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

Препринты ИПМ • Препринт № 24 за 2016 г.

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

Краснов М.М., Ладонкина М.Е.

Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++

Рекомендуемая форма библиографической ссылки: Краснов М.М., Ладонкина М.Е. Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++ // Препринты ИПМ им. М.В.Келдыша. 2016. № 24. 23 с. doi:10.20948/prepr-2016-24
URL: http://library.keldysh.ru/preprint.asp?id=2016-24

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Российской академии наук

М.М. Краснов, М.Е. Ладонкина

Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++

Москва — 2016

М.М. Краснов, М.Е. Ладонкина

Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++

Многие задачи математической физики обладают существенной вычислительной сложностью, особенно при решении задач на трёхмерных сетках, которые бывают очень большими. Разрывный метод Галёркина – как раз пример такой задачи. Поэтому уменьшение объёма вычислений – весьма актуальная задача. Один из возможных способов уменьшения объёма вычислений состоит в переносе части вычислений на стадию компиляции. Язык C++ с появлением в нём шаблонов (которых в первоначальной версии языка не было) предоставляет такую возможность. В данной работе иллюстрируется применение шаблонного метапрограммирования для ускорения вычислений в разрывном методе Галёркина. Кроме того, шаблонное метапрограммирование иногда позволяет упростить алгоритм за счёт его обобщения.

Ключевые слова: шаблонное метапрограммирование, трёхмерные тетраэдральные сетки, разрывный метод Галёркина

Mikhail Mikhailovich Krasnov, Marina Evgenievna Ladonkina

Discontinuous Galerkin method on three-dimensional tetrahedral meshes. The usage of C++ template metaprogramming

Many mathematical physics problems possess considerable computational complexity, especially when solving problems on three-dimensional meshes, which sometimes may be very large. Discontinuous Galerkin method is an example of such a problem. That is why the decreasing of the computing amount is an actual task. One of the possible methods of decreasing of the computing amount is to move some part of computations to the compile stage. C++ language with the appearance of templates (in the initial version of the language they were absent) gives such a possibility. This work illustrates the usage of template metaprogramming to speed-up the discontinuous Galerkin method. Besides, template metaprogramming sometimes allows to simplify an algorithm by its generalizing.

Key words: template metaprogramming, three-dimensional tetrahedral meshes, discontinuous Galerkin method

Работа выполнена при поддержке грантов РФФИ №14-01-00145, № 16-01-00333

Оглавление

1. Введение

Многие языки программирования позволяют производить часть вычислений на стадии компиляции. Никого не удивит, что если в программе написать такой оператор:

```
int i = 5 + 6;
```

то компилятор выражение 5 + 6 посчитает на стадии компиляции и переменной i сразу присвоит значение 11. Мы вправе ожидать, что так же себя будут вести и компиляторы с других языков программирования.

С появлением в языке C++ шаблонов возможности для вычислений на стадии компиляции многократно выросли. В языке появился новый термин – шаблонное метапрограммирование [1], под которым подразумеваются вычисления на стадии компиляции. На этом этапе могут быть вычислены типы и целочисленные константы. Теоретически часть вычислений (целочисленных констант) может быть перенесена со стадии исполнения на стадию компиляции, и, таким образом, работу программы можно ускорить. Важным условием того, что это может быть сделано, является то, что данные для вычислений должны быть доступны уже на стадии компиляции. Для разрывного метода Галёркина [2,3] оказывается, что программу можно организовать так, что это условие оказывается выполненным. Основным механизмом для метавычислений являются метафункции – специальным образом написанные шаблонные

классы. Отличием метафункций от обычных функций является то, что вычисленные результаты (один или несколько) хранятся внутри класса в именованных переменных. Целочисленные результаты хранятся в static const переменных одного из целочисленных типов, а типы – в typedef-ах.

Когда мы говорим об ускорении работы программы с помощью метапрограммирования, мы имеем в виду вынос вычислений части целочисленных значений на стадию компиляции. Но у метапрограммирования есть и другая не менее важная сторона – вычисление типов, не имеющее, на первый взгляд, прямого отношения к ускорению вычислений. Одно из интереснейших применений шаблонного метапрограммирования – шаблоны выражений (expression templates). Примеров применения шаблонов выражений можно привести множество. Например, в операторном методе программирования [4] шаблоны выражений используются для запоминания цепочки вычислений и применения этой цепочки ко всем ячейкам плотной сеточной функции. Другие примеры – это матричная арифметика и арифметика чисел с высокой точностью, символьное дифференцирование. С помощью шаблонного метапрограммирования можно реализовать язык с простой грамматикой (некоторые элементарные объекты какой-либо природы, например, матрицы, сеточные функции и арифметика с ними). Именно такой простой язык был реализован в методе Галёркина в операторе объёмного интегрирования.

Шаблоны выражений позволяют, во-первых, сильно сократить написание сложных формул: их вид в программах становится похожим на вид формул в математических работах. Во-вторых, применение шаблонов вычислений позволяет избавиться от переменных для хранения промежуточных значений (сэкономить память) и часто существенно ускорить вычисления.

2. Применение шаблонного метапрограммирования

Приведём несколько примеров использования метапрограммирования. Начнём с простых примеров, затем приведём примеры использования метапрограммирования на стыке компиляции и исполнения и в конце главы покажем более сложные примеры.

2.1. Простые примеры

Факториал. Пусть число N известно на этапе компиляции. Тогда N! можно вычислить на этапе компиляции с помощью следующей метафункции:

```
template<unsigned N> struct factorial {
  static const unsigned value = N * factorial<N - 1>::value;
};
template<> struct factorial<0> {
  static const unsigned value = 1;
};
```

Первый класс является общим случаем и реализует рекурсивный вызов по известному правилу. Второй класс – это т.н. «специализация» класса, и он служит для завершения рекурсии. Пример обращения к этой метафункции:

```
int i = factorial<5>::value;     // 120
```

Биномиальные коэффициенты. Пусть числа N и K известны во время компиляции. Тогда биномиальные коэффициенты могут быть вычислены с помощью следующей метафункции:

```
template<unsigned N, unsigned K> struct C {
  static const unsigned value =
    factorial<N>::value / factorial<K>::value / factorial<N-K>::value;
};
// Пример обращения:
int i = C<5, 2>::value; // 10
```

Ещё один полезный простой пример: $(-1)^N$.

```
template<unsigned N>
struct _1_pow {
     static const int value = N % 2 == 0 ? 1 : -1;
};
// Пример обращения:
int i = _1_pow<5>::value;  // -1
```

2.2. На стыке компиляции и исполнения

Пусть нужно вычислить степенную функцию x^N. Здесь x – вещественное число, а N – целое (положительное, ноль или отрицательное). Обычная реализация такой функции могла бы выглядеть примерно так:

```
double pow(double x, int N){
     if(N < 0) return 1 / pow(x, -N);
     else if(N == 0) return 1;
     else if(N % 2 == 0){
          double p = pow(x, N / 2);
          return p * p;
     } else return pow(x, N - 1) * x;
}
```

Пусть теперь N известно во время компиляции. Т.к. x – вещественное число, то и результат – тоже вещественное число и на стадии компиляции вычислен быть не может. Кроме того, x становится известным только во время исполнения. Значит, написать обычную метафункцию (как для факториала) нельзя. Но оказывается, что на стадию компиляции можно вынести все условные операторы (а они зависят только от N, которое во время компиляции известно). На стадии исполнения останутся только умножения. Кроме того, рекурсивные вызовы будут подставлены (inline), и вычисление степенной функции будет очень быстрым. Покажем, как это можно сделать. Вначале пишется метафункция enable_if:

```
template<bool B, typename T>
struct enable_if { typedef T type;};
template<typename T>
struct enable_if<false, T>{};
```

Если первый параметр B равен true, то эта метафункция в переменной type возвращает свой второй параметр (тип), иначе она ничего не возвращает.

Теперь сама функция pow:

```
template<int N, typename T>
typename enable_if<(N<0), T>::type
pow(T x){ return 1 / pow<-N>(x); }

template<int N, typename T>
typename enable_if<(N==0), T>::type
pow(T x){ return 1; }

template<int N, typename T>
typename enable_if<(N>0)&&(N%2==0), T>::type
pow(T x){ T p = pow<N/2>(x); return p*p; }

template<int N, typename T>
typename enable_if<(N>0)&&(N%2==1), T>::type
pow(T x){ return pow<N-1>(x)*x; }

y=pow<3>(x); // Пример вызова
```

Особенностью языка C++ является то, что если enable_if не содержит переменную с указанным именем (в данном случае type), то компилятор не выдаёт сообщения об ошибке, а просто «не видит» функцию. Таким образом, при любом значении N видимой будет ровно одна реализация функции pow.

2.3. Более сложные примеры

Интегрирование одночленов. Рассмотрим интеграл от одночлена вида $x^\alpha y^\beta z^\gamma$ по стандартному трёхмерному симплексу. Стандартный трёхмерный симплекс – это тетраэдр с координатами вершин в точках (0, 0, 0), (1, 0, 0), (0, 1, 0) и (0, 0, 1). Такой интеграл вычисляется аналитически и задаётся формулой:

$$\int_0^1 x^\alpha dx \int_0^{1-x} y^\beta dy \int_0^{1-x-y} z^\gamma dz = \frac{\alpha!\beta!\gamma!}{(\alpha+\beta+\gamma+3)!}.$$

Пусть α, β и γ – константы, известные во время компиляции. Значение интеграла – вещественное число, меньшее единицы (знаменатель дроби больше числителя), и не может быть вычислено во время компиляции, но, как легко заметить (из комбинаторных соображений), обратное число целое (знаменатель дроби делится на числитель нацело) и, соответственно, во время компиляции вычислено быть может. Напишем метафункцию, вычисляющую число, обратное интегралу от одночлена:

```
template<unsigned alpha, unsigned beta, unsigned gamma>
```

```
struct monomial3d {
  static const unsigned value =
    factorial<alpha+beta+gamma+3>::value / factorial<alpha>::value
    / factorial<beta>::value / factorial<gamma>::value;
};
```

Аналогичный интеграл от одночлена по стандартному двумерному симплексу задаётся формулой:

$$\int_0^1 x^\alpha dx \int_0^{1-x} y^\beta dy = \frac{\alpha!\beta!}{(\alpha+\beta+2)!}.$$

Соответствующая метафункция выгладит так:

```
template<unsigned alpha, unsigned beta>
struct monomial2d {
  static const unsigned value =
    factorial<alpha+beta+2>::value
      / factorial<alpha>::value / factorial<beta>::value;
};
```

Вычисление интеграла от бинома. Вычислим интеграл по стандартному двумерному симплексу от бинома – выражения вида $(ax+by)^N$, где N – константа, известная во время исполнения. Разложив бином по известной формуле, получим:

$$\iint (ax+by)^N dS = \sum_{K=0}^{N} C_N^K a^K b^{N-K} \iint x^K y^{N-K} dS.$$

Так как a и b – это вещественные числа, известные только во время исполнения, то этот интеграл не может быть вычислен во время компиляции. Но интегралы от одночленов по стандартному двумерному симплексу мы уже умеем считать во время компиляции. Так что полученный код опять (как и для степенной функции) будет на стыке компиляции и исполнения. Вот что в итоге получается:

```
template<unsigned N, unsigned K, typename T>
typename enable_if<(K>0), T>::type
binom(T a, T b){
  return binom<N, K - 1>::value(a, b) +
    C<N, K>::value * pow<K>(a) * pow<N-K>(b) /
    monomial2d<K, N-K>::value;
}
template<unsigned N, unsigned K, typename T>
typename enable_if<(K==0), T>::type
binom(T a, T b){ return pow<N>(b) / monomial2d<0,N>::value; }
// Пример использования
double s = binom<5, 5>(a, b);
```

В данном примере на стадии компиляции вычисляются биномиальные коэффициенты и величины, обратные интегралам от одночленов по стандартному двумерному симплексу. Шаблонная функция времени исполнения binom вычисляет частичную сумму от 0 до K, вызывая рекурсивно

саму себя, причём для K=0 вызывается вторая версия функции, завершающая рекурсию. Чтобы вычислить полную сумму, нужно передать значение K=N. Параметр шаблона T (тип параметров и возвращаемого значения) можно не указывать, он вычисляется компилятором автоматически по типам входных параметров a и b. Пример с биномом демонстрирует т.н. «автоматическое» раскрытие скобок.

Обобщение суммирования. При интегрировании бинома был продемонстрирован механизм вычисления сумм, когда основная версия функции вычисляет частичную сумму, вызывая рекурсивно саму себя, и имеется версия функции, завершающая рекурсию. Такой способ вычислений используется не только при вычислении интеграла от бинома, но и в множестве других случаев, поэтому имеет смысл обобщить суммирование, написав шаблонную функцию, вычисляющую аналогичным способом произвольную сумму. Разные суммы отличаются друг от друга только формулой для вычисления очередного члена ряда. Будем передавать класс, вычисляющий этот очередной член, как параметр шаблона нашей функции обобщённого суммирования. При вычислении очередного члена ряда могут потребоваться дополнительные параметры времени исполнения. Будем их передавать как параметр функции суммирования. Тип этого параметра может быть любым и передаётся как параметр шаблона функции суммирования. Вот текст обобщённой функции суммирования:

```
template<unsigned N, unsigned K, typename T,
  class action_class, typename A>
typename enable_if<(K>0), T>::type
abstract_sum(const A &args){
  return abstract_sum<N, K - 1, T, action_class, A>(args)
    + action_class::template value<N, K>(args);
}
template<unsigned N, unsigned K, typename T,
  class action_class, typename A>
typename enable_if<(K==0), T>::type
abstract_sum(const A &args){
  return action_class::template value<N, K>(args);
}
```

Видно, что эта функция обобщённого суммирования похожа на функцию binom, но очередной член ряда вычисляется не явно, а для этого вызывается шаблонная функция класса action_class:

```
action_class::template value<N, K>(args);
```

С использованием этой обобщенной функции суммирования интеграл от бинома можно вычислить так:

```
struct binom_action {
  template<unsigned N, unsigned K, typename T>
  static T value(const std::pair<T, T> &args){
```

```
    return C<N, K>::value *
      pow<K>(args.first) * pow<N-K>(args.second) /
      monomial2d<K, N - K>::value;
  }
};
template<unsigned N, typename T>
T binom(T a, T b){
  return abstract_sum<N, N, T, binom_action>
    (std::pair<T, T>(a, b));
}
```

Шаблонной функции value класса binom_action для вычисления требуются два числа (a и b), которые нужно передать одним параметром (согласно интерфейсу функции abstract_sum), поэтому эти два числа объединяются в пару и эта пара чисел передаётся как один параметр. Функция value сама извлекает числа из переданной пары (args.first и args.second).

Текст функции binom достаточно простой, и поэтому при переходе на обобщённое суммирование он короче не стал, однако в более сложных случаях (например, при интегрировании по трёхмерному симплексу) функция обобщённого суммирования будет очень полезной.

3. Метапрограммирование в методе Галёркина

3.1. Базисные функции

В методе Галёркина искомая функция в каждой ячейке разлагается на базисные функции, имеющие вид:

$$\varphi = \left(\frac{x - x_c}{\Delta x}\right)^{\alpha} \cdot \left(\frac{y - y_c}{\Delta y}\right)^{\beta} \cdot \left(\frac{z - z_c}{\Delta z}\right)^{\gamma}.$$

Здесь x_c, y_c, z_c – координаты центра ячейки,
Δx, Δy, Δz – характерные размеры ячейки.

Будем считать, что целые неотрицательные числа α, β, γ для каждой базисной функции известны во время компиляции. Базисные функции можно определить с помощью следующего шаблона:

```
template<unsigned ALPHA, unsigned BETA, unsigned GAMMA>
struct base_function
{
     static const unsigned
          alpha = ALPHA,
          beta = BETA,
          gamma = GAMMA;
};
```

Базисных функций может быть или 4 (линейное разложение, сумма α+β+γ не превышает 1), или 10 (квадратичное разложение, сумма α+β+γ не превышает

2). В дальнейшем будет рассматривать квадратичное разложение. Сами базисные функции теперь можно определить следующим образом:

```
struct bf_1 : base_function<0, 0, 0>{};
struct bf_x : base_function<1, 0, 0>{};
struct bf_y : base_function<0, 1, 0>{};
struct bf_z : base_function<0, 0, 1>{};
struct bf_x2 : base_function<2, 0, 0>{};
struct bf_y2 : base_function<0, 2, 0>{};
struct bf_z2 : base_function<0, 0, 2>{};
struct bf_xy : base_function<1, 1, 0>{};
struct bf_xz : base_function<1, 0, 1>{};
struct bf_yz : base_function<0, 1, 1>{};
```

Объединим все базисные функции в одну структуру:

```
template<class BF0, class BF1, class BF2, class BF3, class BF4,
  class BF5, class BF6, class BF7, class BF8, class BF9>
struct test_functions {
     typedef BF0 bf_type0;
     typedef BF1 bf_type1;
     typedef BF2 bf_type2;
     typedef BF3 bf_type3;
     typedef BF4 bf_type4;
     typedef BF5 bf_type5;
     typedef BF6 bf_type6;
     typedef BF7 bf_type7;
     typedef BF8 bf_type8;
     typedef BF9 bf_type9;
};
```

С помощью библиотеки boost (см. [4]) эту структуру можно описать более компактно следующим образом:

```
#define BASE_FUNCTION_COUNT 10
template<BOOST_PP_ENUM_PARAMS(BASE_FUNCTION_COUNT, class BF)>
struct base_functions {
#define BASE_FUNCION_TYPES(z, n, unused) \
     typedef BF##n bf_type##n;
BOOST_PP_REPEAT(BASE_FUNCTION_COUNT, BASE_FUNCION_TYPES, ~)
#undef BASE_FUNCION_TYPES
};
```

Макрос BOOST_PP_REPEAT имеет три параметра:

- число повторов;
- пользовательский макрос, который нужно повторить указанное число раз;
- дополнительный параметр, передаваемый в пользовательский макрос.

Пользовательский макрос должен принимать три параметра. Первый параметр нам пока не существенен. Во втором параметре передаётся текущее значение индекса цикла, в третьем – дополнительный параметр.

Дополнительный параметр в данном случае не используется, поэтому в качестве его значения можно передать что угодно (например, тильду).

Важным преимуществом такого описания является то, что переход между квадратичным и линейным разложением становится тривиальным. Достаточно переопределить макропеременную `BASE_FUNCTION_COUNT`.

Теперь мы можем объединить все базисные функции в одну структуру:

```
typedef base_functions<
     bf_1, bf_x, bf_y, bf_z,    // Линейные члены
     bf_x2, bf_y2, bf_z2, bf_xy, bf_xz, bf_yz // Квадратичные члены
> bf_t;
```

Теперь, например, значение параметра α для базисной функции номер 3 можно узнать так:

```
const unsigned alpha = bf_t::bf_type3::alpha;
```

3.2. Матрица масс

В разрывном методе Галёркина для каждого тетраэдра строятся т.н. матрица масс – интегралы по тетраэдру от произведения двух базисных функций, т.е. интегралы вида: $a_{ij} = \iiint \varphi_i \varphi_j d\,\Omega$, где интегрирование ведётся по тетраэдру.

Подынтегральное выражение имеет вид:

$$\varphi_{ij} = \left(\frac{x - x_c}{\Delta x}\right)^{\alpha_{ij}} \cdot \left(\frac{y - y_c}{\Delta y}\right)^{\beta_{ij}} \cdot \left(\frac{z - z_c}{\Delta z}\right)^{\gamma_{ij}},$$

где $\alpha_{ij}=\alpha_i+\alpha_j$, $\beta_{ij}=\beta_i+\beta_j$, $\gamma_{ij}=\gamma_i+\gamma_j$.

Пусть тетраэдр задан своими четырьмя вершинами $p_i=(x_i, y_i, z_i)$, i=1…4. Сделаем замену переменных и перейдём к интегрированию по стандартному трёхмерному симплексу:

$x=a_1\xi+a_2\eta+a_3\zeta+a_4$,
$y=b_1\xi+b_2\eta+b_3\zeta+b_4$,
$z=c_1\xi+c_2\eta+c_3\zeta+c_4$.

Здесь

$a_1=x_1-x_4$, $a_2=x_2-x_4$, $a_3=x_3-x_4$, $a_4=x_4$,
$b_1=y_1-y_4$, $b_2=y_2-y_4$, $b_3=y_3-y_4$, $b_4=y_4$,
$c_1=z_1-z_4$, $c_2=z_2-z_4$, $c_3=z_3-z_4$, $c_4=z_4$.

В новых координатах базисные функции будут иметь вид:

$$\varphi = ((a_1\xi+a_2\eta+a_3\zeta+a_4 - x_c)/\Delta x)^{\alpha} \cdot ((b_1\xi+b_2\eta+b_3\zeta+b_4 - y_c)/\Delta y)^{\beta} \cdot ((c_1\xi+c_2\eta+c_3\zeta+c_4 - z_c)/\Delta z)^{\gamma}$$

Таким образом, вычисление матрицы масс сводится к интегрированию по стандартному трёхмерному симплексу выражения вида:

$$(a_1 x + a_2 y + a_3 z + a_4)^{\alpha} \cdot (b_1 x + b_2 y + b_3 z + b_4)^{\beta} \cdot$$

$$(c_1x + c_2y + c_3z + c_4)^\gamma$$

В этом выражении мы, как обычно, полагаем, что α, β и γ – целочисленные неотрицательные константы, известные во время компиляции.

Наша цель – «автоматически» раскрыть все скобки, после чего интегрирование всего выражения сведётся к интегрированию одночленов по стандартному симплексу, что мы уже умеем. Для начала запишем тождество для раскрытия одной скобки:

$$(ax + by + cz + d)^N =$$
$$\sum_{i=0}^{N} C_i^N d^{N-i} \cdot \sum_{j=0}^{i} C_j^i (cz)^{i-j} \cdot \sum_{k=0}^{j} C_k^j (ax)^k (by)^{j-k}.$$

В этом тождестве C_K^N – биномиальные коэффициенты.

Таким образом, для того чтобы раскрыть скобку, нужно сосчитать три вложенных суммы. Будем раскрывать скобки, начиная с первой. После раскрытия первой скобки исходное выражение будет иметь вид:

$$\sum_i s_i x^{p_i} y^{q_i} z^{r_i} \cdot (b_1x + b_2y + b_3z + b_4)^\beta \cdot (c_1x + c_2y + c_3z + c_4)^\gamma.$$

Здесь p_i, q_i, r_i – целочисленные константы, известные во время компиляции, s_i – вещественные числа.

После раскрытия второй скобки исходное выражение будет иметь вид:

$$\sum_j t_j x^{u_j} y^{v_j} z^{w_j} \cdot (c_1x + c_2y + c_3z + c_4)^\gamma.$$

Здесь u_j, v_j, w_j – целочисленные константы, известные во время компиляции, t_j – вещественные числа.

Таким образом, полное раскрытие всех скобок состоит из трёх шагов: раскрываются первая, вторая и третья скобки. В начале покажем, как раскрывается одна скобка. Как уже говорилось выше, для этого надо сосчитать три вложенных суммы. Для вычисления одной суммы будем использовать описанную выше функцию abstract_sum. Для вычисления одного члена ряда суммы нам нужны наборы коэффициентов a_i, b_i, c_i. Их надо передать в функцию abstract_sum одним параметром. Будем для этого использовать следующий класс abc:

```
template<typename T>
struct abc {
  abc(const T a_[], const T b_[], const T c_[]) : a(a_), b(b_), c(c_){}

  template<unsigned step>
  typename enable_if<step == 1, const T*>::type
  get() const { return a; }

  template<unsigned step>
  typename enable_if<step == 2, const T*>::type
  get() const { return b; }

  template<unsigned step>
  typename enable_if<step == 3, const T*>::type
  get() const { return c; }
```

```
private:
  const T *a, *b, *c;
};
```

У этого класса есть функция-член get, которая по номеру шага возвращает соответствующий набор коэффициентов (a, b или c).

Теперь покажем набор из трёх функций, вычисляющих три вложенные суммы:

```
template<unsigned step, class next_step,
  unsigned PX, unsigned PY, unsigned PZ>
struct tetra_integrate_polinomial_sum3 {
  template<unsigned J, unsigned K, typename T>
  static T value(const abc<T> &args){
    return C<J, K>::value * pow<K>(args.template get<step>()[0]) *
      pow<J - K>(args.template get<step>()[1]) *
      next_step::template value<PX + K, PY + J - K, PZ>(args);
  }
};
template<unsigned step, class next_step,
  unsigned PX, unsigned PY, unsigned PZ>
struct tetra_integrate_polinomial_sum2 {
  template<unsigned I, unsigned J, typename T>
  static T value(const abc<T> &args){
    return C<I, J>::value * pow<I-J>(args.template get<step>()[2]) *
      abstract_sum<J, J, T, tetra_integrate_polinomial_sum3<
        step, next_step, PX, PY, PZ + I - J> >(args);
  }
};
template<unsigned step, class next_step,
  unsigned PX, unsigned PY, unsigned PZ>
struct tetra_integrate_polinomial_sum1 {
  template<unsigned N, unsigned I, typename T>
  static T value(const abc<T> &args){
    return C<N, I>::value * pow<N-I>(args.template get<step>()[3]) *
      abstract_sum<I, I, T, tetra_integrate_polinomial_sum2<
        step, next_step, PX, PY, PZ> >(args);
  }
};
```

Каждая из этих функций вычисляет частичную сумму ряда. Функция tetra_integrate_polinomial_sum1 для вычисления одного члена ряда использует функцию tetra_integrate_polinomial_sum2, которая, в свою очередь, использует функцию tetra_integrate_polinomial_sum3. Последняя для вычисления члена ряда использует внешний класс next_step.

Поясним смысл шаблонных параметров классов:

- step – шаг, порядковый номер раскрываемой скобки. Предназначен для извлечения набора коэффициентов из параметра args.

- next_step – внешний класс, используемый для вычисления одного члена ряда самой внутренней суммы. Для шага 1 это раскрытые второй скобки, для шага 2 – раскрытие третьей скобки, а для шага 3 – интегрирование одночлена.
- PX, PY, PZ – целочисленные константы, степени, которые нужно дополнительно прибавить к степеням переменных x, y и z при интегрировании одночленов. При раскрытии первой скобки они равны нулю.

Далее – функции, используемые в качестве параметра next_step для описанных выше функций:

```
struct tetra_integrate_monomial {
  template<unsigned PX, unsigned PY, unsigned PZ, typename T>
  static T value(const abc<T> &){
    return T(1) / monomial3d<PX, PY, PZ>::value;
  }
};
template<unsigned gamma>
struct tetra_integrate_polinomial_step_3 {
  template<unsigned PX, unsigned PY, unsigned PZ, typename T>
  static T value(const abc<T> &args){
    return abstract_sum<gamma, gamma, T, tetra_integrate_polinomial_sum1<
      3, tetra_integrate_monomial, PX, PY, PZ> >(args);
  }
};
template<unsigned beta, unsigned gamma>
struct tetra_integrate_polinomial_step_2 {
  template<unsigned PX, unsigned PY, unsigned PZ, typename T>
  static T value(const abc<T> &args){
    return abstract_sum<beta, beta, T, tetra_integrate_polinomial_sum1<
      2, tetra_integrate_polinomial_step_3<gamma>, PX, PY, PZ> >(args);
  }
};
```

Первый класс tetra_integrate_monomial вычисляет интеграл от одночлена и используется при вычислении самой вложенной суммы при раскрытии третьей скобки, второй класс tetra_integrate_polinomial_step_3 раскрывает третью скобку и используется при вычислении самой вложенной суммы при раскрытии второй скобки, и, наконец, класс tetra_integrate_polinomial_step_2 раскрывает вторую скобку и используется при вычислении самой вложенной суммы при раскрытии первой скобки.

Ну и, наконец, функция верхнего уровня:

```
template<unsigned alpha, unsigned beta, unsigned gamma, typename T>
T tetra_integrate_polinomial_3(const T a[4], const T b[4], const T c[4]){
```

```
  return abstract_sum<alpha, alpha, T, tetra_integrate_polinomial_sum1<
      1, tetra_integrate_polinomial_step_2<beta, gamma>, 0, 0, 0>
    >(abc<T>(a, b, c));
}
```

Эта функция раскрывает первую скобку. При этом, как уже говорилось, для вычисления самой вложенной суммы используется класс tetra_integrate_polinomial_step_2, а в качестве параметров PX, PY и PZ передаются нули.

Для вычисления матрицы масс нужно для всех комбинаций индексов i и j повторить следующий код (пример для i = 0 и j=1):

```
{
  typedef typename tf_t::tf_type0 type_i;
  typedef typename tf_t::tf_type1 type_j;
  const unsigned
    alpha = type_i::alpha + type_j::alpha,
    beta = type_i::beta + type_j::beta,
    gamma = type_i::gamma + type_j::gamma;
  A(1, 0) = J * tetra_integrate_polynomial_3
    <alpha, beta, gamma>(a, b, c) /
    (pow<alpha>(dx)*pow<beta>(dy)*pow<gamma>(dz));
}
```

В этом коде J – якобиан преобразования тетраэдра в стандартный трёхмерный симплекс, а dx, dy и dz – характерные размеры ячейки (тетраэдра).

Матрица масс симметричная, поэтому для десяти базисных функций этот код нужно повторить 55 раз. Мы не можем сделать два вложенных цикла по переменным i и j, т.к. в этом случае переменные i и j будут известны только во время исполнения, а не во время компиляции. Этот код должен быть повторен 55 раз явно. Для этого можно воспользоваться препроцессором и библиотекой boost. В этом случае код будет выглядеть так:

```
#define CALC_A_IJ(z, j, i) { \
  typedef typename tf_t::tf_type##i type_i; \
  typedef typename tf_t::tf_type##j type_j; \
  const unsigned \
    alpha = type_i::alpha + type_j::alpha, \
    beta = type_i::beta + type_j::beta, \
    gamma = type_i::gamma + type_j::gamma; \
  A(i, j) = J * tetra_integrate_polynomial_3 \
    <alpha, beta, gamma>(a, b, c) / \
    (pow<alpha>(dx)*pow<beta>(dy)*pow<gamma>(dz)); \
}
#define CALC_A_I(z, i, unused) \
  BOOST_PP_REPEAT(BOOST_PP_INC(i), CALC_A_IJ, i)
BOOST_PP_REPEAT(BASE_FUNCTION_COUNT, CALC_A_I, ~)
#undef CALC_A_IJ
#undef CALC_A_I
```

Вычислить интеграл от многочлена, заданного как произведение трёх степенных функций, – непростая задача. Можно все 55 формул (реально их меньше, т.к. встречаются повторяющиеся степени) выписать явно, но при этом велика вероятность допустить ошибку. Описанный выше метод может поначалу показаться сложным, но он обладает несколькими важными преимуществами. Во-первых, все скобки раскрываются автоматически, и вероятность допустить ошибку минимальна. Во-вторых, при изменении числа базисных функций не нужно исправлять вообще ничего. И, в-третьих, за счёт того, что часть вычислений (биномиальные коэффициенты, интегралы от одночленов, условные операторы при вычисления степенной функции) выносится на стадию компиляции, вычисления выполняются существенно быстрее. При большом размере сетки это может быть весьма актуальным.

3.3. Вычисление значений базисных функций и их производных

Покажем теперь, как можно рассчитать значения базисных функций и частных производных от базисных функций в произвольной точке тетраэдра с координатами (x, y, z). В начале покажем, как рассчитываются значения коэффициентов разложения по базисным функциям:

```
data_type value[BASE_FUNCTION_COUNT];
const data_type xx = (x-xc)/dx, yy=(y-yc)/dy, zz=(z-zc)/dz;
#define CALC_VALUE(z, i, unused) { \
  typedef typename tf_t::tf_type##i type; \
  value[i] = pow<type::alpha>(xx) * \
    pow<type::beta>(yy) * pow<type::gamma>(zz); \
}
BOOST_PP_REPEAT(BASE_FUNCTION_COUNT,CALC_VALUE,~)
#undef CALC_VALUE
```

В начале вычисляются значения $xx=(x-x_c)/\Delta x$, $yy=(y-y_c)/\Delta y$, $zz=(z-z_c)/\Delta z$, затем для каждой базисной функции вычисляется значение $xx^{\alpha}\cdot yy^{\beta}\cdot zz^{\gamma}$. Цикл по базисным функциям разворачивается с помощью препроцессора (BOOST_PP_REPEAT). Это обеспечивает то, что показатели степеней α, β и γ для каждой из базисных функций будут известны на стадии компиляции.

Вычисление частной производной покажем на примере вычисления частной производной по переменной x от одночлена вида $f=x^{\alpha}y^{\beta}z^{\gamma}$. Значение частной производной по переменной x определяется формулой:

$$f' = \text{if } \alpha = 0 \text{ then } 0 \text{ else } \alpha\cdot x^{\alpha-1}y^{\beta}z^{\gamma}.$$

Если α – константа, известная во время компиляции, то, так же, как и для функции pow, условный оператор можно вынести на стадию компиляции. Вот как это можно сделать.

```
template<unsigned alpha, unsigned beta, unsigned gamma, typename T>
typename enable_if<alpha==0, T>::type
deriv_x(T xx, T yy, T zz, T dx){ return 0; }

template<unsigned alpha, unsigned beta, unsigned gamma, typename T>
```

```
typename enable_if<alpha!=0, T>::type
deriv_x(T xx, T yy, T zz, T dx){
  return alpha * pow<alpha - 1>(xx) / dx *
    pow<beta>(yy) * pow<gamma>(zz);
}
```

В этом примере величины xx, yy, zz те же, что и в предыдущем примере для вычисления базисных функций. Для каждого значения α видна только одна из двух функций. Во второй функции (для α!=0) результат дополнительно делится на dx, т.к. производная берётся по переменной x, а не xx, и $xx'_x=1/\Delta x$.

4. Шаблоны выражений

Шаблоны выражений (expression templates) [6] – важный инструмент шаблонного метапрограммирования, применяемый в случаях, когда нужно не просто однократно вычислить значение в соответствие с некоторым выражением, а многократно использовать цепочку вычислений, задаваемую этим выражением. Шаблоны выражений позволяют также избавиться от вспомогательных переменных для хранения промежуточных результатов, что иногда позволяет существенно сэкономить оперативную память. В данной реализации метода Галёркина шаблоны выражений используются, во-первых, в операторном методе программирования (он, по существу, основан на шаблонном метапрограммировании) и, во-вторых, в операторе объёмного интегрирования для задания интегрируемой функции. Основой шаблонов выражений является т.н. шаблонный полиморфизм.

4.1. Шаблонный полиморфизм

Полиморфизм является важным свойством объектно-ориентированного программирования. Он позволяет реализовывать алгоритмы, принимающие на вход объекты произвольных классов, обладающих общим интерфейсом. В исходной версии языка C++ для реализации полиморфизма существовал единственный механизм – виртуальные функции. В базовом классе виртуальные методы можно было продекларировать, а в дочерних – реализовать. Метод виртуальных функций имеет два существенных недостатка. Во-первых, вызов виртуального метода – относительно дорогая операция, т.к. он использует косвенную адресацию и виртуальные методы не могут быть инлайновыми. Если реализация самого метода очень короткая, то временные затраты на вызов могут быть сравнимы с затратами на исполнение самого метода. Если вызов виртуального метода осуществляется внутри многократно (например, десятки миллионов раз) повторяющегося цикла, это может привести к существенному замедлению программы. Второй недостаток состоит в том, что объект с виртуальными методами нельзя скопировать в другое адресное пространство. А при работе с CUDA это может стать актуальным.

С появлением шаблонов в языке C++ появился другой способ реализации полиморфизма – т.н. шаблонный полиморфизм. Он основан на шаблоне

программирования под названием Curiously recurring template pattern (CRTP), см. [7].

4.2. Curiously recurring template pattern

Данный шаблон программирования основан на том, что в базовый класс в качестве параметра шаблона класса передаётся конечный класс. Таким образом, ссылку на базовый класс всегда можно привести к ссылке на конечный класс и вызвать в нём любой метод (или обратиться к любой переменной). Вот как выглядит реализация базового класса:

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

Метод self как раз и делает такое приведение. Конечный класс передаёт себя в базовый класс в качестве параметра шаблона класса:

```
struct my_class : math_object_base<my_class>
{
  void some_method(){ … }
};
```

Теперь мы можем написать полиморфную функцию, принимающую в качестве параметра ссылку на объект базового класса:

```
template<class O>
void my_func(math_object_base<O> &obj){
  obj.self().some_method();
}
```

При таком подходе базовый класс «пустой», в нём кроме метода self ничего нет. В частности, нет виртуальных функций, и, соответственно, нет препятствий для копирования объекта в другое адресное пространство. Если вызываемый метод инлайновый (как в приведённом примере), то также не будет затрат на вызов метода.

Недостатком шаблонного полиморфизма является то, что при компиляции полиморфных функций (таких как my_func из примера) компилятор должен всё знать про конечный класс, что в большинстве практических случаев не является большой проблемой. Для каждого конечного класса, с которым вызывается полиморфный метод, компилятор создаёт свою версию этой полиморфной функции. Исходный текст функции – это, по сути, текст шаблона функции, заготовка, из которой при подстановке конкретного класса компилятор будет создавать конкретные (для данного класса) экземпляры функции. В каждой такой функции при вызове инлайнового метода будет подставлено тело этого метода из нужного класса.

4.3. Шаблонны выражений в методе Галёркина

Помимо того, что в данной реализации метода Галёркина использовался операторный подход, основанный на шаблонах выражений, при реализации оператора объёмного интегрирования шаблоны выражений использовались явно для реализации интегрируемых выражений. Оператор объёмного интегрирования вычисляет следующий интеграл по тетраэдру:

$$I_{kj}^{nev} = \int_{T_k} \left[f_x^n \frac{\partial \varphi_j}{\partial x} + f_y^n \frac{\partial \varphi_j}{\partial y} + f_z^n \frac{\partial \varphi_j}{\partial z} \right] dV.$$

При этом функции f^n могут быть произвольными выражениями от переменных ρ, u, v, w, p, E, задаваемыми следующей грамматикой:

«выражение»::=«элементарное выражение» | (*«выражение»*) |
«выражение»+«выражение» | *«выражение»-«выражение»* |
«выражение»«выражение»* | *«выражение»/«выражение»;*
«элементарное выражение»::=ρ|E|u|v|w|e|p;

Для реализации этой грамматики как раз и используются шаблоны выражений. Поддержку скобок «бесплатно» обеспечивает компилятор.

В начале создаётся «маркерный» класс без дополнительного функционала:

```
template<class IE>
struct integrate_expression : math_object_base<IE>{};
```

Все остальные классы выражений наследуются от этого класса. Для вычисления значения выражения объекту (выражению) передаётся ссылка на объект класса integrate_data_type, хранящий значения элементарных переменных:

```
struct integrate_data_type {
  data_type rho, E, u, v, w, e, p;
};
```

Реализацию элементарного выражения продемонстрируем на примере плотности (остальные элементарные выражения реализуются аналогично):

```
struct _rho_t : integrate_expression<_rho_t> {
  data_type operator()(const integrate_data_type &id) const {
    return id.rho;
  }
};
extern _rho_t _rho;
```

Переменная _rho является единственным экземпляром данного класса, который и нужно использовать в выражениях.

Все арифметические операции реализуются с помощью единого класса, принимающего в качестве параметров, помимо операндов операции, и саму операцию:

```
template<class IE1, class OP, class IE2>
struct integrate_expression_binary_op :
  integrate_expression<integrate_expression_binary_op<IE1, OP, IE2> >
```

```
{
  integrate_expression_binary_op(const integrate_expression<IE1> &ie1,
    OP op, const integrate_expression<IE2> &ie2) :
    ie1(ie1.self()), op(op), ie2(ie2.self()){}
  data_type operator()(const integrate_data_type &id) const {
    return op(ie1(id), ie2(id));
  }
private:
  const IE1 ie1;
  OP op;
  const IE2 ie2;
}
```

Реализацию арифметической операции с выражениями продемонстрируем на примере операции сложения:

```
template<class IE1, class IE2>
integrate_expression_binary_op<IE1, plus<data_type>, IE2>
operator+(const integrate_expression<IE1> &ie1,
  const integrate_expression<IE2> &ie2)
{
  return integrate_expression_binary_op<IE1, plus<data_type>, IE2>
    (ie1, plus<data_type>(), ie2);
}
```

Остальные арифметические операции реализуются аналогично. При этом используются объекты, реализующие арифметические операции из стандартной библиотеки: plus, minus, multiplies, divides. Класс integrate_expression_binary_op сохраняет копии передаваемых ему выражений. Ссылки (или указатели) на выражения сохранять нельзя, т.к. передаваемые переменные могут быть временными, которым не соответствуют реальные переменные в программе. Такие переменные имеют ограниченный «срок жизни».

Оператор объёмного интегрирования имеет следующие заголовок:

```
template<class G, class EOS, class IE1, class IE2, class IE3, class I>
struct volume_integral_operator :
  ugrid_operator<G, volume_integral_operator<G, EOS, IE1, IE2, IE3, I> >
```

Здесь G – класс сетки, EOS – класс, реализующий уравнение состояния, IE1, IE2, IE3 – классы интегрируемых выражений, а I – класс, реализующий интегрирование по тетраэдру методом Гаусса. Для получения оператора объёмного интегрирования служит следующая функция:

```
template<class I, class G, class EOS, class IE1, class IE2, class IE3>
volume_integral_operator<G, EOS, IE1, IE2, IE3, I>
volume_integral(const G &ugrid, const EOS &eos,
  const integrate_expression<IE1> &ie1,
  const integrate_expression<IE2> &ie2,
  const integrate_expression<IE3> &ie3
){
```

```
  return volume_integral_operator<G, EOS, IE1, IE2, IE3, I>
    (ugrid, eos, ie1, ie2, ie3);
}
```

Эта функция хороша тем, что ей обязательно требуется указать только первый параметр шаблона I, остальные параметры шаблона функции компилятор определяет автоматически, исходя из типов переданных параметров функции. Окончательно использование оператора объёмного интегрирования выглядит следующим образом:

```
R = A * (join_polynom_operator<grid_type>(ugrid)(
  volume_integral<integrate_tetra_t>
    (ugrid, eos, _rho*_u, _rho*_v, _rho*_w)(U),
  volume_integral<integrate_tetra_t>
    (ugrid, eos, _rho*_u*_u + _p, _rho*_u*_v, _rho*_u*_w)(U),
  volume_integral<integrate_tetra_t>
    (ugrid, eos, _rho*_v*_u, _rho*_v*_v + _p, _rho*_v*_w)(U),
  volume_integral<integrate_tetra_t>
    (ugrid, eos, _rho*_w*_u, _rho*_w*_v, _rho*_w*_w + _p)(U),
  volume_integral<integrate_tetra_t>
    (ugrid, eos, (_E + _p)*_u, (_E + _p)*_v, (_E + _p)*_w)(U)
) - hflow_op(U));
```

Здесь join_polynom_operator – оператор, объединяющий все пять компонент в одну структуру, а hflow_op – оператор, вычисляющий потоки через грани.

5. Заключение

В настоящей работе показывается эффективность применения методов шаблонного метапрограммирования языка C++ для решения задач математической физики. Большинство современных компиляторов с языка C++ (включая компилятор для CUDA) поддерживают использование шаблонов, поэтому их применение не нарушит переносимость программы на другие системы. Возможность применения шаблонного метапрограммирования при реализации той или иной задачи – предмет для исследования, что уже является непростой задачей. Но игра стоит свеч: чем активнее будет применяться метапрограммирование, тем больше удастся ускорить задачу и тем элегантнее будет её решение. В разрывном методе Галёркина оказалось, что возможностей для использования метапрограммирования много, соответствующим был и эффект: по сравнению с базовой реализацией метода Галёркина, которая была дана автору для сравнения результатов, ускорение на одном и том же компьютере составило 6-7 раз. Кроме того, при вычислении матрицы масс, где требовалось раскрыть произведения трёх скобок, в базовой реализации, в которой это раскрытие делалось вручную, было найдено две ошибки (опечатки). При автоматическом раскрытии скобок с использованием метапрограммирования вероятность ошибки исключена.

При реализации разрывного метода Галёркина, помимо метапрограммирования, использовался также операторный метод программирования, для чего этот метод был реализован на трёхмерных тетраэдральных сетках. Операторный метод программирования имеет два важных преимущества. Во-первых, он, используя то же самое метапрограммирование, позволяет кратко записывать в программе математические формулы с сеточными функциями. Во-вторых, он позволяет легко переносить программы на параллельные архитектуры, такие как CUDA (см. [9]) и OpenMP, в частности, Intel Xeon Phi (см. [10]). Перенос на эти архитектуры делается простой перекомпиляцией исходного текста. При включении OpenMP программа ускорилась в 4,5 раза (процессор Intel Core i7, Visual Studio 2013). На CUDA программа ускорилась по сравнению с последовательной версией в 6-7 раз (GTX-670).

Библиографический список

1. David Abrahams, Aleksey Gurtovoy. C++ Template Metaprogramming. Addison-Wesley. — 2004.
2. Bernardo Cockburn, An Introduction to the Discontinuous Galerkin Method for Convection - Dominated Problems, Advanced Numerical Approximation of Nonlinear Hyperbolic Equations (Lecture Notes in Mathematics), 1998, V. 1697, pp. 151-268.
3. М.Е.Ладонкина, О.А.Неклюдова, В.Ф.Тишкин. Использование разрывного метода Галеркина при решении задач гидродинамики. // Математическое моделирование. 2014. т.25. №1, с. 17-32
4. М.М. Краснов. Операторная библиотека для решения трёхмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA. // Математическое моделирование, 2015, т. 27, № 3, с. 109-120.
5. В.Т. Жуков, М.М. Краснов, Н.Д. Новикова, О.Б. Феодоритова. Сравнение эффективности многосеточного метода на современных вычислительных архитектурах. // Программирование, 2015, № 1, с. 21-31.
6. T. Veldhuizen, Expression Templates, C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31
7. Curiously recurring template pattern.
   URL: http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
8. Boost C++ Libraries. URL: http://www.boost.org/
9. NVidia CUDA.URL: http://www.nvidia.com/object/cuda_home_new.html
10. Intel Xeon Phi. URL: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html
11. TOP500 Supercomputer Sites. URL: http://www.top500.org
12. Суперкомпьютер "Ломоносов". URL: http://parallel.ru/cluster/lomonosov.html
13. Гибридный вычислительный кластер К-100
    URL: http://www.kiam.ru/MVS/resourses/k100.html