ИПМ им.М.В.Келдыша РАН

Онлайновая библиотека

М.М.Краснов

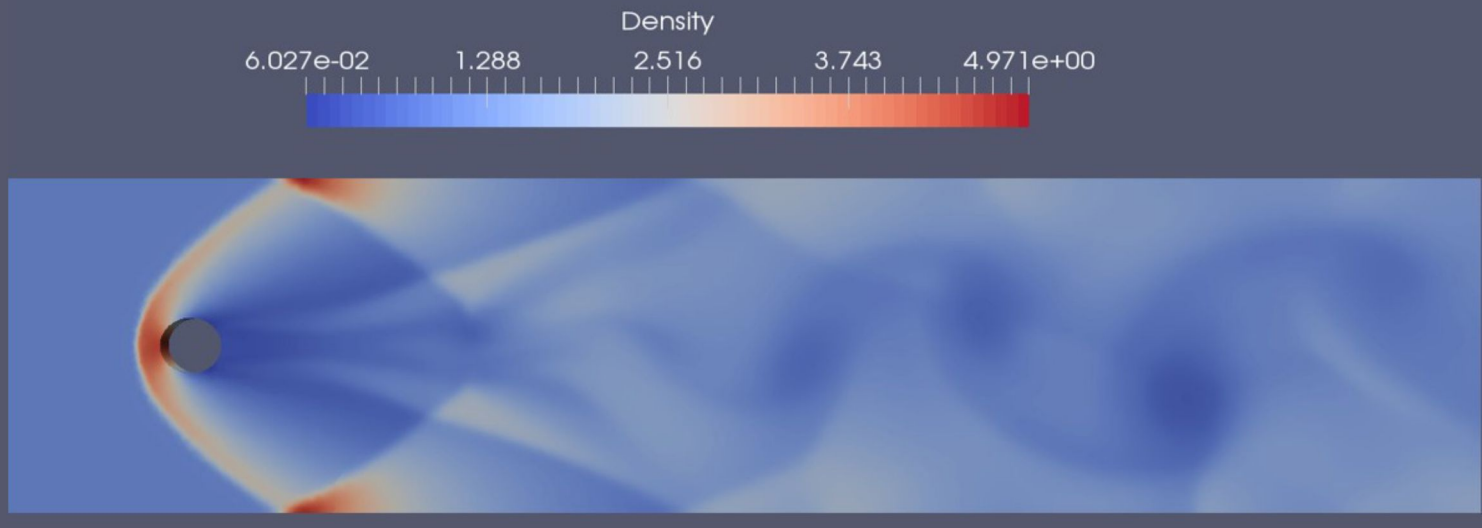
# Метапрограммирование шаблонов C++ в задачах математической физики

***Рекомендуемая форма библиографической ссылки***

Краснов М.М. Метапрограммирование шаблонов C++ в задачах математической физики. М.: ИПМ им.М.В.Келдыша, 2017. 84 с.
doi:10.20948/mono-2017-krasnov
URL: http://keldysh.ru/e-biblio/krasnov

М.М.Краснов

# Метапрограммирование шаблонов C++ в задачах математической физики



**М.М. Краснов**

# Метапрограммирование шаблонов C++ в задачах математической физики

**ИПМ им. М.В. Келдыша**
**Москва 2017**

УДК 519.6
ББК 22.193
К 78

**Краснов Михаил Михайлович**
Метапрограммирование шаблонов C++ в задачах математической физики. – М.: ИПМ им. М.В. Келдыша, 2017. – 84 с.

Рассматривается применение метапрограммирования шаблонов языка C++ для упрощения записи алгоритмов и для переноса части вычислений (различных целочисленных констант, например, биномиальных коэффициентов) на стадию компиляции, что теоретически позволяет ускорить выполнение программ и может быть весьма актуальным для численного решения задач математической физики. Метапрограммирование шаблонов позволяет автоматизировать некоторые сложные вычисления, такие, например, как раскрытие скобок в сложных выражениях и вычисление символьных производных от формул. Это помогает избавиться от трудно обнаруживаемых ошибок, возникающих при проведении подобных вычислений вручную (на бумаге) из-за невнимательности.

Для математиков-программистов, занимающихся численным моделированием, и студентов вузов старших курсов, желающих глубже изучить возможности языка C++ для решения численных задач.

Рецензенты:

Лацис Алексей Оттович, д.ф.-м.н.
Гаранжа Владимир Анатольевич, д.ф.-м.н., профессор РАН

Издание осуществлено при поддержке
Российского научного фонда, проект № 17-71-30014

*Электронная версия книги в формате PDF*
*доступна по адресу http://keldysh.ru/e-biblio/krasnov*

ISBN 978-5-98354-036-1 

# Оглавление

# 1. Введение

Под метапрограммированием в языке C++ подразумеваются вычисления на стадии компиляции. Основным инструментом для этого являются шаблоны классов, называемые в этом случае (в случае их использования для вычислений на стадии компиляции) метафункциями. Параметрами таких метафункций служат параметры шаблонов классов, а значения метафункций (их может быть несколько) хранятся в статических (static) переменных или определениях типов (typedef-ax) класса. Язык C++ накладывает определённые ограничения на то, что может быть передано в качестве параметров шаблонов, и на типы статических переменных, значения которых могут быть вычислены на стадии компиляции. Это могут быть константы целочисленных типов и произвольные типы (как встроенные в язык простые типы, так и классы). В частности, нельзя передавать в качестве параметров шаблонов и вычислять на стадии компиляции константы вещественных типов, а также шаблоны классов. Если статическая переменная не целочисленного типа (например, double), то её значение может быть вычислено только на стадии исполнения и на стадии компиляции недоступно. Целочисленных типов в языке C++ довольно много. К ним относятся булевский тип (bool), символьные типы (char и unsigned char), а также знаковые и беззнаковые целые типы разных размеров (short, int, long, long long). Таким образом, метапрограммирование оперирует целочисленными константами и произвольными типами. В остальном метафункции не имеют никаких ограничений по возможностям для вычислений.

Язык метапрограммирования (если его рассматривать как самостоятельную часть языка C++) является полным в том смысле, в каком полной является машина Тьюринга или лямбда-исчисление, т.е. на нём можно реализовать любую вычислимую функцию. Этот язык можно также рассмотреть с точки зрения функциональных языков программирования. Он является чисто функциональным языком, в нём отсутствует оператор присваивания значений переменным, переменные возможны только в качестве параметров метафункций. Как и в других чисто функциональных языках программирования (LISP, Haskell), единственно возможный способ организации циклов – это рекурсия.

При подстановке значений параметров в шаблон класса получается класс. При этом для разных наборов значений параметров компилятор создаёт разные классы. Кроме того, для некоторых конкретных значений параметров можно создавать «особые» классы, имеющие другую реализацию, чем общий шаблон класса. Такие «особые» классы называются специализациями классов. Специализации классов могут быть частичными (когда конкретные значения указываются для части параметров) или полными (когда конкретные значения указы-

ваются для всех параметров). Под полным шаблоном класса подразумевается совокупность основного определения шаблона класса и всех его специализаций. В частности, специализации классов используются для организации рекурсивных циклов. В рекурсивных функциях важно вовремя завершить рекурсию, в метапрограммировании для этого как раз и служат специализации классов для предельных значений параметров шаблонов.

Шаблоны (классов и функций) – один из самых сложных элементов языка C++. Они появились в языке не сразу и первоначально были задуманы для того, чтобы можно было писать универсальные коллекции (которые могут хранить значения произвольных типов) и алгоритмы (например, алгоритм сортировки набора объектов произвольного типа). Метапрограммирование – другое, не менее важное применение шаблонов. Для полноценного освоения всех возможностей шаблонов автор отсылает любознательного читателя к описанию и многочисленным учебникам по языку C++ (например, [1-6]), а также хорошей книге Дэвида Абрахамса и Алексей Гуртового (см. [7]). Цель настоящей работы – показать на конкретных примерах, как метапрограммирование шаблонов может помочь вынести часть вычислений на стадию компиляции и, таким образом, ускорить (иногда существенно) выполнение программ, что бывает особенно важно при численном решении задач математической физики, которые могут считаться весьма долго (по многу часов и даже дней). Таким образом, данная работа рассчитана в первую очередь на математиков, решающих задачи численного моделирования и готовых к глубокому освоению возможностей языка C++ ради повышения эффективности выполнения своих программ. Кроме того, применения метапрограммирования шаблонов может упростить решение некоторых логически сложных задач и уменьшить вероятность ошибок в программах, связанных с невнимательностью. Такие ошибки часто бывает очень трудно найти. В качестве примеров таких задач можно привести автоматическое раскрытие скобок в сложных выражениях и символьное дифференцирование произвольных выражений.

Одним из самых интересных применений метапрограммирования шаблонов (см. [8]) являются шаблоны выражений (expression templates, см. [9]). При этом средствами языка C++ реализуется некоторый (обычно несложный) проблемно-ориентированный язык (Domain Specific Language, DSL) для написания выражений той или иной природы. В качестве примера использования шаблонов выражений можно привести символьное дифференцирование.

Существует весьма распространённое мнение, что объектно-ориентированное программирование не может быть эффективно применено в численном моделировании, в частности, из-за того, что «программы, написанные на голом C, считаются быстрее, чем написанные на C++, особенно с использованием объектно-

ориентированного программирования». Автор не согласен с данным мнением и в данной работе, как и в других своих работах, пытается показать, что «правильное» использование возможностей языка C++ не только не замедляет работу программ, но позволяет уменьшить как время написания и отладки программ, так и время их исполнения за счёт выноса части вычислений на стадию компиляции. В этой жизни за всё приходится платить. Здесь платой является необходимость освоения этих самых новых возможностей языка C++, порой весьма непростых и непривычных. Но игра стоит свеч. Всякое знание расширяет возможности. Время, затраченное на освоение новых методов программирования, вернётся за счёт сокращения времени на написание и отладку программ. Кроме того, программы могут начать работать быстрее.

Важным вопросом является переносимость программ. Все мы хотим, чтобы наши программы работали под любыми операционными системами, чтобы можно было, например, писать и отлаживать программу на небольших сетках на домашнем компьютере под Microsoft Windows и Microsoft Visual Studio, а затем запускать её на кластере, где стоит Linux и или gcc, или Intel C++. Наверное, в наше время уже не осталось систем, где нет компилятора с языка C++, причём с поддержкой шаблонов. Так что проблем с переносимостью при использовании метапрограммирования шаблонов быть не должно. Это относится, в частности, и к программированию для графических ускорителей CUDA: компилятор nvcc для CUDA также поддерживает шаблоны. Немного сложнее обстоит дело с версией языка C++. Язык постоянно развивается, особенно в последние годы. Выход новых версий языка планируется каждые 3 года. Первым таким существенным изменением после 1998 года (когда были введены шаблоны) и 2003 года (касалось в основном не языка, а библиотеки, в частности в библиотеку были введены элементы метапрограммирования) была версия языка от 2011 года (т.н. C++11). Позже вышел стандарт языка C++14, ожидаются новые стандарты C++17 и C++20. Все современные версии компиляторов (включая Microsoft C++, Intel C++, gcc и nvcc) поддерживают версию языка C++11 (и даже C++14), но, к сожалению, на многих кластерах компиляторы не обновляются (или обновляются редко), и, если программа использует возможности новых версий языка, проблема переносимости может возникнуть. С другой стороны, в новых стандартах языка появились как новые языковые конструкции, так и новые библиотечные функции (в том числе метафункции), облегчающие применение метапрограммирования шаблонов. Если Ваш компилятор не поддерживает стандарт языка C++11, то для восполнения недостающих конструкций можно использовать библиотеку boost (см. [10]). Boost многие годы является своеобразным «полигоном», на котором обкатываются новые возможности языка C++, прежде всего его стандартной библиотеки. Вошедшие в стандартную библиотеку

языка C++ новых версий конструкции, как правило, перекочёвывают туда из библиотеки boost. Причём под влиянием библиотеки boost меняется не только стандартная библиотека, но и сам язык. В частности, в языке появилось переменное количество параметров шаблонов классов (variadic templates). Некоторые задачи с использованием возможностей языка C++11 можно решить более элегантно, чем с использованием библиотеки boost. В дальнейшем изложении основные конструкции будут показываться для языка C++11, а дополнительно будет показано, как это можно сделать с помощью библиотеки boost.

Несколько слов по поводу графических ускорителей CUDA (см. [11, 12]). Как уже говорилось выше, компилятор для CUDA поддерживает шаблоны, и никаких проблем с использованием метапрограммирования нет. Более того, начиная с версии 7.0 компилятор nvcc поддерживает стандарт языка C++11. Но это не сильно помогает, так как для переноса программы на CUDA её всё равно нужно сильно переписать. Так можно ли написать программу один раз так, чтобы компилировать и запускать её можно было в том числе и на CUDA? Существует несколько решений этой задачи, один из них – стандарт OpenACC (см. [13]). Этот подход аналогичен OpenMP (см. [14]), т.е. решение заключается в автоматическом распараллеливании витков цикла. Применительно к OpenACC это означает, что исходные данные будут скопированы на CUDA, там параллельно обработаны, а затем результаты будут скопированы обратно в главный (host) процессор. Такой подход не представляется оптимальным. Автором предлагается альтернативный, т.н. «сеточно-операторный» подход к программированию, в котором при работе с CUDA данные изначально расположены в памяти CUDA, и копирование «туда-сюда» не требуется. Исходные данные при необходимости копируются в память CUDA в начале работы программы, а в конце работы копируются обратно в главный процессор. Сеточно-операторный подход к программированию также основывается на шаблонном метапрограммировании и будет изложен в данной работе.

Далее в работе будут показаны многочисленные примеры использования метапрограммирования шаблонов, от простейших до достаточно сложных. Все примеры будут сопровождаться подробными пояснениями.

## 2. Простейшие метафункции

В данном разделе мы покажем простейшие метафункции, которые в дальнейшем будут использоваться для более сложных вычислений. Как уже говорилось, метафункция с точки зрения языка C++ – это шаблон класса, параметры метафункции – это параметры этого шаблона класса, в качестве них могут передаваться или целочислен-

ные константы, значения которых известны во время компиляции, или типы.

### 2.1. Факториал

Пусть число N известно во время компиляции. Тогда значение N! во время компиляции можно вычислить 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;
};
// Пример обращения:
const unsigned f5 = factorial<5>::value; // 120
```

Первый шаблон класса factorial является общим случаем и реализует рекурсивный вызов по известному правилу. Второй класс – это т.н. «полная специализация» шаблона класса и служит для завершения рекурсии. Напомним, чем полная специализация шаблона класса отличается от частичной. В полной специализации задаются конкретные значения всех параметров шаблона класса (в данном случае одного). Таким образом, полная специализация задаёт определение одного конкретного класса, при этом список параметров шаблона пуст (как в данном случае). При частичной специализации список параметров шаблона не пустой. Результат метафункции хранится в static const переменной value типа unsigned. Если число N окажется слишком большим, то результат в переменную типа unsigned не поместится. В этом случае компилятор выдаст ошибку. Но в реальности в численных методах слишком большие числа для факториала не встречаются, и таких проблем не возникает.

### 2.2. Биномиальные коэффициенты

Пусть числа N и K известны во время компиляции. Тогда биномиальные коэффициенты в соответствие с известной формулой 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:
const unsigned i = C<5, 2>::value; // 10
```

### 2.3. Интеграл от одночлена

Пусть требуется вычислить трёхмерный интеграл по стандартному трёхмерному симплексу от одночлена вида $x^\alpha y^\beta z^\gamma$, причём числа α, β и γ известны во время компиляции. Напомним, что стандартный трёхмерный симплекс – это тетраэдр с вершинами в точках (0, 0, 0), (1, 0, 0), (0, 1, 0) и (0, 0, 1). Вычисление таких интегралов нам понадобится в дальнейшем, когда мы будем вычислять интегралы от произвольных многочленов по произвольному тетраэдру. Такой интеграл вычисляется аналитически и равен

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

Это число вещественное (меньше единицы) и на стадии компиляции вычислено быть не может. Но обратное к нему число целое (это ясно из комбинаторных соображений). Вот текст метафункции, вычисляющий на стадии компиляции число, обратное интегралу от одночлена:

```
template<unsigned alpha, unsigned beta, unsigned gamma>
struct monomial_3d {
  static const unsigned value =
    factorial<alpha + beta + gamma + 3>::value /
      (factorial<alpha>::value * factorial<beta>::value
        * factorial<gamma>::value);
};
```

### 2.4. Вычисление типа значения итератора

В стандартной библиотеке языка C++ есть функция accumulate, вычисляющая сумму элементов между двумя итераторами. Прототип этой функции следующий:

```
template<typename IT, typename T>
T accumulate(IT first, IT last, T init);
```

Эта функция в качестве параметров принимает два итератора (начала и конца) и начальное значение. Она имеет один недостаток: тип возвращаемого значения определяется типом начального значения, и поэтому нужно быть очень внимательным, чтобы не сделать досадную ошибку. Поясним это на примере следующего кода:

```
double a[3] = { 1.23, 2.34, 3.45 };
double sum = std::accumulate(a, a + 3, 0);
```

Теперь, внимание, вопрос: чему будет равно значение переменной sum? Правильный ответ – 6 (кто не верит, может проверить). Результат, очевидно, совсем не тот, который мы ожидаем (ожидаемый результат 7.02). Причина неправильного результата в том, что в качестве начального значения мы передали ноль целого типа. Соответственно, результат функции также целого типа. Более того, округление до целого делается на каждом шаге внутреннего цикла. Правиль-

ный вызов функции accumulate в данном случае должен быть следующий:

```
double sum = std::accumulate(a, a + 3, 0.);
```

Всего лишь одна точка, а какая большая разница! Теперь поставим цель написать свою функцию accumulate (назовём её, например, my_accumulate), которая тип возвращаемого значения брала бы не из типа начального значения, а из типа значения, который хранится в коллекции, на которую указывают итераторы. При этом нужно, чтобы, как и в функцию accumulate, в нашу функцию можно было передавать как итераторы из стандартной библиотеки языка C++, так и указатели на массив (как в нашем примере). Итераторы из стандартной библиотеки хранят тип значения в определении типа value_type. Решить задачу можно с помощью следующей метафункции get_value_type:

```
template<typename IT>
struct get_value_type {
  typedef typename IT::value_type type;
};
template<typename T>
struct get_value_type<T*>{
  typedef T type;
};
template<typename T>
struct get_value_type<const T*>{
  typedef T type;
};
```

Основное определение метафункции рассчитано на итераторы из стандартной библиотеки, а две частичных специализации рассчитаны на указатели (простой и const). Метафункция get_value_type примечательна для нас в двух отношениях. Во-первых, эта функция принимает на вход в качестве параметра и возвращает в качестве результата (в определении типа type) не целочисленные константы, а типы. Во-вторых, эта метафункция демонстрирует применение частичной специализации шаблона класса. С помощью этой метафункции нашу функцию my_accumulate можно написать следующим образом:

```
template<typename IT, typename T>
typename get_value_type<IT>::type my_accumulate(
  IT first, IT last, T init)
{
  typename get_value_type<IT>::type result = init;
  for (; first != last; ++first)
    result += *first;
  return result;
}
```

Данный пример демонстрирует ещё и другой принцип: если нужно что-то узнать о типе, правильно написать метафункцию, возвращающую нужную информацию (добавить косвенность), а не доставать эту информацию из типа напрямую. В этом случае при появлении нового типа всегда можно написать специализацию класса (метафункции) для нового типа и таким простым способом решить задачу. Поясним это на примере. Допустим, что появился какой-то новый итератор (например, из новой библиотеки), назовём его, например, new_iterator, для которого наша функция my_accumulate не работает из-за того, что в этом новом итераторе тип значения хранится не в переменной (определении типа) value_type, а, например, в переменной data_type. Изменить это мы не можем, так как новый класс не наш. Но мы легко можем написать специализацию метафункции get_value_type для этого нового типа:

```
template<>
struct get_value_type<new_iterator> {
  typedef typename new_iterator::data_type type;
};
```

Таким образом, мы всегда можем адаптировать правильно (с использованием метафункций) написанную чужую для нас функцию для работы с новым (чужим для нас) типом данных, написав для нужной (чужой) метафункции свою специализацию для этого нового типа.

Кстати, необходимости писать метафункцию get_value_type не было. В стандартной библиотеке для работы с итераторами есть метафункция std::iterator_traits, возвращающая в переменной value_type как раз то, что нам нужно. С её использованием прототип нашей функции выглядел бы так:

```
template<typename IT, typename T>
typename std::iterator_traits<IT>::value_type my_accumulate(
  IT first, IT last, T init);
```

## 2.5. Упражнения

1. Написать метафункцию, вычисляющую числа Фибоначчи. В качестве параметра метафункция должна принимать порядковый номер числа Фибоначчи, а возвращать в переменной value значение числа.
2. Написать метафункцию monomial_2d, вычисляющую число, обратное к интегралу по двумерному симплексу от одночлена вида $x^{\alpha}y^{\beta}$. Аналитическое выражение для интеграла имеет вид:

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

## 3. 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 на языке C 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 число N известно во время компиляции. Так как x – вещественное число, то и результат – также вещественное число и на стадии компиляции вычислен быть не может. Кроме того, значение 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, при включенном оптимизаторе рекурсивные вызовы будут подставлены (inline), и вычисление степенной функции будет очень быстрым. Покажем, как это можно сделать.

В начале обратим внимание на одну из важных идиом языка C++ – т.н. SFINAE, Substitution Failure Is Not An Error (неудача при подстановке не является ошибкой, см. [15]). Эта идиома используется в метафункции enable_if из стандартной библиотеки языка C++, которая 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. Покажем реализацию этой метафункции:

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

В 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. Второй параметр метафункции enable_if (тип) может быть опущен, в этом случае подра-

зумевается значение void. Поясним сказанное на примере функции 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
```

Благодаря SFINAE при любом значении N видимой будет ровно одна реализация функции pow. Обратите внимание также на то, что второй шаблонный параметр функции pow при вызове можно не указывать. Он определяется компилятором автоматически по типу параметра x.

## 4. Более сложные примеры

### 4.1. Простые числа

На этапе компиляции несложно проверить целое число на простоту. Вот текст метафункции is_prime, принимающей в качестве параметра целое число и возвращающей значение true или false в зависимости от его простоты:

```
template<int n, int i>
struct is_prime0 : std::conditional<(n % i) == 0,
  std::false_type, is_prime0<n, i - 1>
>::type {};

template<int n>
struct is_prime0<n, 1> : std::true_type {};

template<int n>
struct is_prime : is_prime0<n, n / 2>{};
```

Метафункция is_prime использует вспомогательную функцию is_prime0 с двумя параметрами, которая рекурсивно (а это, как мы уже говорили, единственный способ организации циклов в метапрограммировании) проверяет, делится ли число n на число i и меньшие него.

Если число i достигает единицы, то срабатывает частичная специализация метафункции is_prime0, которая говорит, что число n простое (возвращает значение true). Если же n делится на i, большее единицы, то оно не простое. Метафункция is_prime0, в свою очередь, использует метафункцию conditional из стандартной библиотеки, реализующую условный оператор (if). Вот как устроена эта метафункция:

```
template<bool b, typename T1, typename T2>
struct conditional {
  typedef T1 type;
};
template<typename T1, typename T2>
struct conditional<false, T1, T2> {
  typedef T2 type;
};
```

Метафункция conditional возвращает тип T1, если первый параметр равен true и T2, если первый параметр равен false.

В дальнейшем мы будем широко использовать новые возможности стандарта языка C++11, т.к. он уже достаточно широко распространён. Если же он по какой-то причине не доступен, то можно пользоваться библиотекой boost, в которой многое из того, что нам нужно, реализовано средствами стандарта языка C++98. При этом всё может получиться чуть более громоздко. Например, метафункция is_prime0 с использованием библиотеки boost выглядела бы так:

```
template<int n, int i>
struct is_prime0 : boost::mpl::if_c<(n % i) == 0,
  boost::mpl::false_, is_prime0<n, i - 1>
>::type {};

template<int n>
struct is_prime0<n, 1> : boost::mpl::true_ {};
```

На следующем шаге определим метафункцию prime_at, принимающую в качестве параметра порядковый номер простого числа (начиная с нуля) и возвращающую значение простого числа с указанным порядковым номером:

```
template<unsigned n>
struct find_prime : std::conditional<is_prime<n>::value,
  std::integral_constant<unsigned, n>,
  find_prime<n + 2>
>::type {};

template<unsigned n>
struct prime_at : find_prime<prime_at<n - 1>::value + 2>{};

template<>
struct prime_at<0> : std::integral_constant<unsigned, 2>{};
```

```
template<>
struct prime_at<1> : std::integral_constant<unsigned, 3>{};
```

Метафункция prime_at использует вспомогательную метафункцию find_prime, осуществляющую поиск простого числа, начиная с указанного в качестве параметра числа. Метафункция prime_at передаёт ей в качестве параметра значение предыдущего простого числа плюс два. Два первых простых числа (с порядковыми номерами ноль и один) задаются явно с помощью полных специализаций класса prime_at.

### 4.2. Ошибка при компиляции

Следующий пример в какой-то степени шуточный, так как не имеет практического применения. Но он очень показательный, так как ещё больше раскрывает возможности метапрограммирования. Звучит он так: написать программу, которая компилируется с ошибкой, причём в тексте сообщения об ошибке должны быть напечатаны первые 20 (к примеру) простых чисел. Из текста программы должно быть видно, что эти простые числа вычисляются, а не просто так забиты автором в текст программы.

Простые числа мы находить уже умеем (см. предыдущий пункт). Для этого мы умеем использовать циклы времени компиляции (через рекурсию) и условный оператор времени компиляции (метафункция std::conditional). Теперь нам предстоит создать коллекцию времени компиляции (вектор целых чисел). При этом мы для простоты будем использовать возможности стандарта языка C++11, в частности, переменное число параметров шаблона класса (variadic templates). Вот исходный текст вектора целых чисел времени компиляции:

```
template<int... I> struct int_vector {};
template<class IntVector, int i> struct push_back;

template<int... I, int i>
struct push_back<int_vector<I...>, i> {
  typedef int_vector<I..., i> type;
};

template<class IntVector, int n> struct at;

template<int i, int... I, int n>
struct at<int_vector<i, I...>, n> :
  at<int_vector<I...>, n - 1> {};

template<int i, int... I>
struct at<int_vector<i, I...>, 0> :
  std::integral_constant<int, i> {};
```

На старом стандарте языка вектор целых чисел реализовать тоже можно, но, во-первых, это будет существенно длиннее, и, во-вторых, в

такой реализации будет ограничение на число параметров шаблона (например, не более 10, максимальное число задаётся при компиляции). Итак, для вектора целых чисел мы определяем две операции (по аналогии с вектором из стандартной библиотеки STL): push_back и at. Первая добавляет новый элемент в конец, а вторая получает значение элемента по индексу.

Теперь мы определяем метафункцию primes. Эта метафункция в качестве параметра принимает количество простых чисел, а в типе type возвращает вектор простых чисел указанного размера. Вот определение этой метафункции:

```
template<unsigned n> struct primes {
  typedef typename push_back<
    typename primes<n - 1>::type,
    prime_at<n - 1>::value
  >::type type;
};
template<> struct primes<0>{
  typedef int_vector<> type;
};
```

Видно, как эта метафункция работает. Она вызывает рекурсивно саму себя с параметром n-1 (при этом получается вектор простых чисел длины n-1) и добавляет с помощью метафункции push_back в конец вектора следующее простое число. Для завершений рекурсии предназначена специализация, возвращающая пустой вектор.

Покажем теперь, как этим можно пользоваться:

```
typedef primes<20>::type p20; // вектор из первых двадцати простых чисел
static_assert(at<p10, 4>::value == 11, "p4 == 11");
static_assert(prime_at<5>::value == 13, "p5 == 13");
```

Осталось получить ошибку при компиляции. Это совсем просто. Заводим переменную-вектор и присваиваем ей что-нибудь. Так как оператор присваивания для вектора целых чисел не определён, получаем ошибку при компиляции:

```
typedef primes<20>::type p20;
p20 p;
p = 5; // оператор присваивания не определён.
```

При компиляции компилятором g++ получаем ошибку:

```
error: no match for «operator=» (operand types are «p20 {aka
int_vector<2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53, 59, 61, 67, 71>}» and «int»)
  p = 5;
    ^
```

Компилятор от Microsoft выдаёт аналогичную ошибку. Как мы видим, поставленная задача решена. Смысл этой забавной задачи число познавательный – на её примере мы чему-то научились.

Несколько замечаний по поводу скорости компиляции. Мы перекладываем часть вычислений на компилятор, и это, естественно, увеличивает время компиляции. Если вычисления небольшие, то это увеличение может быть «на глаз» практически не заметно. Но при больших числах оно может стать очень большим. Например, при вычислении первых десяти простых чисел увеличение времени практически не заметно, компиляция длится доли секунды. При вычислении ста простых чисел оно уже заметно и составляет около пяти секунд, а при вычислении двухсот простых чисел компилятор Microsoft выдаёт ошибку о том, что код слишком сложный и он не справляется. Вывод – не стоит возлагать на метапрограммирование «слишком» сложные задачи. При этом критичной является не алгоритмическая сложность вычислений, а их объём, прежде всего – глубина рекурсии, компиляторы, как правило, «ломаются» именно на этом.

### 4.3. Упражнения

1. Написать программу, которая компилируется с ошибкой и в тексте ошибки выдаёт первые 20 чисел Фибоначчи.

## 5. Цикл времени компиляции

### 5.1. Вычисление интеграла от бинома.

Продемонстрируем применение цикла времени компиляции на примере вычисления интеграла по стандартному двумерному симплексу от бинома – выражения вида $(ax+by)^N$, где a и b – вещественные числа, известные только во время исполнения программы, а N – неотрицательная целочисленная константа, известная во время компиляции. 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 мы уже умеем считать во время компиляции биномиальные коэффициенты и интегралы от одночленов по стандартному двумерному симплексу. Кроме того, мы умеем вычислять степенную функцию оптимальным образом, вынося все условные операторы на стадию компиляции. Единственное, чего нам не хватает, это организации цикла. При этом на каждом витке цикла текущий индекс цикла (число K) должен быть известен во время компиляции, иначе мы не сможем вычислить ни биномиальные коэффициенты, ни интегралы от одночленов, ни степенную функцию. То есть обычный цикл времени

исполнения нам не подходит. Организовать цикл времени компиляции можно разными способами, как средствами препроцессора, так и с помощью метапрограммирования. С помощью препроцессора это можно сделать с использованием макроса BOOST_PP_REPEAT из библиотеки boost. При этом тело цикла просто будет повторено нужное число раз с разными значениями индекса цикла. Но это не наш путь, так как мы изучаем метапрограммирование. Итак, вот тело функции (обычной, времени исполнения) meta_loop, исполняющей цикл времени компиляции указанное число раз:

```
template<unsigned N, unsigned I, class Closure>
typename std::enable_if<(I == N)>::type
meta_loop0(Closure &closure){}

template<unsigned N, unsigned I, class Closure>
typename std::enable_if<(I < N)>::type
meta_loop0(Closure &closure){
      closure.template apply<I>();
      meta_loop_<N, I + 1>(closure);
}
template<unsigned N, class Closure>
void meta_loop(Closure &closure){
      meta_loop0<N, 0>(closure);
}
```

Функция meta_loop шаблонная, и в качестве первого параметра шаблона она принимает общее число итераций, которые надо сделать. В качестве обычного параметра (времени исполнения) она принимает объект (т.н. «замыкание», closure) произвольного класса, в котором на каждой итерации нужно вызвать шаблонный метод apply без параметров, которому в качестве параметра шаблона передаётся индекс текущей итерации. Функция meta_loop использует вспомогательную шаблонную функцию meta_loop0, принимающую два шаблонных параметра – общее число итераций и номер текущей итерации (начиная, как принято в языках C/C++, от нуля). Естественно, в качестве начального индекса цикла передаётся значение ноль. Функция meta_loop0 в том случае, если индекс цикла меньше общего числа итераций, вызывает в объекте closure метод apply, передав ему в качестве параметра шаблона индекс текущей итерации, а затем рекурсивно вызывает саму себя с индексом цикла на единицу больше (как мы уже говорили, в метапрограммировании рекурсия – единственно возможный способ организации циклов). Когда индекс цикла достигает общего числа итераций, срабатывает первая версия функции meta_loop0, которая ничего не делает и, таким образом, завершает рекурсию.

Теперь нам надо организовать суммирование. Мы также будем использовать пользовательский объект «замыкание» (closure), но теперь будем предполагать, во-первых, что в нём (точнее, в его классе) определён тип value_type, указывающий тип вычисляемого значения,

и, во-вторых, что у объекта есть шаблонный метод value без параметров, принимающий в качестве параметра шаблона индекс текущей итерации и возвращающий значение члена ряда с указанным индексом. Вот тело соответствующей функции abstract_sum, вычисляющий значение ряда указанной длины:

```
template<class Closure>
struct abstract_sum_closure
{
  typedef typename Closure::value_type value_type;
  abstract_sum_closure(Closure &closure) :
    closure(closure), result(value_type()){}

  template<unsigned I>
  void apply(){
    result += closure.template value<I>();
  }
  Closure &closure;
  value_type result;
};

template<unsigned N, class Closure>
typename Closure::value_type abstract_sum(Closure &closure)
{
  abstract_sum_closure<Closure> my_closure(closure);
  meta_loop<N>(my_closure);
  return my_closure.result;
}
```

Функция abstract_sum реализована через функцию meta_loop, поэтому она вначале создаёт своё замыкание (класса abstract_sum_closure), удовлетворяющее требованиям (концепции) этой функции. А именно, в классе abstract_sum_closure есть шаблонный метод apply без параметров, принимающий в качестве параметра шаблона текущий индекс итерации.

Ну и, наконец, для реализации нашей задачи (интегрирования бинома) нужно реализовать замыкание для бинома и саму функцию binom:

```
template<unsigned N, typename T>
struct binom_closure
{
  typedef T value_type;

  binom_closure(value_type a, value_type b) : a(a), b(b){}

  template<unsigned K>
  value_type value() const {
    return C<N, K>::value * pow<K>(a) * pow<N-K>(b) /
      monomial_2d<K, N-K>::value;
  }
```

```
private:
  const value_type a, b;
};

template<unsigned N, typename T>
T binom(T a, T b)
{
  binom_closure<N, T> closure;
  return abstract_sum<N + 1>(closure);
}
```

Так как в разложении бинома степени N будет N+1 слагаемых, то в функцию abstract_sum в качестве параметра шаблона передаётся число N+1. Значение параметра шаблона T можно не передавать, оно определяется компилятором автоматически из типов параметров a и b. Пример обращения к функции binom:

```
const double a = 1.2, b = 3.4;
const unsigned N = 5;
const double bi = binom<N>(a, b);
```

### 5.2. Автоматическое раскрытие скобок. Матрица масс

Двумерный интеграл от бинома, на самом деле, уже демонстрирует т.н. автоматическое раскрытие скобок, для этого используется цикл времени компиляции. Тем не менее, этот пример, хотя и наглядный, но достаточно простой и не демонстрирует всю мощь описываемого приёма. В разрывном методе Галёркина (см [16, 17]) на трёхмерных тетраэдральных сетках требуется решить существенно более сложную задачу. В этом методе для каждого тетраэдра, из которых состоит сетка, строятся т.н. матрица масс – матрица интегралов по тетраэдру от произведения двух базисных функций, т.е. интегралов вида: $a_{ij} = \iiint \varphi_i \varphi_j d\,\Omega$, где интегрирование ведётся по тетраэдру. Каждая базисная функция имеет вид:

$$\varphi_i = \left(\frac{x - x_c}{\Delta x}\right)^{\alpha_i} \cdot \left(\frac{y - y_c}{\Delta y}\right)^{\beta_i} \cdot \left(\frac{z - z_c}{\Delta z}\right)^{\gamma_i},$$

где $x_c$, $y_c$, $z_c$ – координаты центра тетраэдра; $\Delta x$, $\Delta y$, $\Delta z$ – характерные размеры тетраэдра. В трёхмерном случае в зависимости от порядка многочлена разложения может быть одна (кусочно-постоянное разложение), четыре (линейное разложение) или десять (квадратичное разложение) базисных функций. В двумерном случае их будет соответственно одна, три или шесть. Для описания базисных функций имеется структура bf_t. Покажем, как она задаётся, на примере квадратичных функций:

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

```
};
struct bf_t {
  typedef base_function_base<0, 0, 0> type0;
  typedef base_function_base<1, 0, 0> type1;
  typedef base_function_base<0, 1, 0> type2;
  typedef base_function_base<0, 0, 1> type3;
  typedef base_function_base<2, 0, 0> type4;
  typedef base_function_base<0, 2, 0> type5;
  typedef base_function_base<0, 0, 2> type6;
  typedef base_function_base<1, 1, 0> type7;
  typedef base_function_base<1, 0, 1> type8;
  typedef base_function_base<0, 1, 1> type9;
};
```

Для доступа к параметрам базисной функции по её порядковому номеру имеется соответствующая метафункция:

```
template<unsigned i> struct get_bf_type;

#define GET_BF_TYPE(z, i, unused) \
  template<> struct get_bf_type<i> \
  { typedef bf_t::type##i type; };

BOOST_PP_REPEAT(10, GET_BF_TYPE, ~)
#undef GET_BF_TYPE
```

В данном случае также организуется цикл времени компиляции, но уже с помощью препроцессора. Для каждого значения i от нуля до девяти создаётся своя специализация класса (метафункции) get_bf_type, возвращающая нужный тип. Например, для i=0 это будет:

```
template<>
struct get_bf_type<0>{
  typedef bf_t::type0 type;
};
```

С помощью макроса BOOST_PP_REPEAT такие определения создаются для всех значений i от нуля до девяти включительно.

Подынтегральное выражение матрицы масс 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

$a_1=x_1-x_4$, $a_2=x_2-x_4$, $a_3=x_3-x_4$, $a_4=x_4$,

$b_1=y_1-y_4$, $b_2=y_2-y_4$, $b_3=y_3-y_4$, $b_4=y_4$,

$c_1=z_1-z_4$, $c_2=z_2-z_4$, $c_3=z_3-z_4$, $c_4=z_4$.

После вычисления интеграла по стандартному трёхмерному симплексу для получения правильного значения интеграла по произвольному тетраэдру значение интеграла нужно умножить на значение якобиана преобразования, который в данном случае равен объёму гексаэдра, построенному на трёх рёбрах тетраэдра, выходящих из одной (любой) вершины. Этот объём равен абсолютному значению следующего определителя:

$$Det = \begin{vmatrix} x_1 - x_4 & x_2 - x_4 & x_3 - x_4 \\ y_1 - y_4 & y_2 - y_4 & y_3 - y_4 \\ z_1 - z_4 & z_2 - z_4 & z_3 - z_4 \end{vmatrix}$$

J=|Det|.

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:

$$(a_1x + a_2y + a_3z + a_4)^{\alpha} \cdot (b_1x + b_2y + b_3z + b_4)^{\beta} \cdot (c_1x + c_2y + c_3z + c_4)^{\gamma}$$

В этом выражении мы, как обычно, полагаем, что α, β и γ – целочисленные неотрицательные константы, известные во время компиляции. Данное выражение существенно сложнее, чем бином Ньютона, кроме того, интегрирование мы будем вести не по двумерному симплексу, а по трёхмерному. Наша цель – «автоматически» раскрыть все скобки, после чего интегрирование всего выражения сведётся к интегрированию одночленов по стандартному трёхмерному 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 одну скобку, нужно сосчитать три вложенных суммы. Будем раскрывать скобки, начиная с первой. После раскрытия первой скобки исходное выражение примет вид:

$$\sum_i s_i x^{p_i} y^{q_i} z^{r_i} \cdot (b_1 x + b_2 y + b_3 z + b_4)^{\beta} \cdot (c_1 x + c_2 y + c_3 z + c_4)^{\gamma}$$

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

После раскрытия второй скобки исходное выражение примет вид:

$$\sum_j t_j x^{u_j} y^{v_j} z^{w_j} \cdot (c_1 x + c_2 y + c_3 z + c_4)^{\gamma}$$

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

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 каждой суммы будем использовать описанную выше функцию abstract_sum. При этом нам надо будет написать несколько замыканий, причём многим из них для вычислений нужны массивы коэффициентов a, b и c. Поэтому имеет смысл написать один базовый класс для всех наших замыканий, который будет хранить эти массивы чисел:

```
struct tetra_integrate_polynomial_base0
{
  typedef double value_type;
  typedef const value_type *const_pointer;

protected:
  tetra_integrate_polynomial_base0(const value_type a[],
    const value_type b[], const value_type c[]) : a(a), b(b),
c(c){}

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

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

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

private:
  const_pointer a, b, c;
};
```

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

или c). Следующий общий класс в иерархии классов замыканий позволяет хранить следующий шаг вычислений:

```
template<class NEXT_STEP>
struct tetra_integrate_polynomial_base :
  tetra_integrate_polynomial_base0
{
  typedef typename tetra_integrate_polynomial_base0::value_type
    value_type;

  tetra_integrate_polynomial_base(const value_type a[],
    const value_type b[], const value_type c[],
    const NEXT_STEP &next_step) :
    tetra_integrate_polynomial_base0(a, b, c),
    next_step(next_step){}

protected:
     const NEXT_STEP &next_step;
};
```

Теперь покажем набор из трёх классов (замыканий), вычисляющих члены ряда для трёх вложенных сумм:

```
template<unsigned J, unsigned step, unsigned PX, unsigned PY,
  unsigned PZ, class NEXT_STEP>
struct tetra_integrate_polynomial_sum3 :
  tetra_integrate_polynomial_base<NEXT_STEP>
{
  typedef tetra_integrate_polynomial_base<NEXT_STEP> super;
  typedef typename super::value_type value_type;

  tetra_integrate_polynomial_sum3(const value_type a[],
    const value_type b[], const value_type c[],
    const NEXT_STEP &next_step) : super(a, b, c, next_step){}

  template<unsigned K>
  value_type value() const {
    return C<J, K>::value *
      pow<K>(super::template get<step>()[0]) *
      pow<J - K>(super::template get<step>()[1]) *
      super::next_step.template value<PX + K, PY + J - K, PZ>();
  }
};
template<unsigned I, unsigned step, unsigned PX, unsigned PY,
  unsigned PZ, class NEXT_STEP>
struct tetra_integrate_polynomial_sum2 :
  tetra_integrate_polynomial_base<NEXT_STEP>
{
  typedef tetra_integrate_polynomial_base<NEXT_STEP> super;
  typedef typename super::value_type value_type;

  tetra_integrate_polynomial_sum2(const value_type a[],
```

```
    const value_type b[], const value_type c[],
    const NEXT_STEP &next_step) : super(a, b, c, next_step){}

  template<unsigned J>
  value_type value() const
  {
    const tetra_integrate_polynomial_sum3
      <J, step, PX, PY, PZ + I - J, NEXT_STEP>
      closure(super::a, super::b, super::c, super::next_step);
    return C<I, J>::value *
      pow<I - J>(super::template get<step>()[2]) *
      abstract_sum<J + 1>(closure);
  }
};
template<unsigned N, unsigned step, unsigned PX, unsigned PY,
  unsigned PZ, class NEXT_STEP>
struct tetra_integrate_polynomial_sum1 :
  tetra_integrate_polynomial_base<NEXT_STEP>
{
  typedef tetra_integrate_polynomial_base<NEXT_STEP> super;
  typedef typename super::value_type value_type;

  tetra_integrate_polynomial_sum1(const value_type a[],
    const value_type b[], const value_type c[],
    const NEXT_STEP &next_step) : super(a, b, c, next_step){}

  template<unsigned I>
  value_type value() const
  {
    const tetra_integrate_polynomial_sum2
      <I, step, PX, PY, PZ, NEXT_STEP>
      closure(super::a, super::b, super::c, super::next_step);
    return C<N, I>::value *
      pow<N - I>(super::template get<step>()[3]) *
      abstract_sum<I + 1>(closure);
  }
};
```

Каждая из этих функций вычисляет частичную сумму ряда. Функция tetra_integrate_polynomial_sum1 для вычисления одного члена ряда использует функцию tetra_integrate_polynomial_sum2, которая, в свою очередь, использует функцию tetra_integrate_polynomial_sum3. Последняя для вычисления члена ряда использует внешний объект next_step, в котором вызывается шаблонный метод value, которому в качестве параметров шаблона передаются три неотрицательных целых числа – степени, которые нужно дополнительно прибавить к степеням переменных x, y и z при интегрировании одночленов (те самые числа $p_i$, $q_i$, $r_i$, $u_j$, $v_j$, $w_j$, о которых говорилось выше).

Поясним смысл шаблонных параметров классов:

первый параметр (J, I или N) – степень, в которую возводится многочлен;

step – шаг, порядковый номер раскрываемой скобки. Предназначен для извлечения набора коэффициентов;

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;

next_step – внешний класс, используемый для вычисления одного члена ряда самой внутренней суммы (классом tetra_integrate_polynomial_sum3). Для шага 1 это раскрытые второй скобки, для шага 2 – раскрытие третьей скобки, а для шага 3 – интегрирование одночлена.

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

```
struct tetra_integrate_monomial
{
  typedef double value_type;

  template<unsigned PX, unsigned PY, unsigned PZ>
  value_type value() const {
    return value_type(1) / monomial_3d<PX, PY, PZ>::value;
  }
};
template<unsigned gamma>
struct tetra_integrate_polynomial_step_3 :
  tetra_integrate_polynomial_base0
{
  typedef tetra_integrate_polynomial_base0 super;
  typedef typename super::value_type value_type;

  tetra_integrate_polynomial_step_3(const value_type a[],
    const value_type b[], const value_type c[]) : super(a, b,
c){}

  template<unsigned PX, unsigned PY, unsigned PZ>
  value_type value() const
  {
    const tetra_integrate_monomial next_step;
    const tetra_integrate_polynomial_sum1
      <gamma, 3, PX, PY, PZ, tetra_integrate_monomial> closure
        (super::a, super::b, super::c, next_step);
    return abstract_sum<gamma + 1>(closure);
  }
};
template<unsigned beta, unsigned gamma>
struct tetra_integrate_polynomial_step_2 :
  tetra_integrate_polynomial_base0
{
```

```
  typedef tetra_integrate_polynomial_base0 super;
  typedef typename super::value_type value_type;

  tetra_integrate_polynomial_step_2(const value_type a[],
    const value_type b[], const value_type c[]) : super(a, b,
c){}

  template<unsigned PX, unsigned PY, unsigned PZ>
  value_type value() const
  {
    typedef tetra_integrate_polynomial_step_3<gamma> NEXT_STEP;
    const NEXT_STEP next_step(super::a, super::b, super::c);
    const tetra_integrate_polynomial_sum1<beta, 2, PX, PY, PZ,
NEXT_STEP> closure(super::a, super::b, super::c, next_step);
    return abstract_sum<beta + 1>(closure);
  }
};
```

Первый класс tetra_integrate_monomial вычисляет интеграл от одночлена и используется при вычислении самой вложенной суммы при раскрытии третьей скобки, второй класс tetra_integrate_polynomial_step_3 раскрывает третью скобку и используется при вычислении самой вложенной суммы при раскрытии второй скобки, и, наконец, класс tetra_integrate_polynomial_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>
double tetra_integrate_polynomial_3(const double a[4],
  const double b[4], const double c[4])
{
  typedef tetra_integrate_polynomial_step_2<beta, gamma>
NEXT_STEP;
  const NEXT_STEP next_step(a, b, c);
  const tetra_integrate_polynomial_sum1<alpha, 1, 0, 0, 0,
NEXT_STEP>
    closure(a, b, c, next_step);
  return abstract_sum<alpha + 1>(closure);
}
```

Эта функция раскрывает первую скобку. При этом, как уже говорилось, для вычисления самой вложенной суммы используется класс tetra_integrate_polynomial_step_2, а в качестве параметров PX, PY и PZ передаются нули.

Для заполнения матрицы масс нужно сделать два вложенных цикла (по строкам и столбцам). Мы, естественно, будем это делать с помощью функции meta_loop. Для этого нужно создать два класса замыкания. Вот их текст:

```
template<unsigned i>
```

```
struct calc_A {
  typedef typename get_bf_type<i>::type type_i;

  calc_A(matrix_t &A, data_type J, const double a[],
    const double b[], const double c[], const vector_t &dx) :
    A(A), J(J), a(a), b(b), c(c), dx(dx){}

  template<unsigned j>
  void apply(){
    typedef typename get_bf_type<j>::type 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.value_x()) *
        pow<beta>(dx.value_y()) *
        pow<gamma>(dx.value_z())
      );
  }
private:
  matrix_t &A;
  const double J, *a, *b, *c;
  const vector_t &dx;
};

struct calc_A_i {
  calc_A_i(matrix_t &A, double J, const double a[],
    const double b[],const double c[], const vector_t &dx) :
    A(A), J(J), a(a), b(b), c(c), dx(dx){}

  template<unsigned i>
  void apply(){
    calc_A<i> closure(A, J, a, b, c, dx);
    meta_loop<i + 1>(closure);
  }
private:
  matrix_t &A;
  const double J, *a, *b, *c;
  const vector_t &dx;
};
// Функция верхнего уровня
void calc_matrix(matrix_t &A, double J, const double a[],
  const double b[],const double c[], const vector_t &dx)
{
  calc_A_i closure(A, J, a, b, c, dx);
  meta_loop<BASE_FUNCTIONS_COUNT>(closure);
}
```

В этом коде J – якобиан преобразования тетраэдра в стандартный трёхмерный симплекс, а dx – вектор характерных размеров ячейки (тетраэдра).

Описанный метод вычисления матрицы масс 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 изменений практически нет. 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.

# 6. Механизмы метапрограммирования шаблонов

## 6.1. Шаблонный полиморфизм

Шаблонный полиморфизм основан на шаблоне проектирования под названием Curiously Recurring Template Pattern, CRTP (см. [16]). Как известно, при появлении языка C++, когда в языке ещё не было шаблонов, полиморфизм реализовывался только 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 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 O>
struct math_object_base {
  O& self(){
    return static_cast<O&>(*this);
  }
  const O& self() const {
    return static_cast<const O&>(*this);
  }
};
```

Этот класс (math_object_base) мы 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 полиморф-

ных 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.

## 6.2. 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 способ задания выражений, основанный на шаблонах, под названием «шаблоны выражений» (expression templates, см. [9]). С помощью этого механизма можно, оставаясь в рамках языка C++, определить свой 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 более общая 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 левую ассоциативность арифметических операций. 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 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. Предположим, что все наши конечные классы выражений вычисляют своё значение от одной переменной (назовём её x), причём тип значения выражения (для простоты) – double. Простейшие выражения (в соответствие с нашей грамматикой) – это константа и переменная. Вот определения этих простейших выражений:

```
struct constant : expression<constant>
{
  constant(double value) : value(value){}

  double operator()(double x) const {
    return value; // Возвращаемое значение не зависит от значе-
ния переменной.
  }
private:
  double value;
};

struct variable : expression<variable>
{
  double operator()(double x) const {
    return x;
  }
};
```

Следующее по сложности выражение – это отрицание выражения (класс negate). Класс отрицаемого выражения передаётся как параметр шаблона класса. Кроме того, мы для произвольного выражения определяем оператор отрицания (operator-). Вот исходный текст отрицания выражения:

```
template<class E>
struct negate : expression<negate<E> >
{
  negate(const expression<E> &expr) : expr(expr.self()){}

  double operator()(double x) const {
    return -expr(x);
  }
```

```
private:
  const E expr;
};
template<class E>
negate<E> operator-(const expression<E> &expr){
  return negate<E>(expr);
}
```

Выражению отрицания нужно сохранить выражение, которое оно отрицает, для последующих вычислений. При этом мы не имеем права сохранять ссылку на это выражение, т.к. это выражение может быть временным и для него может не быть переменной, ссылку на которую можно сохранять. Поэтому приходится делать копию выражения. В нашем случае это не страшно, т.к. все выражения занимают мало памяти. Далее (в описании операторного подхода к программированию) будет рассказано, что делать, если объекты могут быть большими и их копии делать нельзя.

Следующее по сложности – это арифметические операции с выражениями. Они уже оперируют с двумя выражениями, кроме того, появляется арифметическая операция (сложение, вычитание, умножение или деление). Для всех арифметических операций мы сделаем единый класс, а саму операцию передадим как параметр. Вот текст класса для бинарного выражения:

```
template<class E1, class OP, class E2>
struct binary_expression :
  expression<binary_expression<E1, OP, E2> >
{
  binary_expression(const expression<E1> &expr1,
    const OP &op, const expression<E2> &expr2) :
    expr1(expr1.self()), op(op), expr2(expr2.self()){}

  double operator()(double x) const {
    return op(expr1(x), expr2(x));
  }
private:
  const E1 expr1;
  const OP op;
  const E2 expr2;
};

#define DEFINE_BIN_OP(oper, OP) \
  template<class E1, class E2> \
  binary_expression<E1, std::OP<double>, E2> operator oper \
  (const expression<E1> &expr1, const expression<E2> &expr2){ \
    return binary_expression<E1, std::OP<double>, E2> \
      (expr1, std::OP<double>(), expr2); \
  } \
template<class E> \
 binary_expression<E, std::OP<double>, constant> operator oper \
```

```
  (const expression<E> &expr, double value){ \
    return binary_expression<E, std::OP<double>, constant> \
      (expr, std::OP<double>(), constant(value)); \
  } \
template<class E> \
 binary_expression<constant, std::OP<double>, E> operator oper \
  (double value, const expression<E> &expr){ \
    return binary_expression<constant, std::OP<double>, E> \
      (constant(value), std::OP<double>(), expr); \
  }

DEFINE_BIN_OP(+, plus)
DEFINE_BIN_OP(-, minus)
DEFINE_BIN_OP(*, multiplies)
DEFINE_BIN_OP(/, divides)
```

Макрос DEFINE_BIN_OP определяет арифметическую операцию для двух произвольных выражений, а также для выражения и константы и константы и выражения.

Покажем теперь, как можно вычислять математические функции:

```
template<class E>
struct func_expression : expression< func_expression<E> >
{
  typedef double (*func_t)(double);

  func_expression(const expression<E> &expr, func_t func) :
    expr(expr.self()), func(func){}

  double operator()(double x) const {
    return func(expr(x));
  }
private:
  const E expr;
  func_t func;
};

#define DEFINE_FUNC(func) \
  template<class E> \
  func_expression<E> func(const expression<E> &expr){ \
    return func_expression<E>(expr, std::func); \
  }

DEFINE_FUNC(sin)
DEFINE_FUNC(cos)
DEFINE_FUNC(tg)
DEFINE_FUNC(ctg)
DEFINE_FUNC(exp)
DEFINE_FUNC(log)
DEFINE_FUNC(sqrt)
```

Теперь мы можем строить выражения любой сложности, передавать их в качестве параметров, многократно производить по ним вычисления и т.д.

```
template<class E>
void f(const expression<E> &expr0)
{
  const E &expr = expr0.self();
  std::cout << expr(3) << std::endl;
  std::cout << expr(1.5) << std::endl;
}
int main(int argc, char* argv[])
{
  variable x;
  f(sin(x * x + M_PI));
  auto expr2 = 5 * cos(-x * (x + 1));
  f(expr2);
  return 0;
}
```

## 7. Размерные величины

Часто в вычислениях используются безразмерные физические величины. Например, в качестве скорости используется отношение реальной скорости к скорости звука в среде (например, в газе), т.н. число Маха, в качестве давления – отношения реального давления к нормальному атмосферному давлению и т.д. Но бывает, что используются и реальные физические размерные величины, правда, при этом в программе всё равно используются обычные типы данных (float, double), не имеющие размерностей. В физике на операции с размерностями накладываются определённые ограничения, прежде всего на сложение и вычитание. Эти операции можно выполнять только с величинами одинаковой размерности. При умножении и делении физических величин получаются величины новых размерностей. Нарушение этих ограничений в физике запрещено, но в программе такое нарушение может возникнуть (например, по ошибке), и никто, ни на стадии компиляции, ни на стадии исполнения, эту ошибку не заметит. Такие ошибки бывает очень трудно обнаружить. Кроме того, использование безразмерных типов данных для обозначения размерных величин ухудшает читаемость программы, так как тип данных (скорость, объём и т.п.), хранящийся в той или иной переменной, часто из имени переменной может быть непонятен.

Метапрограммирование позволяет легко решить эту задачу. Можно ввести новые типы данных, содержащие в качестве метаданных (данных времени компиляции) размерность и использовать их вместо обычных простых типов данных. При этом использование этих размерных типов данных не будет увеличивать ни объём занимаемой памяти, ни время выполнения программы. Покажем, как это можно

сделать. Для работы со скалярными размерными величинами (плотность, давление) введём класс quantity, а с векторными (скорость, ускорение, сила) – vector_quantity. Оба класса принимают два шаблонных параметра – тип хранимого значения (например, float или double) и собственно размерность:

```
template<typename T, class Dimensions>
struct quantity
{
  typedef T value_type;
  … // Реализация класса
private:
  value_type m_value;
};

template<typename T, class Dimensions>
struct vector_quantity
{
  typedef T value_type;
  … // Реализация класса
private:
  value_type m_value_x, m_value_y, m_value_z;
};
```

В качестве второго шаблонного параметра классов передаётся специальный тип quantity_dim, принимающий в параметрах шаблона и хранящий в метаданных семь целых чисел, соответствующий семи основным физическим величинам: масса, длина, время, сила тока, температура, сила света и количество вещества. Единицы физических величин (например, граммы или килограммы для массы, метры или сантиметры для длины) в библиотеке не определены, но подразумеваются одинаковыми:

```
template<int N1, int N2, int N3, int N4, int N5, int N6, int N7>
struct quantity_dim {
  static const int value1 = N1;
  static const int value2 = N2;
  static const int value3 = N3;
  static const int value4 = N4;
  static const int value5 = N5;
  static const int value6 = N6;
  static const int value7 = N7;
};
```

Для величин одинаковой размерности определены арифметические операции сложения и вычитания:

```
template<typename T, class D>
quantity<T, D>
operator+(const quantity<T, D>& x, const quantity<T, D>& y){
  return quantity<T, D>(x.value() + y.value());
```

```
}
template<typename T, class D>
quantity<T, D>
operator-(const quantity<T, D>& x, const quantity<T, D>& y){
  return quantity<T, D>(x.value() - y.value());
}
```

Для величин разных размерностей эти операции не определены и при их использовании будут вызывать ошибку при компиляции. Для умножения и деления размерных величин имеются две метафункции для сложения (add_dimensions) и вычитания (sub_dimensions) размерностей:

```
template<class D1, class D2> struct add_dimensions;

template<
  int N1, int N2, int N3, int N4, int N5, int N6, int N7,
  int M1, int M2, int M3, int M4, int M5, int M6, int M7
> struct add_dimensions <
  quantity_dim<N1, N2, N3, N4, N5, N6, N7>,
  quantity_dim<M1, M2, M3, M4, M5, M6, M7>
>{
  typedef quantity_dim<N1 + M1, N2 + M2, N3 + M3, N4 + M4,
    N5 + M5, N6 + M6, N7 + M7> type;
};

template<class D1, class D2> struct sub_dimensions;

template<
  int N1, int N2, int N3, int N4, int N5, int N6, int N7,
  int M1, int M2, int M3, int M4, int M5, int M6, int M7
> struct sub_dimensions <
  quantity_dim<N1, N2, N3, N4, N5, N6, N7>,
  quantity_dim<M1, M2, M3, M4, M5, M6, M7>
>{
  typedef quantity_dim<N1 - M1, N2 - M2, N3 - M3, N4 - M4,
    N5 - M5, N6 - M6, N7 - M7> type;
};
```

Обе метафункции на вход принимают две размерности, а возвращают в типе type результирующую размерность. Вот как эти метафункции используются при умножении и делении размерных величин:

```
template<typename T, class D1, class D2>
quantity<T, typename add_dimensions<D1, D2>::type>
operator*(const quantity<T, D1>& x, const quantity<T, D2>& y)
{
  typedef typename add_dimensions<D1, D2>::type dim;
  return quantity<T, dim>(x.value() * y.value());
}
```

```
template<typename T, class D1, class D2>
quantity<T, typename sub_dimensions<D1, D2>::type>
operator/(const quantity<T, D1>& x, const quantity<T, D2>& y)
{
  typedef typename sub_dimensions<D1, D2>::type dim;
  return quantity<T, dim>(x.value() / y.value());
}
```

В классе quantity определены операции сложения и вычитания с величинами той же размерности, умножение и деление на безразмерные скаляры. Для класса vector_quantity определён более широкий набор операций. Операция умножения двух векторных величин определена как векторное умножение, операция деления двух векторных величин не определена, определены операции умножения и деления векторной величины на скалярную размерную величину. Также определена операция скалярного умножения двух векторных величин, возвращающая скалярную размерную величину (через оператор &) и операция покомпонентного произведения двух векторных величин (через оператор ^).

Чтобы задать размерность, нужно указать степени каждой из семи основных величин. Например, размерности ускорения и давления задаются так:

```
typedef quantity_dim<0,1,-2,0,0,0,0> acceleration;
typedef quantity_dim<1,-1,-2,0,0,0,0> pressure;
```

Для задания размерной величины нужно указать тип хранимого значения и размерность. Например, переменная, хранящая ускорение, задаётся так:

```
vector_quantity<double, acceleration> a;
```

# 8. Символьное дифференцирование

## 8.1. 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 приближённое значение производной в узле сетки вычисляется в соответствии с конечно-разностным 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 $\Delta x$ – шаг сетки (постоянный).

При автоматическом дифференцировании (см. [17]) 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. Излагаемые в данной главе идеи для символьного дифференцирования вполне могут быть использованы для автоматического дифференцирования при определении формул для производных. Кроме того, автоматическое дифференцирование легко реализуется в рамках сеточно-операторного подхода к программированию, излагаемому в следующей главе.

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 главы. 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.

## 8.2. Реализация выражений

Выражения для символьного дифференцирования также, как и простые выражения для вычислений, реализуются с помощью шаблонов выражений, о которых было рассказано в предыдущей главе. Важное отличие состоит в том, что выражения в предыдущей главе могли только вычислять своё значение в зависимости от значения переменной. Теперь нам этого мало. Каждое выражение должно также уметь возвращать выражение для своей символьной производной. Кроме того, позже мы будем рассматривать выражения не от одной, а от нескольких переменных, соответственно, в этом случае нужно уметь вычислять значение выражения от значений нескольких переменных, а также выражения для символьных частных производных.

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, 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 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>();
}

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. Это связано с тем, что идиома SFINAE, реализуемая метафункцией enable_if, применима только к шаблонным функциям и методам, а метод diff не шаблонный.

Умножение и деление реализуются аналогично сложению и вычитанию и также включают многочисленные оптимизации (например, умножение 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++.

### 8.3. 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).

### 8.4. 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);
```

## 8.5. 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

$\mathbf{x}=\mathbf{x_0}+\mathbf{dx}$.

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. Для хранения вектора значений используется класс val_array, который пронаследован от класса std::array из стандартной библиотеки языка и дополнительно реализует арифметические операции с вектором значений. Реализация этого класса предлагается читателю в качестве упражнения.

Функция newton вначале 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. [10]), который может хранить значение произвольного типа. Чтобы из объекта класса boost::any можно было извлечь хранящееся в нём значение, нужно знать тип это-

го значения. Мы можем это узнать следующим образом. Тип выражения для I-ого уравнения определяется так:

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

Тип выражения для частной производной I-ого уравнения 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-ого уравнения 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. Для этого используется уже описанная выше шаблонная функция meta_loop. Покажем реализацию функции jacobian, вычисляющую якобиан нашей системы уравнений:

```
template<class E, size_t I, 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 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<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 size_t N = sizeof...(E);
  square_matrix<boost::any, N> result;
  jacobian0_closure<E...> closure(tp, result);
  meta_loop<N>()(closure);
  return result;
}
```

Для реализации якобиана используются два класса-замыкания: jacobian0_closure для реализации цикла по уравнениям и jacobian1_closure для реализации цикла по переменным для одного уравнения. Метод apply класса jacobian0_closure для каждой (I-й) итерации цикла по уравнениям «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 уравнения и общее число уравнений (и переменных). Метод apply класса jacobian1_closure уже имеет всю необходимую информацию для вычисления выражения для частной производной I-го уравнения 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. Покажем реализацию функции eval_tuple:

```
template<typename T, class... E>
struct eval_tuple_closure {
  static const std::size_t N =
    std::tuple_size<std::tuple<E...> >::value;

  eval_tuple_closure(const std::tuple<E...> &tp,
    const val_vector<T, N> &x,  val_vector<T, N> &result) :
    tp(tp), x(x), result(result){}
```

```
  template<unsigned I>
  void apply(){
    result[I] = std::get<I>(tp)(x);
  }

private:
     const std::tuple<E...> &tp;
     const val_vector<T, N> &x;
     val_vector<T, N> &result;
};

template<typename T, class... E>
val_vector<T, sizeof...(E)>
eval_tuple(const std::tuple<E...> &tp,
  const val_vector<T, sizeof...(E)> &x)
{
  const std::size_t N = sizeof...(E);
  val_vector<T, N> result;
  eval_tuple_closure<T, E...> closure(tp, x, result);
  meta_loop<N>(closure);
  return result;
}
```

Написание функции eval_jacobian предлагается читателю в качестве упражнения.

### 8.6. Решение задачи на нахождение экстремальной точки

В данной задаче имеется одно уравнение от нескольких переменных (в виде выражения). Требуется найти набор значений этих переменных, при которых все частные производные данного уравнения 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 нашего уравнения. 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 исходным уравнением. Приведём исходный текст соответствующей функции:

```
template<typename E, typename T, 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, size_t I, 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, 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<N>()(closure);
  }
private:
  const E &expr;
  square_matrix<boost::any, N> &result;
};
template<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<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. Написание функций eval_grad и eval_hessian предлагается читателю в качестве упражнения.

### 8.7. 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 $A_\psi$ – 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. [18]). При этом первое является, в известном смысле, аппроксимацией второго. Однако возможны и другие подходы (см. [19]).

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. [18])

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

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);
```

## 8.8. 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 обнаруживаемая 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.

# 9. Сеточно-операторный подход к программированию

## 9.1. Введение

В задачах математического моделирования широко используются сеточные функции – величины, определённые в каждом узле некоторой сетки (см. [20]). Для численного решения задач математического моделирования с этими сеточными функциями делаются определённые преобразования. В математической литературе эти преобразования описываются операторами, такими, например, как оператор Лапласа, градиента, дивергенции, ротора. Кроме того, в уравнениях могут встречаться линейные комбинации и композиции операторов. Например, при решении эллиптического уравнения многосеточным методом итерирующий оператор для двух уровней имеет вид (см. [20]):

$Q = S_p(I - PA_H^{-1}RA_h)\,S_p$,

где $A_H$ и $A_h$ – операторы (матрицы) на подробной и грубой сетках соответственно, P – оператор интерполяции (продолжения), R – оператор сборки (проектирования), $S_p$ – сглаживающий оператор, p – число пре- и пост-сглаживающих шагов (число применений оператора $A_h$), I – единичный оператор.

В теоретических работах описание алгоритмов выглядит очень компактно и элегантно. При реальном программировании текст программы выглядит гораздо более громоздко и часто требует дополнительной памяти для сохранения промежуточных результатов вычислений.

Ещё одной немаловажной проблемой является то, что в настоящее время большое распространение получили вычислительные комплексы с нетрадиционной (не фон-Неймановской) архитектурой, основу вычислительной мощности которых составляют графические ускорители NVIDIA CUDA (Compute Unified Device Architecture, см.

[11]) или новейшие процессоры Intel Xeon Phi с 60 ядрами на одном кристалле (см. [21]). Например, в списке top500 самых мощных суперкомпьютеров (см [22]) за ноябрь 2016 года на первом месте стоит суперкомпьютер Sunway TaihuLight (Китай – NRCPC), основанный на многоядерных процессорах Sunway SW26010 260C, на втором – Tianhe-2 (Китай - NUDT), основанный на процессорах Intel Xeon Phi 31S1P, а на третьем – Titan (США - Cray Inc), основанный на графических ускорителях NVIDIA K20x. Многие Российские суперкомпьютеры, например, самый мощный Российский суперкомпьютер «Ломоносов» (см. [23]) и вычислительный кластер К-100 в ИПМ им. М.В. Келдыша РАН (см. [24]) также содержат графические ускорители. Например, на К-100 на каждом из 64-х узлов, помимо двух основных 6-ядерных процессоров Intel Xeon X5670 стоят по три платы NVIDIA Fermi C2050 (по 448 GPU и 2,8 Гбайт памяти на каждом). Эффективное же использование этих весьма внушительных вычислительных мощностей затруднено, т.к. требует освоения большого объёма новой и непривычной информации о методах программирования на них.

## 9.2. Краткий обзор работ

Обозначенные выше проблемы (громоздкая запись формул по сравнению с математической литературой и сложность переноса программ на нетрадиционные архитектуры, прежде всего на CUDA) не новы и есть ряд работ, направленных на решение указанных проблем.

Одним из важных направлений развития системного программного обеспечения становится создание систем, упрощающих для прикладного программиста написание программ, использующих вычислительные возможности суперкомпьютеров с новейшими вычислителями. Основная цель при создании подобных систем – обеспечить переносимость исходного текста программ на новые архитектуры. При этом исходный текст может снабжаться псевдокомментариями или прагмами, с помощью которых можно управлять работой специализированных компиляторов. Или же один и тот же исходный текст можно компилировать разными компиляторами и получать код для разных архитектур. Такие работы активно ведутся в том числе и в ИПМ им. М.В. Келдыша РАН. Например, язык НОРМА (см. [25-30]) и система DVM (см. [31]). Среди других работ можно упомянуть высокоуровневый язык Liszt (см. [32]) и OpenACC (см. [13]).

Второе направление связано с попыткой упростить написание сложных программ за счёт выноса часто повторяющегося кода в отдельные модули или классы и затем многократного их использования. При этом исходный текст становится более структурированным и понятным. Так как выносимый код обычно используется внутри многократно повторяющегося тела цикла, важным становится вопрос эффективности. В частности, этот код нельзя выносить в отдельные обычные функции, т.к. вызов функции – дорогая по времени опера-

ция. При программировании на языке C++ функции могут быть объявленными инлайновыми (inline). В этом случае код этих функций вставляется компилятором в точке вызова. При использовании шаблонов функций и классов компилятор такие функции и методы таких классов делает инлайновыми автоматически. При хорошо работающем оптимизаторе полученный машинный код по своей производительности может не уступать коду, полученному с помощью компилятора с языка Фортран. Одной из лучших библиотек, эффективно использующих метапрограммирование шаблонов, является библиотека Blitz++ (см. [33, 34]).

Есть системы, упрощающие запись вычислений за счёт реализации матрично-векторной алгебры, в которых оперирование векторами и матрицами производится как с едиными объектами. К таким системам можно отнести, например, систему Matlab (см. [35]) и язык SIMIT (см. [36]). Запись сложных манипуляций с матрицами и векторами в этих системах очень компактна и не уступает по своей краткости записи в математической литературе.

Существуют также системы для решения задач математического моделирования с уже готовыми решателями различных задач. Как правило, пользователь может добавлять в них свои расширения. Среди таких систем можно упомянуть библиотеку PETSc (см. [37]) и пакет OpenFOAM (см. [38]).

Интересный вариант решения проблемы переносимости программы на различные архитектуры предложен в работе [39]. В системе PyFR управляющая (главная) программа пишется на высокоуровневом языке Python (см. [40]). На нём легко решаются многие общие проблемы, такие, как считывание параметров из файла конфигурации, управление памятью, обработка ошибок. Эти и многие другие вещи на языке Python пишутся гораздо быстрее, чем на C/C++. Кроме того, для это языка написано множество библиотек, в том числе для работы с MPI, CUDA, OpenCL, Intel Xeon Phi и множество других. С другой стороны, Python – интерпретируемый язык, и численные методы на нём писать нельзя, т.к. они будут слишком долго работать. Для того, чтобы создать быстро работающий, и, кроме того, переносимый код, в системе PyFR вычислительные модули (ядра, kernels) пишутся на специальном относительно простом проблемно-ориентированном языке (Domain Specific Language, DSL), реализованном с помощью библиотеки шаблонов Mako (см. [41]). Эти вычислительные модули компилируются главной программой при запуске, причём, в зависимости от выбранной архитектуры вычислителя, могут компилироваться для обычного процессора (CPU) с использованием OpenMP, CUDA, OpenCL или Intel Xeon Phi (offload режим). При желании можно написать этот компилятор и для других архитектур. После компиляции модулей они могут быть вызваны напрямую из программы на языке Python. Из недостатков этого подхода можно отметить, во-первых,

необходимость освоения новых языков (Python, Mako, DSL), и, во-вторых, то, что в нём никак не решается проблема краткой записи выражений (операторы).

Перечисленные работы в той или иной степени позволяют решить одну из обозначенных проблем, но никакая из них не решает обе задачи в комплексе. Предложенный автором сеточно-операторный подход (см. [42, 43]) как раз и направлен на такое комплексное решение. С одной стороны, он позволяет записывать формулы в текстах программ почти столь же кратко и элегантно, как в математической литературе, а с другой стороны позволяет легко переносить программы на нетрадиционные архитектуры почти без изменений. В частности, введено понятие программного оператора, аналогичного математическому понятию оператора. Основой сеточно-операторного подхода является метапрограммирование шаблонов.

## 9.3. Общее описание сеточно-операторного подхода

### 9.3.1. Назначение

Сеточно-операторный подход к программированию предназначен для упрощенной записи вычислений на произвольных сетках. Основными объектами, с которыми позволяет работать библиотека, написанная в соответствие с данным подходом, являются: вычисляемый объект (evaluable object), сеточная функция (grid function), сеточный оператор (grid operator) и сеточный вычислитель (grid evaluator), который получается в результате «применения» сеточного оператора к вычисляемому объекту. К вычисляемым объектам относятся сеточные функции и сеточные вычислители. Для сеточных операторов реализована своя арифметика, при этом порождаются новые составные сеточные операторы. Для сеточных вычислителей также реализована своя арифметика с другими сеточными вычислителями и скалярными величинами, при этом порождаются новые составные сеточные вычислители.

В дальнейшем изложении слово «сеточный» в словосочетаниях «сеточный оператор» и «сеточный вычислитель» часто будет опускаться, чтобы не загромождать текст.

Вычисляемые объекты можно присваивать плотным сеточным функциям (сеточным функциям, хранящим все свои значения во всех узлах сетки). Запуск вычислений производится именно при таком присваивании, не раньше. До момента присваивания вычисляемого объекта плотной сеточной функции цепочка вычислений просто запоминается. Для запоминания цепочки вычислений используется механизм шаблонов выражений, описанный ранее. При присваивании происходит запуск вычислений для всех требуемых точек плотной сеточной функции в левой части оператора присваивания. Для запуска вычислений используется специальный объект – исполнитель (executor).

Он может быть указан пользователем явно или браться по умолчанию. При компиляции для CUDA (компилятором nvcc) и для обычного процессора по умолчанию подставляются разные исполнители. Для CUDA исполнитель вычисляет все точки параллельно путём вызова ядра (kernel) в графическом ускорителе. В последовательном случае исполнитель обходит все точки последовательно или параллельно с помощью OpenMP.

Применение оператора к вычисляемому объекту реализуется с помощью функционального оператора `()`. Приведём примеры. Пусть `f`, `g` – сеточные функции, `h` – сеточная функция, `A`, `B` – сеточные операторы. Тогда мы можем написать такие операторы:

```
h=A(f);
```

В этом примере оператор `A` применяется к сеточной функции `f`, при этом получается вычислитель, который присваивается плотной сеточной функции `h`.

```
h=(A+B)(f);
```

Данная запись является сокращённой формой записи `h=A(f)+B(f);` В этом примере складываются два оператора `A` и `B`, при этом получается новый составной оператор. Этот новый оператор применяется к сеточной функции `f`, при этом получается вычислитель, который присваивается плотной сеточной функции `h`.

```
h=A(f+g);
```

В этом примере складываются две сеточные функции `f` и `g`, при этом получается вычислитель (как сумма двух вычисляемых объектов, которыми являются сеточные функции). К этому вычислителю применяется оператор `A`, при этом получается ещё один вычислитель, который присваивается плотной сеточной функции `h`.

```
h=B(A(f)+g);
```

В этом примере вначале оператор `A` применяется к сеточной функции `f`, при этом создаётся вычислитель. Затем этот вычислитель складывается с сеточной функцией `g`, в результате чего создаётся новый составной вычислитель. К этому составному вычислителю применяется сеточный оператор `B`, в результате создаётся ещё один вычислитель, который присваивается плотной сеточной функции `h`. При исполнении данного оператора создаются три вычислителя.

Во всех этих примерах важно обратить внимание на следующее. Каким бы сложным ни было выражение в правой части оператора присваивания, процесс вычисления запускается один раз при присваивании. Это особенно важно при работе на графических ускорителях CUDA, когда вызов ядра является относительно дорогостоящей операцией. Независимо от сложности выражения в правой части при при-

сваивании делается один вызов одного ядра, и все вычисления производятся в этом ядре.

Главное назначение вычислителей – «запомнить» последовательность и параметры вычислений. Таким образом, в библиотеке реализуется концепция отложенных вычислений, позволяющая избавиться от промежуточных переменных и за счёт этого сэкономить оперативную память (что становится важным при больших размерах сеток).

### 9.3.2. Типы обрабатываемых данных

Величины, определённые на трехмерных сетках, это, по сути, физические поля. В физике поля могут быть скалярными (поля температур, плотностей и давлений) или векторными (поля скоростей, ускорений, сил, импульсов и т.п.). Кроме того, все величины в физике имеют размерность. При этом величины разных размерностей нельзя складывать и вычитать, а при умножении и делении получаются величины новых размерностей.

Библиотека *gridmath* поддерживает работу как со скалярными величинами, так и с векторными, причём и те, и другие могут быть как безразмерными, так и иметь размерность. Обычно при решении задач вычислительной математики работают с безразмерными типами данных, такими, как double, float или std::complex. Однако работа с типами данных, имеющими размерность, имеет свои преимущества. Во-первых, программа становится более наглядной, так как размерность той или иной переменной видна уже из её определения. Во-вторых, часть возможных программистских ошибок (таких, как сложение и вычитание величин с разными размерностями) обнаруживаются уже на стадии компиляции. Что касается эффективности вычислений (а это чрезвычайно важный аспект), то информация о размерности той или иной величины (переменной или вычисленного значения) является информацией времени компиляции (метаданными) и не приводит ни к увеличению объёма занимаемой оперативной памяти, ни к замедлению выполнения программы. Возможно, использование размерных величин несколько замедлит время компиляции программы, но несущественно.

Иногда встречаются многомерные величины, в которых различные компоненты имеют разные размерности. Например, уравнение Эйлера в гидродинамике идеальной жидкости в консервативной векторной форме записывается следующим образом:

$$\partial \mathbf{m}/\partial t + \partial \mathbf{f}_x/\partial x + \partial \mathbf{f}_y/\partial y + \partial \mathbf{f}_z/\partial z = 0,$$

где

$$\mathbf{m} = (\rho, \rho u, \rho v, \rho w, E)^T,$$

$$\mathbf{f}_x = (\rho u, p+\rho u^2, \rho uv, \rho uw, u(E+p))^T,$$

$$\mathbf{f}_y = (\rho v, \rho vu, p+\rho v^2, \rho vw, v(E+p))^T,$$

$$\mathbf{f}_z = (\rho w, \rho wu, \rho wv, p+\rho w^2, w(E+p))^T.$$

Здесь ρ – это плотность жидкости, u, v, w – компоненты скорости, p – давление, E – полная энергия единицы объёма жидкости, $E=\rho e+\frac{1}{2}\rho(u^2+v^2+w^2)$, где e – внутренняя энергия единицы массы жидкости. Величины $\mathbf{f}_x$, $\mathbf{f}_y$, $\mathbf{f}_z$ задают потоки жидкости в направлениях трёх осей.

Каждая из этих векторных величин состоит из пяти компонент, имеющих разные размерности. Уравнение Эйлера в данной форме задаёт законы сохранения массы, трёх компонент момента импульса и энергии. Библиотека *gridmath* позволяет работать и с подобными величинами.

Поддержка различных типов обрабатываемых данных в библиотеке обеспечивается естественным образом за счёт параметризации основных классов библиотеки. Все основные классы библиотеки шаблонные и принимают в качестве параметра шаблона тип обрабатываемого (или хранимого для плотных сеточных функций) значения. Таким типом может быть или любой встроенный в язык простой арифметический тип (float, double, long double), или любой класс (в том числе созданный пользователем), в котором имеются переопределённые основные арифметические операции (например, стандартный библиотечный класс std::complex). Несколько таких классов имеются и в самой библиотеке для работы с размерными или безразмерными скалярными, или векторными величинами. Прикладному программисту при необходимости не составит большого труда написать собственный класс, хранящий, например, пять чисел в каждой точке.

Скалярные и векторные величины образуют в библиотеке самостоятельные пространства объектов (сеточных функций и вычисляемых значений), но эти пространства незамкнуты. Можно написать операторы, преобразующие скалярные значения (размерные или безразмерные) в векторные или наоборот. Например, в библиотеке имеются операторы grad и quantity_grad (градиент), преобразующие скалярную величину в векторную, и операторы div и quantity_div (дивергенция), преобразующие векторную величину в скалярную.

### 9.3.3. Поддержка автоматического дифференцирования

Автоматическое дифференцирование – популярное направление в вычислительной математике (см. [17]). Основная идея автоматического дифференцирования состоит в том, чтобы в каждой точке наряду со значением функции хранить и её производную (одну или несколько по разным независимым переменным). При преобразованиях вычисляется не только новое значение функции, но и её производная, причём делается это автоматически исходя из формул при преобразовании. Пусть, например, значение новой функции g вычисляется как $g(x)=\sin(f(x)^2)$, тогда производная равна $g'(x)=\cos(f(x)^2)*2f(x)*f'(x)$. Ясно, что если в некоторой точке известно значение функции f(x) и её производной f'(x), то вычислить значение функции g(x) и её произ-

водной g’(x) очень просто. Проблема в том, как это сделать автоматически для всех точек (например, плотной сеточной функции), причём независимо от сложности формулы. Это требует «запоминания» всей цепочки вычислений по формуле и «применения» этой цепочки ко всем точкам. Но «запоминание» цепочки вычислений – это именно то, чем занимается данная библиотека.

Для решения данной задачи в библиотеке имеется класс dual, хранящий два числа: значение функции и значение её производной. Для этого класса имеются переопределённые операторы для всех основных арифметических операций, правильно вычисляющие значение как операции, так и её производной. Например, произведение определено следующим образом:

```
template<typename T>
dual<T> operator*(const dual<T>& a, const dual<T>& b){
  return dual<T>(a.value() * b.value(),
    a.derivative() * b.value() + a.value() * b.derivative());
}
```

Имеются также переопределённые основные функции из стандартной математической библиотеки. Например, синус определён так:

```
template<typename T>
dual<T> sin(dual<T> a){
  return dual<T>(sin(a.value()), a.derivative() *
cos(a.value()));
}
```

Теперь достаточно подставить этот класс в качестве параметра шаблона плотной сеточной функции. Вот пример исходного кода:

```
dense_grid_function<dual<double> > f(100, 100, 100);
fill(f); // заполняем функцию f (вместе с производной)
// создаём новую сеточную функцию того же размера.
dense_grid_function<dual<double> > g = f.clone();
g = sin(f*f);
```

Функция g автоматически заполнится правильными значениями как самой функции, так и её производной. По поводу последнего оператора в примере нужно сделать одно замечание: в библиотеке переопределены основные математические функции из стандартной библиотеки, принимающие в качестве параметра вычисляемый объект и возвращающие сеточный вычислитель, вызывающий требуемую функцию при вычислении значений в точках.

В библиотеке также имеется класс dual3, хранящий частные производные функции по всем трём осям (класс хранит 4 значения). Для него также переопределены основные математические операции и основные математические функции из стандартной библиотеки. Использование данного класса в библиотеке аналогично использованию класса dual.

Далее рассмотрим более подробно основные классы и понятия библиотеки.

### 9.3.4. Сеточная функция

Под сеточной функцией понимается объект, принимающий определённые значения (одно или несколько) в каждом узле некоторой сетки. В качестве параметра принимается индекс узла сетки. Возможны разные реализации (классы) сеточных функций. В библиотеке реализованы три таких класса. По аналогии с понятиями плотной и разреженной матриц введём понятия плотной и разреженной сеточных функций. Плотная сеточная функция (класс *dense_grid_function*) принимает, вообще говоря, различные значения в различных узлах сетки и хранит все значения в оперативной памяти. Константная сеточная функция (*constant_grid_function*) принимает во всех узлах сетки одно и то же заданное значение. Вычисляемая сеточная функция (*computable_grid_function*) не хранит свои значения, а каждый раз их вычисляет на основе заданного функционального объекта. Размер, занимаемый в оперативной памяти константной и вычисляемой сеточными функциями, пренебрежимо мал. Для оценки размера требуемой оперативной памяти достаточно учесть только плотные сеточные функции. Следует обратить внимание на то, что при компиляции для CUDA плотная сеточная функция хранит свои данные в памяти графического процессора, поэтому то, какого размера сетку удастся разместить в памяти, определяется не объёмом оперативной памяти основного процессора (каким бы большим он ни был), а объёмом оперативной памяти графического процессора, который может быть существенно меньше. Например, на суперкомпьютере К-100 объём оперативной памяти основного (host) процессора – 96 Гбайт на узел, а графического ускорителя – всего 2,8 Гбайт на одну графическую плату.

Сеточная функция является вычисляемым объектом (наряду с сеточным вычислителем, см. далее). Это означает, что к сеточным функциям можно применять сеточные операторы, и для сеточных функций определены арифметические операции с другими вычисляемыми объектами и со скалярными величинами.

Для плотной сеточной функции реализован оператор присваивания, принимающий в качестве параметра сеточный вычислитель. Именно этот оператор запускает отложенные вычисления. Порядок вычисления значений в узлах сетки для оператора присваивания не определён и, вообще говоря, может осуществляться параллельно для разных узлов сетки. В связи с этим может возникнуть вопрос: может ли в левой части оператора присваивания стоять плотная сеточная функция, участвующая в вычислениях в правой части? Ответ неоднозначный. Это, как правило, можно делать в том случае, если для вычисления значения в узле сетки не используются значения из других узлов той же сеточной функции.

Пользователь библиотеки для своих нужд может реализовать собственные сеточные функции. Например, можно реализовать разреженную сеточную функцию, хранящую значения только для определённых комбинаций координат узлов, а для остальных возвращающую ноль.

### 9.3.5. Вычисляемый объект

Вычисляемый объект (evaluable object) – это объект, к которому может быть применён сеточный оператор. К вычисляемым объектам относятся сеточные функции и сеточные вычислители.

Для вычисляемых объектов определены операции сложения и вычитания с другими вычисляемыми объектами и все четыре арифметических операции со скалярными величинами, причём скалярная величина может быть как в левой, так и в правой части арифметической операции. Результатом арифметической операции (с другим вычисляемым объектом или со скалярной величиной) является сеточный вычислитель, который, в свою очередь, также является вычисляемым объектом.

### 9.3.6. Сеточный оператор

Сеточный оператор (grid operator) – это объект, главное назначение которого – «применение» к вычисляемому объекту (сеточной функции или сеточному вычислителю). Результатом этого применения является сеточный вычислитель (grid evaluator).

### 9.3.7. Сеточный вычислитель

Сеточный вычислитель (grid evaluator) создаётся автоматически при применении сеточного оператора к вычисляемому объекту (сеточной функции или другому вычислителю) или как результат арифметической операции между вычисляемыми объектами (сеточными функциями и вычислителями), и его можно присвоить плотной сеточной функции. Как правило, вычислитель возникает как промежуточный результат вычислений и явно нигде не встречается.

### 9.3.8. Исполнители

Как уже говорилось выше, исполнители (executor) используются для запуска вычислений при присваивании вычисляемого объекта плотной сеточной функции. При этом исполнитель можно указать явно или воспользоваться исполнителем по умолчанию. Именно исполнитель определяет, в какой последовательности будут производиться вычисления в разных точках сеточной функции. В библиотеке уже имеется несколько исполнителей, кроме того, пользователь имеет возможность написать свои. Два из уже имеющихся в библиотеке исполнителя используются по умолчанию, т.е. в том случае, если исполнитель не указан явно. Первый – для CUDA, осуществляющий параллельный запуск вычислений для всех точек сетки путём вызова ядра.

Второй – для последовательного случая, обходящий точки в одном или нескольких вложенных циклах (в зависимости от версии библиотеки). Хотя этот исполнитель и называется последовательным, он также может распараллелить обход точек за счёт использования технологии OpenMP. Перед циклами стоит прагма

```
#pragma omp parallel for
```

и если при компиляции включить опцию -openmp, то исполнение циклов будет распараллелено.

Кроме того, есть несколько исполнителей, которые пользователь может указать явно. Например, есть исполнители, обрабатывающие точки в соответствие с определённой последовательностью, что позволяет решать с помощью библиотеки задачи по неявной схеме типа Гаусса-Зейделя, когда часть точек берётся с предыдущего временного слоя, а часть с текущего. При этом все вычисляемые точки делятся на несколько групп и эти группы обрабатываются последовательно. Для точек в самой группе вычисления могут производиться параллельно. При использовании таких исполнителей, как правило, вычисления можно производить «на месте», т.е. в правой части оператора присваивания использовать ту же сеточную функцию, что и в левой части, так как при вычислениях в точке никогда не используются точки из той же самой группы.

Библиотека поддерживает две подобных схемы вычислений. Первая – это шахматная раскраска, при которой все точки делятся на две группы – «чёрные» с чётной суммой индексов и «белые» с нечётной. В начале вычисления производятся параллельно во всех «чёрных» точках, а затем, также параллельно – во всех «белых».

Вторая схема – это схема, при которой, например, значения из точек с меньшими индексами берутся с текущего временного слоя, а с большими – с предыдущего. Подобные схемы решаются методом гиперплоскости фронта вычислений.

### 9.3.9. Индексаторы

При работе на графических ускорителях чрезвычайно важным для производительности является то, что CUDA-потоки с последовательными номерами должны обращаться к последовательным ячейкам глобальной памяти. Если это условие не выполняется, то производительность может упасть в несколько раз (от 3 по 5). В случае регулярных сеток это требование обычно выполняется. Данные сеточной функции располагаются в памяти в начале по оси x, затем по оси y и в конце по оси z. Исполнитель по умолчанию для CUDA осуществляет трёхмерный запуск ядра, при этом соседние потоки имеют соседние индексы по оси x.

Ситуация меняется, когда используется нерегулярные сетки или используется гиперплоскость фронта вычислений. При стандартном расположении данных плотной сеточной функции ни при каком по-

рядке обхода точек гиперплоскости на очередном шаге движения этой гиперплоскости соседние потоки не будут обращаться к соседним ячейкам памяти. Выход был подсказан разработчиками системы DVM (см. [31]), столкнувшимися при реализации своей системы для CUDA с той же проблемой. В системе DVM данные в таком случае переупорядочиваются так, чтобы в соседних ячейках памяти лежали данные из ячеек трёхмерного массива, расположенных на одной горизонтальной диагонали (в горизонтальной плоскости xy). Для параллельного обхода точек гиперплоскости применяется алгоритм, при котором на очередном шаге соседние потоки обрабатывают соседние точки, лежащие на одном горизонтальном отрезке, принадлежащем гиперплоскости, и расположенном на диагонали в плоскости xy.

В данной библиотеке нет необходимости явно переупорядочивать данные сеточной функции. Вместо этого был применён другой подход – в библиотеку были введены новые объекты – индексаторы. Индексаторы в библиотеке не являются самостоятельными объектами, а служат для параметризации плотных сеточных функций. В библиотеке сеточная функция является не классом языка C++, а шаблоном класса. Параметрами шаблона являются тип хранимых данных и класс индексатора. При этом класс индексатора имеет значение по умолчанию, т.е. его можно не указывать. Прототип шаблона класса плотной сеточной функции следующий:

```
template<typename T, class I = default_grid_index>
struct dense_grid_function;
```

В классе индексатора должен быть определён функциональный оператор, принимающий три индекса (i, j, k) и возвращающий линейный индекс данной ячейки. Размеры сеточной функции, которые могут понадобиться индексатору, могут быть указаны или в конструкторе индексатора, или в методе resize:

```
void resize(size_t size_x, size_t size_y, size_t size_z);
```

Индексатор по умолчанию (класс default_grid_index) вычисляет линейный индекс для трёхмерной регулярной сетки следующим образом:

```
size_t operator()(size_t i, size_t j, size_t k) const {
     return (k * m_size_y + j) * m_size_x + i;
}
```

Таким образом, ячейки плотной сеточной функции, у которых индексы j и k совпадают, а индекс i отличается на единицу, будут иметь соседние линейные индексы.

Для работы с гиперплоскостью фронта вычислений в библиотеке имеется индексатор (класс diagonal_grid_index), в котором соседние линейные индексы имеют ячейки сеточной функции, лежащие в одной горизонтальной плоскости (с равными индексами k) на одной го-

ризонтальной диагонали (с равными суммами индексов i+j) и у которых индексы i отличаются на единицу (чем больше i, тем больше линейный индекс).

В библиотеке имеется также «правильный» исполнитель, в котором соседние потоки обращаются к соседним ячейкам сеточной функции на горизонтальной диагонали. При использовании данного исполнителя замена индексатора на диагональный даёт на графических ускорителях ускорение примерно в 4 раза по сравнению с индексатором по умолчанию.

Эти «диагональные» исполнитель и индексатор работают только с гиперплоскостью, проходящей под углом 45° ко всем осям и движущейся от начала координат, т.е. с гиперплоскостью с параметрами (1, 1, 1). Гиперплоскость с такими параметрами встречается чаще всего. Если параметры гиперплоскости оказались другими, то можно использовать более общие универсальные исполнители для гиперплоскости (см, например, [19]) и индексатор по умолчанию. В этом случае всё будет работать правильно, но не столь эффективно.

### 9.3.10. Общий вид запуска вычислений

Как уже говорилось выше, вычислитель можно присвоить плотной сеточной функции, при этом будут запущены вычисления для всех точек сеточной функции в левой части оператора присваивания. В этом (простейшем) случае вычисления производятся во всех точках плотной сеточной функции. Однако, бывают случай, когда вычисления нужно провести не во всех точках. Например, оператор Лапласа не определён в граничных точках. Кроме того, в простейшем случае всегда подставляется исполнитель по умолчанию.

Общий вид операции запуска вычислений для трёхмерных регулярных сеток следующий:

```
grid_assign(assignment1, assignment2, …).exec(x0, xn, y0, yn,
z0, zn, executor);
```

Для нерегулярных сеток операция запуска имеет следующий вид:

```
grid_assign(assignment1, assignment2, …).exec(i0, in, executor);
```

Здесь assignment1, assignment2 – это т.н. «присваиватели», выражения вида «func <<= expr», где func – это плотная сеточная функция, а expr – произвольный вычисляемый объект. Данное выражение очень похоже на оператор присваивания и отличается только тем, что вместо оператора «=» используется оператор «<<=». Это выражение, в отличие от оператора присваивания, не запускает вычисления, а создаёт объект (присваиватель), который запоминает, какой сеточной функции и что нужно присвоить. Функция grid_assign принимает в качестве параметров от 1 до 10 присваивателей. Если 10 присваивателей мало, то перед включением заголовочных файлов библиотеки

можно задать константу MAX_ASSIGNENT_SIZE равной нужному максимальному числу присваивателей.

Смысл задания нескольких присваивателей в том, чтобы осуществить все эти вычисления в едином цикле (или, в случае CUDA, в одном вызове ядра). Функция grid_assign возвращает объект класса grid_package, в котором запоминаются все присваиватели. Затем в этом объекте вызывается метод exec, который и осуществляет запуск вычислений. В качестве параметров этому методу передаются границы области, в которой нужно произвести вычисления и исполнитель. Исполнитель можно не указывать, в таком случае подставится исполнитель по умолчанию. Использовать эту более сложную форму запуска вычислений нужно в одном из трёх случаев (если выполнено хотя бы одно из трёх условий):

- Нужно выполнить несколько присваиваний в одном вызове;
- Вычисления нужно провести не на всей области, а на её части (например, только во внутренних точках);
- Нужно использовать исполнитель, отличный от исполнителя по умолчанию.

Оператор присваивания на самом деле реализован через вызов функции grid_assign следующим образом (для трёхмерных регулярных сеток):

```
template<typename T, class I = default_grid_index>
struct dense_grid_function {
  …
  template<class EO>
  void operator=(const grid_evaluable_object<
    EO, typename EO::proxy_type> &eobj){
    grid_assign(*this <<= eobj).exec(
      0, size_x(), 0, size_y(), 0, size_z());
  }
};
```

Функции grid_assign передаётся единственный присваиватель (присваивающий данной сеточной функции указанный вычисляемый объект), а в качестве границ вычисляемой области в метод exec передаются полные границы сеточной функции. Исполнитель не указывается (используется исполнитель по умолчанию).

Приведём пример явного использования функции grid_assign из программы, реализующей метод верхней релаксации (SOR). В ней на каждом шаге итерации нужно вычислить невязку (разницу между старым и новым значениями). Основная сеточная функция – A, невязка хранится в сеточной функции S. Основной оператор итерации следующий:

```
hyperplane_executor executor;
grid_assign(
     S <<= A,              // Запоминаем старое значение
```

```
      A <<= sor(A),          // Вычисляем новое значение
      S <<= abs(A - S)       // Вычисляем невязку
).exec(1, size_x - 1, 1, size_y - 1, 1, size_z - 1, executor);
```

В данном примере функции grid_assign передаются три присваивателя, вычисления проводятся только во внутренних точках и используется исполнитель, обходящий область вычислений по гиперплоскостям.

#### 9.3.11. Примеры

Приведём примеры исходного кода, который может встречаться в программе, использующей данную библиотеку. Пусть `f`, `g` – сеточные функции, `h` – плотная сеточная функция, `A`, `B` – сеточные операторы. Тогда можно написать такой код:

```
h=A(f)+B(g); // сложение двух вычислителей.
h=f+B(g);  // сложение сеточной функции f и вычислителя B(g).
h=2.0*(A+B)(f); // сложение операторов A и B и умножение скаляра
2.0 на вычислитель (A+B)(f).
h=2.0*(3+A(f)); // сложение скаляра 3 и вычислителя A(f) и умно-
жение скаляра 2.0 на вычислитель (3+A(f)).
h=2.0*(f-B(g)); // вычитание вычислителя B(g) из сеточной функ-
ции f и умножение скаляра 2.0 на вычислитель (f-B(g)).
h=3.0+(A(B(f)); // применение оператора A к вычислителю B(f) и
сложение скаляра 3.0 и вычислителя (A(B(f)).
h=B(f-A(g));    // вычитание вычислителя A(g) из сеточной функ-
ции f и применение оператора B к полученному вычислителю (f-
A(g)).
```

### 9.4. Принципы реализации

#### 9.4.1. Общая информация

Сеточно-операторный подход к программированию реализован в виде библиотеки шаблонов gridmath. Большинство её классов и функций шаблонные. Таким образом, вся библиотека поставляется в исходных текстах в виде набора .h и .hpp файлов. Скомпилировать исходные тексты, использующие данную библиотеку, можно большинством современных компиляторов с языка C++, включая Microsoft C++, Intel C++, gcc и nvcc.

Библиотека gridmath, помимо стандартной библиотеки языка C++ (STL) использует библиотеку boost (см. [10]), а при компиляции для CUDA – ещё и библиотеку thrust из состава CUDA SDK (см. [11, 12]). Библиотека thrust содержит аналоги большинства необходимых компонентов как из STL, так и из boost.

#### 9.4.2. Шаблоны выражений

Сеточно-операторный подход к программированию основан на шаблонах выражений (см. раздел 6.2). При этом базовым классом для

всех выражений служит вычисляемый объект (evaluable_object). От него наследуются сеточные функции и сеточные вычислители. Сеточные вычислители получаются или как результат арифметических операций между вычисляемыми объектами (выражениями), или как результат применения сеточного оператора к вычисляемому объекту. Таким образом, любой вычисляемый объект хранит в себе цепочку вычислений, в корне которых лежат сеточные функции, хранящие исходные данные. Вычисляемый объект может быть присвоен плотной сеточной функции, в этот момент запускаются вычисления в соответствии с запомненной в вычисляемом объекте цепочкой вычислений для всех ячеек сетки, на которой определена плотная сеточная функция.

### 9.4.3. Размерные величины

Библиотека gridmath поддерживает работу с размерными величинами (см. главу 7). Для определения плотной сеточной функции, хранящей, например, давление, нужно передать в качестве параметра шаблона плотной сеточной функции вместо типа данных double передать размерный тип данных. Вместо

```
dense_griid_function<double> p;
```

нужно написать:

```
typedef quantity_dim<1,-1,-2,0,0,0,0> pressure;
dense_grid_function<quantity<double, pressure> > p;
```

### 9.4.4. Объекты-заместители

В библиотеке реализована концепция отложенных вычислений: до тех пор, пока не будет исполнен оператор присваивания вычисляемого объекта плотной сеточной функции, вычисления не производятся. Вместо этого последовательность вычислений запоминается. При этом запоминаются сеточные функции, операторы, вычислители и операции с ними и со скалярами. Но что значит «запоминаются»? Ссылки на объекты сохранять нельзя, также, как нельзя сохранять копии объектов. Ссылки на объекты нельзя сохранять по двум причинам. Первая – это то, что объекты (операторы и вычислители) часто создаются «на лету» как результат арифметических операций. Ссылки на такие объекты (созданные «на лету») сохранять нельзя, так как временный объект сразу после использования может быть удалён и затёрт. Вторая причина состоит в том, что при компиляции для CUDA объекты создаются в памяти главного (host) процессора, а исполняются в памяти графического вычислителя CUDA. Чтобы вычисления можно было выполнить, нужно вычисляемый объект скопировать из памяти host-процессора в память графического вычислителя, копировать же ссылки (или указатели) на объекты из памяти host-процессора в память CUDA не имеет смысла, т.к. это разные адресные простран-

ства. Фактически в случае плотной сеточной функции нужно скопировать указатель на данные.

Для того чтобы решить указанную проблему, в библиотеке вводится понятие «заместителя» объекта (proxy). В качестве замещаемого объекта может быть практически любой объект библиотеки: сеточная функция, оператор или вычислитель. Тот или иной объект может быть «лёгким» или «тяжёлым» для копирования. «Лёгкими» для копирования будем называть объекты, не содержащие векторы данных, а «тяжёлыми» – содержащие их. Если объект «лёгкий», то при копировании будет сохраняться копия самого объекта, а если «тяжёлый» – то его «лёгкий» заместитель. Заместитель объекта похож на сам объект за тем исключением, что в заместителе объекта все вектора данных заменяются на указатели на эти данные. Для того, чтобы поддержать объекты заместители, самый базовый класс изменяется, теперь он принимает следующий вид:

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

  _Proxy get_proxy(){ return self(); }
  _Proxy get_proxy() const { return self(); }
};
```

В класс math_object_base добавляется дополнительный шаблонный параметр _Proxy, по умолчанию совпадающий с именем конечного класса, и два метода get_proxy. По умолчанию все объекты (сеточные функции, операторы и вычислители) считаются «лёгкими», т.е. в качестве класса-заместителя по умолчанию передаётся сам класс объекта, а в качестве объекта-заместителя создаётся копия самого объекта. Если сеточный оператор или сеточная функция, написанные программистом, является «тяжёлым» (содержит один или несколько векторов данных), то следует создать класс заместителя и в базовый класс передать в качестве второго шаблонного параметра этот класс заместителя. В качестве примера при реализации класса заместителя можно взять класс плотной сеточной функции dense_grid_function. Схематично это можно показать на следующем примере:

```
template<typename T>
struct dense_grid_function_proxy; // Предварительное объявление
temlate<typename T>
struct dense_grid_function :
  math_object<dense_grid_function<T>,
dense_grid_function_proxy<T> >,
```

```
  std::vector<T>
{
  dense_grid_function(size_t size_x, size_t size_y, size_t
size_z) :
    m_size_x(size_x), m_size_y(size_y), m_size_z(size_z){}
  size_t size_x() const { return m_size_x; }
  size_t size_y() const { return m_size_y; }
  size_t size_z() const { return m_size_z; }
     … // Другие методы
private:
     size_t m_size_x, m_size_y, m_size_z;
};
temlate<typename T>
struct dense_grid_function_proxy
{
  typedef T value_type;
  dense_grid_function_proxy(dense_grid_function<T>& func) :
    m_size_x(func.size_x()), m_size_y(func.size_y()),
    m_size_z(func.size_z()), m_data(func.data_pointer()){}
  size_t size_x() const { return m_size_x; }
  size_t size_y() const { return m_size_y; }
  size_t size_z() const { return m_size_z; }
  value_type operator()(size_t i, size_t j, size_t k) const {
          return m_data[(k * m_size_y + j) * m_size_x + i];
}
private:
     size_t m_size_x, m_size_y, m_size_z;
     value_type *m_data; // Указатель на данные вместо массива
};
```

Если нужно сохранить сеточную функцию, оператор или вычислитель для последующих отложенных вычислений, то библиотека всегда сохраняет заместитель объекта, а не сам объект путём вызова в сохраняемом объекте метода get_proxy().

### 9.4.5. Контекст исполнения

В «обычной» программе (без использования данной библиотеки) при одновременном вычислении нескольких величин для ускорения счёта для каждой точки могут однократно вычисляться дополнительные вспомогательные переменные, которые затем многократно используются. Поясним это на примере. Пусть одновременно вычисляются две величины (плотных сеточных функции), a=sin(c*c) и b=cos(c*c), где c – некая третья сеточная функция. Тогда цикл в «обычной» программе мог бы выглядеть так:

```
for(size_t i = 0; i < n1; ++i){
  for(size_t j = 0; j < n2; ++j){
     for(size_t k = 0; k < n3; ++k){
       double temp = sqr(c(i, j, k));
       a(i, j, k) = sin(temp);
```

```
        b(i, j, k) = cos(temp);
      }
  }
}
```

Ясно, что мы экономим вычисления за счёт того, что значение квадрата величины c в каждой точке вычисляется однократно. Если мы просто перепишем эту программу, то получим примерно такой код:

```
grid_assign(
  a <<= sin(sqr(c)),
  b <<= cos(sqr(c))
).exec(0, n1, 0, n2, 0, n3);
```

Этот код существенно лучше предыдущего тем, что он может быть исполнен параллельно для разных точек (например, при использовании CUDA или OpenMP), но, тем не менее, имеет тот недостаток, что одна и та же величина в нём вычисляется дважды. Для решения этой проблемы в библиотеке введено понятие контекста исполнения. Контекст исполнения – это структура, определяемая пользователем (программистом), которая создаётся для каждого набора индексов (каждой точки), и члены которой могут затем многократно использоваться. Для создания контекста используется вспомогательный объект – построитель контекста (context builder). Покажем на примере, как это работает.

```
struct my_context; // forward declaration
// Класс построителя контекста
struct my_context_builder :
grid_context_builder<my_context_builder>{
  // В типе context_type содержится класс контекста
  typedef my_context context_type;
  // В конструктор построителя контекста передаются
  // всё, что нужно для вычисления переменных,
  // в данном случае передаётся сеточная функция c.
  my_context_builder(const dense_grid_function<double>& c) :
    c_proxy(c.get_proxy())

  // Заместитель объекта (плотной сеточной функции)
  const typename dense_grid_function<double>::proxy_type
c_proxy;
};
// Класс контекста
struct my_context : grid_context<my_context>{
  // В конструктор контекста передаются координаты точки
  // и построитель контекста. На основании этой информации
  // вычисляются все вспомогательные переменные
  my_context(size_t i, size_t j, size_t k,
    const my_context_builder& cb)
  {
```

```
    double temp = cb.c_proxy(i, j, k);
    sqrc = temp * temp;
  }
  double sqrc;
};
// Ещё один класс для извлечения вспомогательной переменной
struct context_sqrc : grid_evaluable_object<context_sqrc>{
  double operator()(size_t i, size_t j, size_t k,
    const grid_context<my_context>& context) const
  {
    return context.self().sqrc;
  }
} _sqrc;   // И переменная этого класса
// Оператор присваивания
grid_assign(
     a <<= sin(_sqrc),
     b <<= cos(_sqrc)
).exec(0, n1, 0, n2, 0, n3, my_context_builder(c));
```

### 9.4.6. Исполнение на CUDA

Компилятор nvcc полноценно поддерживает работу с шаблонами функций и классов. В частности, глобальные функции (помеченные ключевым словом __global__), которые исполняются на графическом устройстве, а вызываются из главного (host) процессора, также могут быть шаблонными. Исходя из этого факта возникло решение: для любой задачи, которую нужно исполнить на устройстве, создать объект, содержащий в себе все необходимые данные для решения конкретной задачи и имеющий очень простой внешний интерфейс (например, функциональный метод, в который передаются глобальные индексы нитей по всем трём направлениям. Решение было подсмотрено в библиотеке thrust. Там такой объект называется «замыканием» (closure). Итак, нужно написать одну глобальную шаблонную функцию с одним параметром, в которую передать в качестве параметра шаблона название класса замыкания. Параметром функции будет замыкание. Эта глобальная функция вычисляет глобальные индексы нитей и вызывает функциональный метод в замыкании, передав туда в качестве параметров эти глобальные индексы. Всю основную работу будет исполнять этот метод в замыкании. Исходный текст этой глобальной функции следующий:

```
template<class Closure>
__global__
void launch_3d(Closure closure)
{
  closure(
    blockIdx.x * blockDim.x + threadIdx.x,
    blockIdx.y * blockDim.y + threadIdx.y,
    blockIdx.z * blockDim.z + threadIdx.z);
}
```

```
#define BLOCK1_SIZE   512
template<class Closure>
void launch_closure_3d(Closure& closure,
  size_t size_x, size_t size_y, size_t size_z)
{
  size_t block_size_x = BLOCK1_SIZE;
  while(block_size_x / 2 >= size_x && block_size_x > 8)
    block_size_x /= 2;
  size_t block_size_y = BLOCK1_SIZE / block_size_x;
  while(block_size_y / 2 >= size_y)
    block_size_y /= 2;
  size_t block_size_z = BLOCK1_SIZE / block_size_x /
block_size_y;
  while(block_size_z / 2 >= size_z)
    block_size_z /= 2;
  dim3 dimGrid(
    (size_x + block_size_x - 1) / block_size_x,
    (size_y + block_size_y - 1) / block_size_y,
    (size_z + block_size_z - 1) / block_size_z);
  dim3 dimBlock(block_size_x, block_size_y, block_size_z);
  launch_3d<<<dimGrid, dimBlock>>>(closure);
}
```

На класс замыкания накладываются определённые требования (концепция замыкания). Так как объект этого класса передаётся по значению (копируется) из основного процессора на графическую плату, он должен содержать только переменные или простых типов (в том числе указателей), или классов, имеющих копирующий конструктор по умолчанию. Если в классе есть указатели, то это должны быть указатели на память в CUDA. В нём также должен быть функциональный метод с тремя параметрами – глобальными индексами нитей.

### 9.4.7. Обмен данными между основным процессором и графическим ускорителем

При использовании графического ускорителя очень важно иметь возможность пересылать данные из памяти основного процессора в память графического устройства и наоборот. Это может быть нужно, например, для загрузки исходных данных из файла и сохранения результатов счёта в файл, а также в процессе счёта для обмена данными между процессами по MPI. В библиотеке данные хранятся в векторах. Основной класс для хранения данных – это класс math_vector. Этот класс хранит данные в том устройстве, на котором ведётся счёт. Если это последовательная версия библиотеки, то данные хранятся в памяти основного процессора, а если это CUDA, то в памяти графического ускорителя. Есть также класс host_vector, который хранит данные всегда в памяти основного процессора. Оба класса находятся в пространстве имён kiam::math. Переменные этих двух классов можно присваи-

вать друг другу, при этом данные будут копироваться из памяти основного процессора в память графического ускорителя или наоборот.

Что касается реализации, то в последовательном варианте оба класса пронаследованы от класса std::vector из стандартной библиотеки языка C++ (то есть фактически math_vector и host_vector – это одно и то же). В случае CUDA класс math_vector пронаследован от класса thrust::device_vector, а класс host_vector – от класса thrust::host_vector. Копирование данных фактически осуществляется библиотекой thrust.

В частности, плотная сеточная функция хранит данные в переменной типа math_vector. В переменную того же типа сохраняется срез плотной сеточной функции, которую затем можно скопировать в переменную типа host_vector и затем передать данные по MPI.

### 9.4.8. Использование пула нитей.

На современных компьютерах, как серверных, так и обычных рабочих станциях, стоящих у нас на рабочих столах или дома, часто стоят современные многоядерные процессоры, способные запускать по несколько нитей (от 2 до 8 и более на процессор) параллельно. А на серверах ещё и процессоров может быть несколько на одном узле. Если использовать полностью возможности таких процессов, то программа может считаться существенно быстрее.

Есть разные способы распараллеливания работы программы, например, если есть компилятор, поддерживающий OpenMP, то можно в последовательной программе перед циклом поставить соответствующую прагму и всё заработает. Кстати, в последовательной версии библиотеки эта прагма перед тремя вложенными циклами стоит. Это значит, что если указать опцию компилятору –openmp, то программа автоматически распараллелится. Именно так можно запустить параллельную программу на процессоре Intel Xeon Phi в «native» режиме.

Однако, не все компиляторы поддерживают эту опцию. Чтобы и в этом случае можно было распараллелить работу последовательной программы, библиотека использует возможности распараллеливания, предоставляемые библиотекой boost. Используется библиотека boost-threads, которая должна быть скомпилирована.

Для использования пула нитей предназначен специальный исполнитель – grid_threadpool_executor. В качестве параметра конструктора этого класса передаётся количество нитей. Чтобы использовать все имеющиеся в системе процессоры, нужно передать результат функции boost::thread::hardware_concurrency(). Этот исполнитель нужно передать в метод exec() объекта grid_package, возвращаемого функцией grid_assign.

## 9.5. Заключение

Сеточно-операторный подход объяснялся на примере регулярных трёхмерных сеток. Фактически сеточно-операторная библиотека, реализующая данный подход, имеет несколько реализаций, как на регулярных, так и на нерегулярных двух- и трёхмерных сетках. Отличие библиотеки для нерегулярных сеток в основном заключается в том, что все ячейки сетки пронумерованы и каждая имеет свой уникальный номер. Таким образом, для обхода сетки, в отличие от регулярных трёхмерных сеток, используются не три числа (индекса), а одно.

С помощью сеточно-операторного подхода был решён ряд задач, в том числе:

- Параллельный многосеточный метод для разностных эллиптических уравнений на трёхмерных регулярных сетках (см. [20, 44]);
- Численное решение параболических уравнений на локально-адаптивных сетках чебышевским методом (см. [45]);
- Тест MG из набора тестов NAS Parallel Benchmarks (см. [46]);
- Разрывный метод Галёркина на трёхмерных тетраэдральных нерегулярных сетках (см. [47-52]).
- Задача о решении системы квазигидродинамических уравнений (см. [53-55]);
- Задача теплопроводности на регулярных трёхмерных сетках. Сравнивались разные методы решения задачи, в том числе явный метод Якоби, неявный метод Гаусса-Зейделя (метод шахматной раскраски и метод гиперплоскости фронта вычислений) в двух вариантах – простом и последовательной верхней релаксации (successive over relaxation - SOR). Кроме того, для сравнения были реализованы двухслойный метод простой итерации, двухслойный и трёхслойный Чебышевские методы и многосеточный метод. Все методы были реализованы в последовательном и параллельном (для CUDA) вариантах (см. [56, 57]).
- Метод ENO для одномерного случая (см. [58]).

Все перечисленные задачи были решены как на обычных CPU (с использованием MPI и OpenMP), так и на графических ускорителях CUDA. Применение сеточно-операторного подходя для решения этих задач позволило ускорить написание и отладку программ, а также без проблем быстро перенести их на CUDA. Что касается скорости работы программ, написанных с использованием сеточно-операторного подхода, то тут результат зависит от задачи. Модельная задача (Тест MG из NAS Parallel Benchmarks) замедлилась в 1.5-2 раза (в зависимости от размера задачи) по сравнению с исходной реализацией на Фортране, что представляется допустимым, т.к. трудно догнать специально написанную программу, что же касается реальных задач, то в

том случае, если была альтернативная реализация, ускорение при использовании сеточно-операторного подхода было существенным, иногда даже в разы. Кроме того, перенос на CUDA (а он, как уже говорилось, делается практически простой перекомпиляцией исходного текста) давал дополнительное ускорение от 4 до 8 раз.

## 10. Заключение

В данной работе автор пытается показать, что язык программирования C++ может широко применяться для численного решения задач математической физики, причём эффективно использоваться могут многие отличительные свойства языка, включая объектно-ориентированное программирование и метапрограммирование шаблонов. Опыт программистов, пытавшихся использовать язык C++ для решения численных задач и получавших замедление по сравнению с программами на «чистом» C, говорит лишь о том, что использование языка C++, возможно, было не совсем правильным. Разработчики языка C++ одной из главных своих целей всегда ставили эффективность, и язык C++ – действительно эффективный язык. Причём это относится не только к скорости исполнения пользовательского кода, но и к стандартной библиотеке. В частности, со стандартными коллекциями (например, с вектором std::vector) лучше всего работать с помощью функций из стандартной библиотеки, позволяющих копировать данные (std::copy), переносить с преобразованием (std::transform), просматривать (std::for_each), накапливать результат (std::accumulate) и т.д. Эти функции работают зачастую гораздо эффективнее, чем «вручную» написанные циклы. Применение этих стандартных функций стало ещё удобнее в стандарте языка C++11, в котором появились лямбда-выражения, которые можно передавать в стандартные функции и которые позволяют писать обработчики «на месте», а не в отдельных классах или функциях за пределами текущей исполняемой функции.

Особое место в языке занимают шаблоны, появившиеся в языке далеко не сразу, а только через полтора десятилетия после появления самого языка. Шаблоны оказались удобными не только для реализации коллекций и алгоритмов, но и позволили использовать метапрограммирование шаблонов, что позволяет вынести часть вычислений на стадию компиляции, и, применительно к задачам математического моделирования, позволяет писать ещё более эффективные программы. Программа, написанная с активным использованием метапрограммирования шаблонов, может работать существенно быстрее аналогичной программы, написанной на языке C.

Существует множество книг и учебников по языку C++, в том числе написанных автором языка Бьерном Страуструпом (см, например, [1-6]). Книг, посвящённых именно метапрограммированию шаб-

лонов, гораздо меньше. Очень хорошая и полезная книга Дэвида Абрахамса и Алексея Гуртового (см. [7]). Эта книга довольно старая (2004 г.), но купить её можно (например, на сайте amazon.com), можно и попытаться найти в электронном виде в интернете. Данная работа пытается в какой-то степени восполнить этот пробел применительно к численным задачам математической физики.

## 11. Список литературы


1. Bjarne Stroustrup. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 978-0-321-56384-2, 1368 c.
2. Bjarne Stroustrup. *Programming: Principles and Practice Using C++, Second Edition*. Addison-Wesley, 2013, 1312 c. ISBN 978-0-321-99278-9
3. Bjarne Stroustrup. *A Tour of C++*. Addison-Wesley, 2014, 192 c. ISBN 978-0-321-95831-0.
4. Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994, 480 c. ISBN 978-0-201-54330-8.
5. Бьерн Страуструп. *Программирование: Принципы и практика с использованием C++, второе издание*. пер. с англ., Вильямс, 2016, 1328 с. ISBN 978-5-8459-1949-6, 978-0-321-99278-9.
6. Бьерн Страуструп. *Дизайн и эволюция языка C++*. ДМК Пресс, 2016, 446 с. ISBN 978-5-97060-419-9, 978-0-201-54330-8.
7. David Abrahams, Aleksey Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley, 2004, 400 c. ISBN 978-0-321-22725-6.
8. *Template metaprogramming*. URL: http://en.wikipedia.org/wiki/Template_metaprogramming
9. T. Veldhuizen, *Expression Templates*. C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31.
10. *Boost C++* Libraries. URL: http://www.boost.org/
11. *NVidia CUDA*. URL: http://www.nvidia.com/object/cuda_home_new.html
12. *Thrust – Parallel Algorithms Library*. URL: http://thrust.github.com/
13. *OpenACC Toolkit*. URL: https://developer.nvidia.com/openacc
14. *OpenMP*. URL: http://openmp.org
15. *Substitution failure is not an error* (SFINAE). URL: https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error
16. *Curiously recurring template pattern* (CRTP). URL: http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
17. Automatic Differentiation. URL: http://www.autodiff.org/

18. Liu J. *Thermodynamically Consistent Modeling and Simulation of Multiphase Flows*, DISSERTATION, THE UNIVERSITY OF TEXAS AT AUSTIN, December 2014.
19. 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.
20. Жуков В.Т., Новикова Н.Д., Феодоритова О.Б. *Параллельный многосеточный метод для разностных эллиптических уравнений. Часть I. Основные элементы алгоритма*. // Препринты ИПМ им. М.В. Келдыша. 2012. № 30. 32 с. URL: http://library.keldysh.ru/preprint.asp?id=2012-30
21. *Intel Xeon Phi*. URL: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html
22. *TOP500 Supercomputer Sites*. URL: http://www.top500.org
23. *Суперкомпьютер "Ломоносов"*. URL: http://parallel.ru/cluster/lomonosov.html
24. *Гибридный вычислительный кластер К-100*. URL: http://www.kiam.ru/MVS/resourses/k100.html
25. *Язык программирования НОРМА*. URL: http://keldysh.ru/pages/norma/
26. А.Н. Андрианов, К.Н. Ефимкин, И.Б. Задыхайло, Н.В. Поддерюгина. *Язык Норма*. // Препринты ИПМ им. М.В. Келдыша. 1985. № 165. 34 с.
27. Задыхайло И.Б., Пименов С.П. *Семантика языка Норма*. // Препринты ИПМ им. М.В. Келдыша АН СССР. 1986. № 139
28. Андрианов А.Н. *Организация циклических вычислений в языке Норма*. // Препринты ИПМ им. М.В. Келдыша. 1986. № 171.
29. Андрианов А.Н. *Синтез параллельных и векторных программ по непроцедурной записи на языке Норма. Диссертация на соискание ученой степени кандидата физико-математических наук*. М., 1990 г.
30. Андрианов А.Н. *Система Норма. Разработка, реализация и использование для решения задач математической физики на параллельных ЭВМ. Диссертация на соискание ученой степени доктора физико-математических наук*. М., 2001 г.
31. *DVM система.* URL: http://www.keldysh.ru/dvm/
32. *Liszt DSL*. URL: http://graphics.stanford.edu/hackliszt/
33. *The Blitz++ library. Official website.* URL: http://blitz.sourceforge.net/
34. *Blitz++ Library on SourceForge.net.* URL: http://sourceforge.net/projects/blitz/
35. *MATLAB – The Language of Technical Computing.* URL: http://www.mathworks.com/products/matlab/

36. *SIMIT Language*. URL: http://simit-lang.org/
37. *PETSc*. URL: http://www.mcs.anl.gov/petsc/
38. *OpenFOAM*. URL: http://www.openfoam.com
39. Peter Vincent, Freddie Witherden, Brian Vermeire, Jin Seok Park, Arvind Iyer. *Towards green aviation with python at petascale*. November 2016 SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Salt Lake City, Utah — November 13 - 18, 2016. ISBN: 978-1-4673-8815-3.
40. *Python Language*. URL: https://www.python.org/
41. *Mako Templates for Python*. URL:http://www.makotemplates.org/
42. Краснов М.М. *Операторная библиотека для решения трёхмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA*. // Математическое моделирование, 2015, т. 27, с. № 3, 109-120.
43. Краснов М.М. *Сеточно-операторный подход к программированию задач математической физики. Диссертация на соискание ученой степени кандидата физико-математических наук.* М., 2017.
URL: http://keldysh.ru/council/1/2017-krasnov/diss.pdf
44. Жуков В.Т., Краснов М.М., Новикова Н.Д., Феодоритова О.Б. *Сравнение эффективности многосеточного метода на современных вычислительных архитектурах.*
// Программирование, 2015. № 1. с. 21-31.
45. Жуков В.Т., Краснов М.М., Новикова Н.Д., Феодоритова О.Б. *Численное решение параболических уравнений на локально-адаптивных сетках чебышевским методом.*
// Препринты ИПМ им. М.В. Келдыша. 2015. № 87. 26 с.
URL: http://library.keldysh.ru/preprint.asp?id=2015-87
46. *NAS Parallel Benchmarks*
URL: http://www.nas.nasa.gov/publications/npb.html
47. 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.
48. Галанин М.П., Савенков Е.Б., Токарева С.А. *Применение разрывного метода Галеркина для численного решения квазилинейного уравнения переноса*. // Препринты ИПМ им. М.В. Келдыша, 2005. № 105. 31 с. URL: http://www.keldysh.ru/papers/2005/prep105/prep2005_105.html
49. Краснов М.М., Ладонкина М.Е., Тишкин В.Ф. *Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Использование операторного метода программирования.*
// Препринты ИПМ им. М.В. Келдыша. 2016. № 23. 27 с.
URL: http://library.keldysh.ru/preprint.asp?id=2016-23

50. 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. URL: http://library.keldysh.ru/preprint.asp?id=2016-24
51. Краснов М.М., Кучугов П.А., Ладонкина М.Е., Тишкин В.Ф. *Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Использование операторного метода программирования.* // Математическое моделирование, 2017, т. 29, № 2, с. 3-22.
52. Краснов М.М., Ладонкина М.Е. *Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++.* // Программирование, 2017 г., № 3, с. 56-68.
53. Елизарова Т.Г. *Квазигазодинамические уравнения и методы расчета вязких течений.* М.: Науч. мир, 2007. 352 с.
54. Davydov, Alexander A.; Shilnikov, Evgeny V. *Numerical Simulation of the Low Compressible Viscous Gas Flows on GPU-based Hybrid Supercomputers. International Conference on Parallel Computing* - ParCo2013, 10-13 September 2013, Munich, Germany.
55. B.N. Chetverushkin, E.V. Shilnikov, A.A. Davydov. *Numerical Simulation of Continuous Media Problems on Hybrid Computer Systems", in P. Ivanyi, B.H.V. Topping, (Editors),* "Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering", Civil-Comp Press, Stirlingshire, UK, Paper 25, 2012. doi:10.4203/ccp.95.25, 14 p
56. Деммель Дж. *Вычислительная линейная алгебра. Теория и приложения.* М.: «Мир», 2001 г., 430 с.
57. Самарский А.А., Николаев Е.С. *Методы решения сеточных уравнений.* М.: «Наука», Главная редакция физико-математической литературы, 1978 г., 592 с.
58. Боровик Е.В., Краснов М.М., Рыков Ю.Г., Шалыга Д.К. *Математическая модель и численная методика расчёта на основе ENO-схем течений в каналах переменного сечения с горением.* // Препринты ИПМ им. М.В. Келдыша. 2016. № 112. 20 с. URL: http://library.keldysh.ru/preprint.asp?id=2016-112

Подписано в печать 26.09.2017. Формат 60x84/16. Усл. печ. л. 4,9. Тираж 100 экз. Заказ М-16
ИПМ им. М.В. Келдыша РАН. 125047, Москва, Миусская пл., 4