_constant<E>::value,
		int_constant<-int_constant_value<E>::value>,
		typename std::conditional<
			is_scalar<E>::value, E, negate_expression<E>
		>::type
```

```
      >::type type;
};

template<class E>
struct negate_expression : expression<negate_expression<E> >{
      typedef typename negate_expression_type<
            typename E::diff_type>::type diff_type;

      negate_expression(const expression<E> &e) : e(e.self()){}

      diff_type diff() const {
            return -e.diff();
      }

      template<typename T>
      T operator()(const T &x) const {
            return -e(x);
      }

      const E e;
};
template<class E>
negate_expression<E>
operator-(const expression<E>& e){
      return negate_expression<E>(e);
}
template<int N>
int_constant<-N>
operator-(const int_constant<N>&){
      return int_constant<-N>();
}
template<typename VT>
scalar<VT>
operator-(const scalar<VT> &e){
      return scalar<VT>(-e.value);
}
```

Этот класс интересен тем, что в нём в первый раз мы делаем оптимизацию результата. А именно: если происходит отрицание константы (целочисленной или любого типа), то результатом такого отрицания становится также константа того же типа, но со значением, равным отрицанию значения исходной константы. Т.е. унарный оператор «-» в зависимости от типа аргумента возвращает результаты разных типов. Для того чтобы можно было узнать, какого типа результат вернёт унарный оператор «-», имеется метафункция negate_expression_type, которая в качестве параметра принимает тип произвольного выражения и в типе type возвращает тип, который вернёт унарный оператор «-» для этого типа. В этой метафункции используется условный оператор времени компиляции, реализуемый метафункцией из стандартной библиотеки языке C++ std::conditional. Метафункция std::conditional

в качестве параметров шаблона принимает булевскую константу (true или false) и два типа. В типе type эта метафункция возвращает первый из этих двух типов в случае, если булевская константа равна true, и второй тип в противном случае. Метафункция negate_expression_type сравнивает переданный в качестве параметра шаблона тип с типами int_constant и scalar (с помощью описанных выше метафункций is_int_constant и is_scalar) и эти случаи обрабатывает особым образом, в противном же случае (общий случай) возвращает класс negate_expression.

Класс negate_expression использует метафункцию negate_expression_type для определения типа производной (тип diff_type). В качестве параметра передаётся тип производной отрицаемого выражения.

Далее опишем реализацию суммы и разности выражений. Операции сложения и вычитания по своей реализации очень похожи, поэтому эти две операции реализуются с помощью одного класса, additive_expression. Для того чтобы отличить сложение от вычитания, в этот класс передаётся дополнительный параметр шаблона типа char. Для сложения в качестве значения этого параметра передаётся ‘+’, а для вычитания – ‘-’. Вот текст этого класса:

```
template<char op, class E1, class E2>
typename std::enable_if<op == '+',
     typename additive_expression_type<typename E1::diff_type, op,
          typename E2::diff_type>::type
>::type
additive_expression_diff(const E1& e1, const E2& e2){
     return e1.diff() + e2.diff();
}

template<char op, class E1, class E2>
typename std::enable_if<op == '-',
     typename additive_expression_type<typename E1::diff_type, op,
          typename E2::diff_type>::type
>::type
additive_expression_diff(const E1& e1, const E2& e2){
     return e1.diff() - e2.diff();
}

template<class E1, char op, class E2>
struct additive_expression : expression<additive_expression<E1, op, E2> >
{
     typedef typename additive_expression_type<typename E1::diff_type,
          op, typename E2::diff_type>::type diff_type;

     additive_expression(const expression<E1> &e1,
          const expression<E2> &e2) : e1(e1.self()), e2(e2.self()){}

     diff_type diff() const {
          return additive_expression_diff<op>(e1, e2);
```

```
	}

	template<typename T>
	typename std::enable_if<op == '+', T>::type
	operator()(const T &x) const {
		return e1(x) + e2(x);
	}

	template<typename T>
	typename std::enable_if<op == '-', T>::type
	operator()(const T &x) const {
		return e1(x) - e2(x);
	}

	const E1 e1;
	const E2 e2;
};

template<class E1, class E2>
additive_expression<E1, '+', E2>
operator+(const expression<E1> &e1, const expression<E2> &e2){
	return additive_expression<E1, '+', E2>(e1, e2);
}

template<class E1, class E2>
additive_expression<E1, '-', E2>
operator-(const expression<E1> &e1, const expression<E2> &e2){
	return additive_expression<E1, '-', E2>(e1, e2);
}

template<class E>
const E& operator+(const expression<E> &e, const int_constant<0>&){
	return e.self();
}

template<int N1, int N2>
int_constant<N1 + N2>
operator+(const int_constant<N1>&, const int_constant<N2>&){
	return int_constant<N1 + N2>();
}
```

Исходный код для сложения и вычитания показан далеко не полностью, он включает многочисленные оптимизации, такие как, например, прибавление или вычитание нуля (результат не меняется), сложение и вычитание двух целочисленных констант (результат – также целочисленная константа). Метафункция additive_expression_type также не показана. Из-за многочисленных оптимизаций её тело довольно большое. Для дифференцирования приходится использовать вспомогательный класс additive_expression_diff.

Из новых вещей следует обратить внимание на активное использование в реализации сложения и вычитание одной из важных идиом языка C++ - т.н. SFINAE, substitution failure is not an error (неудача при подстановке не является ошибкой) (см. [5]). Эта идиома используется в метафункции из стандартной библиотеки языка std::enable_if. Эта метафункция принимает два параметра шаблона – булевскую константу и тип. Если булевская константа равна true, то enable_if в типе type возвращает переданный во втором параметре тип, иначе ничего не возвращает. В этом случае (когда булевская константа равна false) использование возвращаемого типа type для задания типа возвращаемого функцией значения не приводит к ошибке компиляции, вместо этого функция просто становится невидимой. Тело функции в этом случае проверяется только на синтаксическую правильность.

Реализация умножения и деления осуществляется аналогично реализации сложения и вычитания и также включает многочисленные оптимизации (например, умножения на ноль в результате также даёт ноль независимо от типа второго операнда). Имеется также степенная функция pow, принимающая целочисленную степень, в которую возводится выражение, в качестве параметра шаблона, что делает известным этот показатель степени уже во время компиляции и даёт возможность сделать ряд оптимизаций, например, возведение в нулевую степень всегда возвращает единицу, а в первую степень – само выражение.

Для выражений определены основные математические функции из стандартной библиотеки языка, а именно: sin, cos, tan, exp, log, sqrt, sign, abs. При необходимости другие функции можно реализовать по образу и подобию этих функций. Покажем реализацию функций sin и log:

```
template<class E>
struct sin_expression : expression<sin_expression<E> >{
	typedef typename mul_expression_type<
		cos_expression<E>, typename E::diff_type
	>::type diff_type;

	sin_expression(const expression<E> &e) : e(e.self()){}

	diff_type diff() const {
		return cos(e) * e.diff();
	}
	template<typename T>
	T operator()(const T &x) const {
		return std::sin(e(x));
	}
	const E e;
};

template<class E>
sin_expression<E> sin(const expression<E>& e){
```

```
	return sin_expression<E>(e);
}

template<class E>
struct log_expression : expression<log_expression<E> >{
	typedef typename div_expression_type<
		typename E::diff_type, E
	>::type diff_type;

	log_expression(const expression<E> &e) : e(e.self()){}

	diff_type diff() const {
		return e.diff() / e;
	}
	template<typename T>
	T operator()(const T &x) const {
		return std::log(e(x));
	}
	const E e;
};

template<class E>
log_expression<E> log(const expression<E>& e){
	return log_expression<E>(e);
}
```

В случае выражений от нескольких переменных принципиально ничего не меняется. Основные изменения касаются определения переменных и дифференцирования, теперь производная становится частной и требуется указать, по какой переменной делается дифференцирование. В качестве идентификатора переменной используется целочисленная константа типа unsigned, порядковый номер переменной (считая от нуля). Это число передаётся как параметр шаблона в определение переменной и в функцию дифференцирования. Покажем всё это на примере определения переменной:

```
template<unsigned N>
struct variable : expression<variable<N> >{
	template<unsigned M>
	struct diff_type {
		typedef int_constant<M == N> type;
	};
	template<unsigned M>
	typename diff_type<M>::type diff() const {
		return typename diff_type<M>::type();
	}
	template<typename T, size_t _Size>
	T operator()(const std::array<T, _Size> &vars) const {
		return vars[N];
	}
```

```
};
```

Теперь производная от переменной равна единице или нулю в зависимости от того, идёт ли дифференцирование по этой переменной или по другой. diff_type, который в случае одной переменной был типом, становится метафункцией, принимающей в качестве параметра шаблона номер переменной, по которой ведётся дифференцирование, и возвращающей тип выражения для производной в типе type. Функция diff также становится шаблонной и принимает в качестве параметра шаблона номер переменной, по которой ведётся дифференцирование.

Ещё одно важное изменение касается функционального оператора (), вычисляющего значение выражения, который теперь принимает в качестве параметра не одиночную переменную, а массив переменных, реализованный в классе array из стандартной библиотеки языка C++.

4. Метод Ньютона

Покажем применение техники символьного дифференцирования на примере метода Ньютона решения нелинейного уравнения. Более точно, рассмотрим три разных задачи на этот метод. Первая – решение нелинейного уравнения с одной неизвестной, вторая – решение системы из нескольких нелинейных уравнений с несколькими переменными (полагаем, что число уравнений равно числу неизвестных), и третья – решение задачи на нахождение экстремальной точки (минимума или максимума) одного уравнения с несколькими неизвестными. Первая задача использует библиотеку символьного дифференцирования с одной переменной (symdiff1), а остальные две – библиотеку с несколькими переменными (symdiff).

Решение уравнения с одной переменной

Пусть имеется нелинейное уравнение $f(x)=0$. Разложим функцию f в ряд Тейлора до первого члена в некоторой точке x_0 (начальном приближении решения уравнения):

$$f(x)=f(x_0) + f'(x_0)(x-x_0) + O((x-x_0)^2).$$

Отбросим слагаемые старше первой степени и будем решать наше уравнение $f(x)=0$. Получим приближённое решение в виде:

$$x=x_0-f(x_0)/f'(x_0).$$

Теперь положим найденное приближённое решение в качестве нового начального приближения решения и так далее до тех пор, пока не получим требуемую точность решения (пока f(x) не станет достаточно малым) или пока число итераций не превысит указанного максимального значения.

Положим, что f(x) задано в виде некоторого выражения так, как это описано в предыдущем параграфе. Тогда можно написать следующую достаточно простую функцию:

```
template<class E>
double newton1(const expression<E> &expr, double x, unsigned maxiter){
      const E &f = expr.self();
      const typename E::diff_type fd = f.diff();
      const double eps = 1e-12;
      unsigned niter = 0;
      double fx;
      while (std::abs(fx = f(x)) > eps) {
            if (++niter > maxiter)
                  throw std::runtime_error("Too many iterations");
            x -= fx / fd(x);
      }
      return x;
}
```

Данная функция принимает три параметра: выражение для решаемого уравнения, начальное приближение и максимальное количество итераций (защита от зацикливания). Функция возвращает приближённое решение уравнения. Обращение к этой функции может выглядеть, например, так:

```
variable x;
double result = newton(exp(x) - 5, 0.1, 100);
```

Решение системы нелинейных уравнений

Пусть есть n уравнений $f_1 \ldots f_n$ от n неизвестных $x_1 \ldots x_n$. Такую систему можно записать как одно векторное уравнение (векторные величины будем отмечать полужирным шрифтом):

$$\mathbf{f}(\mathbf{x})=\mathbf{0}.$$

Разложение такого векторного уравнения в ряд Тейлора до первого члена имеет вид:

$$\mathbf{f}(\mathbf{x_0})+J(\mathbf{x_0})\cdot(\mathbf{x}-\mathbf{x_0})=\mathbf{0}.$$

В этом уравнении J – это якобиан системы уравнений (матрица частных производных), $J_{ik}=\partial f_i/\partial x_k$. Введём новую переменную $\mathbf{dx}=\mathbf{x}-\mathbf{x_0}$, тогда уравнение примет вид:

$$J(\mathbf{x_0})\cdot\mathbf{dx} = -\mathbf{f}(\mathbf{x_0}).$$

Решим это уравнение относительно переменных **dx** (например, методом Гаусса). После этого следующее приближение решения может быть найдено по формуле

$x=x_0+dx$.

Вот текст соответствующей функции:

```
template<typename T, typename... E>
val_vector<T, sizeof...(E)> newton(const std::tuple<E...> &tp,
      val_vector<T, sizeof...(E)> x, unsigned maxiter)
{
      const std::size_t N = sizeof...(E);
      square_matrix<boost::any, N> Jac = jacobian(tp);
      const double eps = 1e-12;
      unsigned niter = 0;
      val_vector<T, N> fx;
      while(max_abs(fx = eval_tuple(tp, x)) > eps){
            if (++niter > maxiter)
                  throw std::runtime_error("Too many iterations");
            x -= solve(eval_jacobian<T, E...>(Jac, x), fx);
      }
      return x;
}
```

Обращение к этой функции может выглядеть, например, так:

```
variable<0> x;
variable<1> y;
variable<2> z;
val_vector<double, 3> xv;
val_vector<double, 3> result = newton(std::make_tuple(
      x + y * y + z * z - 14,
      2 * x - y,
      x + y - z),
      xv, 100);
```

Эта функция хотя и похожа на аналогичную функцию для одного уравнения с одной неизвестной, но имеет и весьма существенные отличия. В первую очередь, эта функция принимает в качестве параметра не одно выражение, а целый кортеж (tuple) выражений. В качестве начального приближения она принимает не значение одной переменной, а вектор таких значений и возвращает в качестве решения также вектор значений. Вначале функция вычисляет якобиан (матрицу частных производных, квадратная матрица реализована во вспомогательном классе square_matrix). Каждый элемент этой матрицы – это выражение некоторого типа. Таким образом, эта матрица должна хранить элементы разных типов. Для того чтобы это было возможно, используется класс boost::any из библиотеки boost (см. [2]), который может хранить значение произвольного типа. Чтобы из объекта класса boost::any можно было извлечь хранящееся в нём значение, нужно знать тип этого значения. Мы можем это узнать следующим образом. Тип выражения для I-й функции определяется так:

```
typedef typename std::tuple_element<I, std::tuple<E...> >::type exp_type;
```

Тип выражения для частной производной I-й функции по J-й переменной определяется так:

```
typedef typename exp_type::template diff_type<J>::type diff_type;
```

Таким образом, для того чтобы узнать тип производной I-й функции по J-й переменной, эти два целых числа I и J должны быть известны на стадии компиляции. А это, в свою очередь, означает, что, например, для вычисления якобиана мы не можем использовать цикл времени исполнения (обычный цикл), а вместо этого должны прокрутить два вложенных цикла (по функциям и для каждой функции по переменным) во время компиляции. Так как язык метапрограммирования (подъязык языка C++ для работы с шаблонами) является чисто функциональным языком, то единственный способ работы с циклами в нём – это рекурсия. Покажем, как это можно сделать. Для этого используется вспомогательный шаблон класса meta_loop:

```
template<unsigned I, unsigned N>
struct meta_loop {
      template<class Closure>
      void operator()(Closure &closure)
      {
            closure.template apply<I>();
            meta_loop<I + 1, N>()(closure);
      }
};
template<unsigned N>
struct meta_loop<N, N> {
      template<class Closure>
      void operator()(Closure &closure){}
};
```

В этом шаблонном классе два параметра шаблона: текущая итерация цикла и общее число итераций цикла. Объект этого класса в своём функциональном операторе () принимает ссылку на замыкание (closure) произвольного класса Closure (передаётся как параметр шаблона этого оператора), вызывает в нём шаблонный метод без параметров apply, передавая ему в качестве параметра шаблона номер текущей итерации, а затем делает рекурсивный вызов в классе meta_loop с первым параметром (текущий номер итерации) на единицу больше. Для завершения рекурсии служит специализация класса meta_loop, в которой номер итерации совпадает с общим числом итераций. В этом специализированном классе соответствующий функциональный оператор ничего не делает. Объект «замыкание» назван так потому, что он должен в себе содержать все необходимые данные для выполнения метода apply (кроме номера текущей итерации, который передаётся как параметр шаблона этого метода), т.е. быть «замкнут» относительно этих данных.

Покажем теперь, как, с использованием класса meta_loop, реализована функция jacobian:

```
template<class E, std::size_t I, std::size_t N>
struct jacobian1_closure {
      jacobian1_closure(const expression<E> &e,
      square_matrix<boost::any, N> &result)
            : e(e.self()), result(result){}

      template<unsigned J>
      void apply(){
            result(I, J) = e.template diff<J>();
      }
private:
      const E &e;
      square_matrix<boost::any, N> &result;
};
template<class... E>
struct jacobian0_closure {
      static const std::size_t N = sizeof...(E);

      jacobian0_closure(const std::tuple<E...> &tp,
            square_matrix<boost::any, N> &result)
            : tp(tp), result(result){}

      template<unsigned I>
      void apply()
      {
            typedef typename std::tuple_element<I, std::tuple<E...>
>::type expr_type;
            jacobian1_closure<expr_type, I, N> closure(std::get<I>(tp),
result);
            meta_loop<0, N>()(closure);
      }
private:
      const std::tuple<E...> &tp;
      square_matrix<boost::any, N> &result;
};

template<class... E>
square_matrix<boost::any, sizeof...(E)>
jacobian(const std::tuple<E...> &tp){
      const std::size_t N = sizeof...(E);
      square_matrix<boost::any, N> result;
      jacobian0_closure<E...> closure(tp, result);
      meta_loop<0, N>()(closure);
      return result;
}
```

Для реализации якобиана используются два класса-замыкания: jacobian0_closure для реализации цикла по функциям и jacobian1_closure для реализации цикла по переменным для одной функции. Метод apply класса jacobian0_closure для каждой (I-й) итерации цикла по функциям «достаёт» I-е выражение из кортежа и запускает для этого выражения вложенный цикл по переменным. Классу jacobian1_closure в качестве параметра шаблона передаются тип выражения, текущий номер функции и общее число функций (и переменных). Метод apply класса jacobian1_closure уже имеет всю необходимую информацию для вычисления выражения для частной производной I-й функции по J-й переменной и присваивает это выражение элементу матрицы.

Обратим внимание также на две функции – eval_tuple и eval_jacobian. Эти функции принимают, соответственно, кортеж выражений или матрицу выражений якобиана и вектор значений переменных и возвращают, соответственно, вектор значений этих выражений или матрицу значений якобиана.

Решение задачи на нахождение экстремальной точки

В данной задаче имеется одна функция от нескольких переменных (в виде выражения). Требуется найти набор значений этих переменных, при которых все частные производные данной функции по этим переменным равны нулю. Это может быть как точка минимума или максимума функции, так и седловая точка (по некоторым переменным минимум, а по другим – максимум) или даже точка перегиба по некоторым переменным, в которой первая отличная от нуля частная производная по переменной имеет нечётный номер.

Эта задача во многом похожа на предыдущую задачу, если положить, что вектор выражений – это градиент нашей функции. Мы ищем точку, в которой этот градиент равен нулевому вектору. В этом случае в качестве якобиана выступает гессиан (матрица вторых частных производных). Единственное существенное отличие состоит в том, что эти выражения не произвольные, а связаны с исходной функцией. Приведём исходный текст соответствующей функции:

```
template<typename E, typename T, std::size_t N>
val_vector<T, N> findmin(const expression<E> &expr,
      val_vector<T, N> x, unsigned maxiter)
{
      std::array<boost::any, N> gradx = grad<N>(expr);
      square_matrix<boost::any, N> hessianx = hessian<N>(expr);
      const double eps = 1e-12;
      unsigned niter = 0;
      val_vector<double, N> fx;
      while (max_abs(fx = eval_grad<E>(gradx, x)) > eps){
            if (++niter > maxiter)
                  throw std::runtime_error("Too many iterations");
            x -= solve(eval_hessian<E>(hessianx, x), fx);
```

```
	}
	return x;
}
```

Обращение к этой функции может выглядеть, например, так:

```
variable<0> x1;
variable<1> x2;
val_vector<double, 2> xv;
xv[0] = -2;
xv[1] = 5;
val_vector<double, 2> result = findmin(
  100 * sd::pow<2>(x2 - sd::pow<2>(x1)) + sd::pow<2>(1 - x1), xv, 100);
```

Эта функция очень похожа на предыдущую, если заменить кортеж исходных выражений на массив выражений, вычисленных как градиент исходного выражения, а якобиан – на его гессиан. Приведём исходный текст функции, возвращающей гессиан исходного выражения:

```
template<class E, std::size_t I, std::size_t N>
struct hessian1_closure {
	hessian1_closure(const expression<E> &expr,
		square_matrix<boost::any, N> &result)
		: expr(expr.self()), result(result){}

	template<unsigned J>
	void apply(){
		result(I, J) = expr.template diff<J>();
	}
private:
	const E &expr;
	square_matrix<boost::any, N> &result;
};

template<class E, std::size_t N>
struct hessian0_closure {
	hessian0_closure(const expression<E> &expr,
		square_matrix<boost::any, N> &result)
		: expr(expr.self()), result(result){}

	template<unsigned I>
	void apply()
	{
		typedef typename E::template diff_type<I>::type diff_type;
		hessian1_closure<diff_type, I, N>
			closure(expr.template diff<I>(), result);
		meta_loop<0, N>()(closure);
	}
private:
```

```
      const E &expr;
      square_matrix<boost::any, N> &result;
};
template<std::size_t N, class E>
square_matrix<boost::any, N> hessian(const expression<E> &expr){
      square_matrix<boost::any, N> result;
      hessian0_closure<E, N> closure(expr, result);
      meta_loop<0, N>()(closure);
      return result;
}
```

Вычисление гессиана от выражения похоже на вычисление якобиана от кортежа выражений. Здесь также есть два замыкания: hessian0_closure и hessian1_closure. Первое в своём методе apply вычисляет частную производную от исходного выражения по I-й переменной, а второе – частную производную от этой производной по J-й переменной.

5. Численное моделирование

Покажем на примере, как можно эффективно использовать символьное дифференцирования при решении задач численного моделирования на примере задачи численного моделирования двумерных течений умеренно-разреженного газа. Рассмотрим двухкомпонентную систему. Свободная энергия Гельмгольца в виде $\Psi = \Psi(\rho, T, C)$, которая является термодинамическим потенциалом, для случая двух компонентов может задаваться уравнением:

$$\Psi=C\Psi_1(\rho)+(1-C)\Psi_2(\rho)+A_\psi C^2(1-C)^2,$$

где A_ψ – постоянная, $\Psi_j=c_{sj}^2\ln(\rho/\rho_j)$ – собственные свободные энергии компонентов. Однако возможен и более сложный вариант задания свободной энергии:

$$\Psi=C\Psi_1(\rho)+(1-C)\Psi_2(\rho)+RT[C\ln C+(1-C)\ln(1-C)]+2RT_{cr}C(1-C).$$

Оба выражения получаются на основе приближения регулярных растворов (см. [6]). При этом первое является, в известном смысле, аппроксимацией второго. Однако возможны и другие подходы (см. [7]).

Как бы ни было задано выражение для $\Psi(\rho, T, C)$, необходимо уметь считать на её основе различные термодинамические характеристики.

Например, для термодинамического давления имеем:

$$p = \rho^2 \frac{\partial \Psi}{\partial \rho},$$

а для «локальной» части обобщённого химического потенциала (см. [6])

$$\mu_{loc} = \frac{\partial \Psi}{\partial C}.$$

Таким образом, нужно, зная выражение для свободной энергии, уметь вычислять её частные производные по ρ и C. С использованием библиотеки

символьного дифференцирования для нескольких переменных (как для решения системы нелинейных уравнений методом Ньютона) задача может быть решена достаточно просто. Покажем это на примере:

```
enum variables { _rho, _T, _C };
const variable<_rho> rho;
const variable<_T> T;
const variable<_C> C;

const double A = 2, rho1 = 1, rho2 = 1, cs1 = 1000, cs2 = 1000;
const auto
     Psi1 = (cs1 * cs1) * log(rho / rho1),
     Psi2 = (cs2 * cs2) * log(rho / rho2);
// Выражение для Psi
const auto Psi = C * Psi1 + (1 - C) * Psi2 + A * sd::pow<2>(C * (1 - C));
// Выражение для p
const auto p = sd::pow<2>(rho) * Psi.diff<_rho>();
// Выражение для mu
const auto muloc = Psi.diff<_C>();
// Эти строки можно исполнять в цикле
val_vector<double, 3> vars;
vars[_rho] = 1;
vars[_T] = 273;
vars[_C] = 2;
const double
     v_p = p(vars),
     v_muloc = muloc(vars);
```

6. Заключение

Применение символьного дифференцирования при решении вычислительных задач помогает автоматизировать ряд действий, реализация которых «вручную» может быть, во-первых, достаточно сложной, а во-вторых, быть подвержена разного рода ошибкам (опечаткам). Например, может быть известно выражение, задающее некую величину как функцию от нескольких переменных, и требуется найти значение частной производной от этой величины по одной или нескольким переменным. Традиционный способ решения таких задач – выписать формулу для производной «на бумаге» и затем запрограммировать её в виде отдельной функции. При изменении вида выражения всю работу требуется повторить «вручную» заново. Помимо того, что это занимает время на рутинную нетворческую работу, никто не может гарантировать, что не будет допущена трудно отлавливаемая ошибка из-за досадной опечатки. Применение символьного дифференцирования избавляет прикладного программиста от обеих проблем. Рутинную работу, как и положено, будет выполнять компьютер, который не бывает невнимательным и не совершает досадных опечаток. Ошибки в самом символьном дифференцировании, конечно же, могут быть, но требуют однократной отладки.

Что касается скорости работы, то, как уже говорилось выше, библиотека символьного дифференцирования делает множество оптимизаций (таких, например, как умножение нуля на любое выражение всегда даёт нуль) и её применение не должно сильно замедлять выполнение программы по сравнению с функцией, написанной «вручную», а может даже и ускорить выполнение за счёт инлайнового исполнения шаблонных методов.

7. Библиографический список

1. David Abrahams, Aleksey Gurtovoy. C++ Template Metaprogramming. Addison-Wesley. — 2004.
2. Boost C++ Libraries. URL: http://www.boost.org/
3. T. Veldhuizen, Expression Templates, C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31.
4. Curiously recurring template pattern (CRTP). URL: http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
5. Substitution failure is not an error (SFINAE). URL: https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error
6. Liu J, Thermodynamically Consistent Modeling and Simulation of Multiphase Flows, DISSERTATION, THE UNIVERSITY OF TEXAS AT AUSTIN, December 2014.
7. K. S. Glavatskiy, D. Bedeaux, Nonequilibrium properties of a two-dimensionally isotropic interface in a two-phase mixture as described by the square gradient model, PHYSICAL REVIEW E 77, 061101, 2008.