УДК 519.683.8
DOI 10.53403/9785951505071_2022_135

ЭФФЕКТИВНОЕ МЕТАПРОГРАММИРОВАНИЕ АЛГОРИТМОВ МЕТОДА КОНЕЧНЫХ ЭЛЕМЕНТОВ СРЕДСТВАМИ ЯЗЫКА C++17

А. М. Гурин, А. Н. Байкин

Институт гидродинамики им. М. А. Лаврентьева СО РАН, Новосибирск

Работа посвящена применению функционального и метапрограмирования для разработки конечно-элементного решателя. Возможности современного C++ позволяют реализовать метод конечных элементов непосредственно через работу с функциями при помощи метафункций. Такой метод обобщает сборку локальной матрицы из слабой формы уравнения. Подход протестирован на решении уравнения Пуассона и выполнено сравнение с известными открытыми кодами.

Ключевые слова: метод конечных элементов, функциональное программирование, метапрограммирование.

Введение

Метод конечных элементов (МКЭ) позволяет дискретизовывать системы уравнений в частных производных на расчетных сетках, состоящих из элементов - малых объемов, для которых записан некоторый интерполирующий полином. Для уравнений записанных в слабой постановке возможно эффективно автоматизировать построение решателя, так как алгоритм перехода от уравнений в частных производных к дискретизации в виде системы линейных алгебраических уравнений (СЛАУ) достаточно прямолинеен. Тем не менее, реализация алгоритма в виде компьютерной программы представляет значительную сложность, так как математический язык плохо согласуется с языком программирования. Существуют известные программные решения реализующие МКЭ в обобщенном виде – deal.ii, FEniCS, FreeFEM++ [1], Comsol и множество других. Перспективный метод разработки таких обобщенных кодов это метапрограммирование. Оно позволяет избежать падения производительности, связанного с динамическим полиморфизмом, так как некоторая часть кода компилируется специально под решаемую задачу. В таком подходе реализованы deal.ii и FeniCS [2]. В лаборатории авторов был реализован МКЭ код FEMEngine с использованием шаблонного метапрограммирования на C++17. Такой подход позволил реализовать МКЭ напрямую в терминах функций и операций над ними.

Алгоритм МКЭ

МКЭ легко проиллюстрировать на решении уравнения Лапласа,

$$\Delta T = 0 \tag{1}$$

Для применения метода уравнение записывается в слабой постановке,

$$\int_{\Omega} \nabla T \cdot \nabla \psi d\Omega = 0 \tag{2}$$

Неизвестная функция $T(X)$ выражается в виде интерполяционного полинома,

$$T(X) = \sum_{i=1}^{N} T_i \phi_i \tag{3}$$

Теперь подставив $T(X)$ в слабую постановку и выбрав интерполяционные функции ϕ_i в качестве тестовых ψ можно получить систему линейных уравнений

$$\sum_{i=1}^{N} T_i \int \nabla\phi_i \cdot \nabla\phi_j d\Omega - 0, j - 1..N \tag{4}$$

$$\sum_{i=1}^{N} T_i M_{ij} = 0, j = 1..N \tag{5}$$

$$M_{ij} = \int \nabla\phi_i \cdot \nabla\phi_j d\Omega \tag{6}$$

Решение системы (5) является решением исходного уравнения (1). Алгоритм решения состоит из нескольких блоков. Сначала требуется выбрать или сконструировать функцию для сборки матрицы (6), аппроксимирующей слабую постановку (2). Для вычисления элементов матрицы требуется интегрировать интерполяционные функции на элементе, который представляет собой некоторую геометрическую фигуру. Для удобства интегрирование проводится численной квадратурой на фиксированном, геометрически простом (каноническом) элементе, к которому производится преобразование координат с реального элемента.

Умножение функций

Лямбда функции вместе с контейнером std::tuple могут быть использованы для хранения интерполяционных функций элемента как показано на рис. 1. Контейнер std::tuple допускает оптимизацию подстановкой. Эта оптимизация является ключевой для всего алгоритма, так как позволяет полностью избавиться от большого дерева вызовов метафункций. Количество и типы аргументов лямбда функции в современном C++ могут быть определены на стадии компиляции.

```
constexpr auto phi1 = [](
  std::tuple<double, double> r
  ){
    auto[xi, eta] = r;
    return -eta - xi + 1.0;
  }
...
std::tuple phi{phi1, phi2, phi3};
```

Рис. 1. Интерполяционные функции

Благодаря этому можно реализовать метафункцию показанную на рис. 2. Эта функция принимает две функции f1 и f2 с одинаковыми аргументами. Класс свойств функции function_traits узнает аргументы этих функций и, с использованием этой информации, конструируется третья функция, являющаяся произведением первых двух. Аналогичным способом можно реализовать метафункции для различных операций с двумя аргументами и даже общую функцию, которая принимает две функции и операцию над ними.

```
template<class F1,
         class F2,
         class ... Args>
auto multiply( F1 f1,
               F2 f2,
               std::tuple<Args...> ) {
  return [=]( Args... args ){
      return f1(args...) * f2(args...); };
}

template<class F1,
         class F2>
auto multiply( F1 f1, F2 f2) {
  return multiply(
    f1,
    f2,
    typename function_traits
    <decltype(&F1::operator())>::args{} );
}
```

Рис. 2. Умножение функций

Арифметические операции над функциями лежат в основе алгоритма. Следующая метафункция, показанная на рис. 3, это декартово произведение наборов функций, которое требуется для вычисления локальной матрицы (6). Оператор многоточие, разворачивающий блоки типов, может быть применен не только непосредственно сразу после блока, но и в других местах, имеющих семантический смысл. Например, многоточие в аргументах функций f1 и f2 на рис. 2 разворачивает аргументы, с которыми будет вызвана функция. Многоточие, примененное снаружи вызова функции сгенерирует множество вызовов функции с разными аргументами из блока. Такое использование многоточия показано на рис. 3.

```
template<class ...Args1,
         class ...Args2,
         size_t ...Is>
auto tensorProd(std::tuple<Args1...> t1,
                std::tuple<Args2...> t2,
                std::index_sequence<Is...>)
{
  return std::make_tuple(
    tupleBinaryOp(
      t1,
      std::get<Is>(t2),
      std::multiplies{}
    ) ... );
}

template<class ...Args1,
         class ...Args2>
auto tensorProd(std::tuple<Args1...> t1,
                std::tuple<Args2...> t2)
{
  return tensorProd(
    t1,
    t2,
    std::make_index_sequence
      <sizeof...(Args2)>{} );
}
```

Рис. 3. Декартово произведение функций

Интегрирование функции на элементе

Интегралы вида (6) обычно вычисляются при помощи квадратуры на каноническом элементе и преобразования координат. Интеграл на произвольном элементе выглядит следующим образом

$$\int_{\Omega_k} f\, d\Omega - \sum_{i=1}^{M} f(\vec{r_i}) \cdot w_i |J_k|, \tag{7}$$

где $\vec{r_i}$, w_i — это узлы и веса квадратуры и $|J_k|$ — Якобиан преобразования координат от канонического элемента к реальному элементу Ω_k в вычислительной области. Интегрирование, так же как умножение и декартово произведение, выполняется в функциональном стиле. Метафункция интегрирования принимает функцию и квадратуру и возвращает функцию, которая проинтегрирована неявно. Такая функция будет возвращать значение интеграла, принимая Якобиан. Также применяется метод частичного выполнения функции. В процессе интегрирования выполняется подстановка аргументов-координат и добавляется аргумент-матрица Якоби. В случае интегрирования производных функции, преобразование координат выглядит следующим образом

$$\frac{\partial \phi_i}{\partial x} = \frac{\partial \phi_i}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial \phi_i}{\partial \eta}\frac{\partial \eta}{\partial x} + \frac{\partial \phi_i}{\partial \zeta}\frac{\partial \zeta}{\partial x}, \tag{8}$$

где ξ, η, ζ — координаты в системе канонического элемента. Такое преобразование координат также выполняется в функциональном стиле с изменением сигнатуры функции.

Сборка матрицы

Вышеописанные метафункции используются для создания функции, генерирующей локальную матрицу. Например, для вычисления локальной матрицы из выражения

$$\int_{\Omega_k} \phi_i \phi_j d\Omega, i = 1, \dots, N, j = 1, \dots, N, \tag{9}$$

на треугольном элементе первого порядка выполняются следующие операции. Интерполяция на элементе задана тремя функциями,

$$\phi = |\phi_1, \phi_2, \phi_3|. \tag{10}$$

Выполняется декартово произведение вектора ϕ состоящего из функций самого на себя. Затем применяется преобразование координат и интегрирование. Полученная функция используется в цикле по элементам для вычисления локальных матриц и сборке их в глобальную.

Эффективность метода

Важная особенность примененной методологии это обобщение МКЭ. Одна и та же реализация вышеописанных функций используется для 1D, 2D, 3D элементов любого порядка. Такой подход менее подвержен ошибкам, так как большая их часть обнаруживается на стадии компиляции.

C++ сущности, такие как tuple и лямбда функции, примененные для реализации алгоритма, хорошо оптимизируются компилятором. Когда функция для сборки локальной матрицы генерируется метафункцией, компилятор удаляет огромное дерево вызовов к лямбдам, tuple и промежуточным метафункциям и оптимизирует окончательный код. Эффективность оптимизации была протестирована на простом интеграле для 1D случая

$$\int_{\Omega_k} T\, \phi_i \phi_j d\Omega \qquad i = 1,2; j = 1,2. \tag{11}$$

Значения коэффициента T вводятся юзером через стандартный поток ввода std::cin. Это важно, поскольку если значения T будут непосредственно внесены в код программы, тогда компилятор

вычисляет все значения матрицы на стадии компиляции и удаляет весь машинный код кроме вывода в терминал.

```
1    call sym std::istream
2    movsd xmm3, qword [rsp + 8]
3    movsd xmm1, qword [rsp + 0x10]
4    movsd xmm4, qword [0x00000db0]
5    movapd xmm5, xmm3
6    movsd xmm0, qword [0x00000da8]
7    movapd xmm2, xmm1
8    mulsd xmm5, xmm4
9    mulsd xmm1, xmm4
10   mulsd xmm2, xmm0
11   mulsd xmm0, xmm3
12   addsd xmm2, xmm5
13   addsd xmm0, xmm1
14   mulsd xmm2, qword [0x00000db8]
15   mulsd xmm0, qword [0x00000dc0]
16   addsd xmm0, xmm2
17   call sym std::ostream
```

Рис. 4. Дизассемблированный код

Тестовый код был скомпилирован компилятором gcc 7.4 с флагом оптимизации “-Ofast” и получившийся двоичный файл был дизассемблирован программой radare2. На рис. 4 показана часть дизассемблированного кода функции main. Этот код выполняется между вводом и выводом в терминал и состоит из низкоуровневых операций. Такой код был сгенерирован вложенными вызовами метафункций декартового произведения, интегрирования, интерполирования коэффициента и преобразования координат и итоговый машинный код состоит только из низкоуровневой арифметики. Таким образом, компилятор удалил все вложенные вызовы метафункций и выполнил оптимизацию кода.

Тест на решении уравнения Пуассона

Чтобы продемонстрировать применимость этого метода для решения задач на реальных сетках, решение уравнения Пуассона (14) выполняется на FEMEngine и сравнивается с широко известными открытыми исходными кодами FEniCS и FreeFEM++. Выражение в правой части уравнения (12) выбрано таким образом, чтобы формула (15) описывала точное решение. Граничные условия определяются выражением (16).

$$f(x,y,z) = 12(x^2 + y^2 + z^2), \tag{12}$$

$$\Delta T = f(x,y,z), \tag{13}$$

$$\int_\Omega \nabla T \cdot \nabla \psi d\Omega = -\int_\Omega f(x,y,z)\psi \, d\Omega, \tag{14}$$

$$T(x,y) = x^4 + y^4 + z^4, \tag{15}$$

$$T(x,y)|_{\partial\Omega} = x^4 + y^4 + z^4. \tag{16}$$

Для тестирования использовалась 1x1x1 кубическая геометрия разбитая на тетраэдрическую сетку из 256000 ячеек. Вычисления производились на процессоре Intel i7-6700K. FEMEngine вычисляет глобальную матрицу за 1.07 с, тогда как FreeFem++ затрачивает 8.32 с. Пакет FEniCS использует компиляцию во время выполнения, поэтому первый запуск расчета затрачивает 3.41 с на сборку матрицы, а последующие только 0.21 с. Численные результаты, полученные с помощью FEMEngine и FreeFEM++ на одной и той же сетке, полностью совпадают, если применяемая квадратура и решатель линейной системы идентичны. Решение FEniCS отличается на 0,03 %, как и ожидалось, поскольку используется другой решатель линейной системы. Численное решение показано на рис. 5.

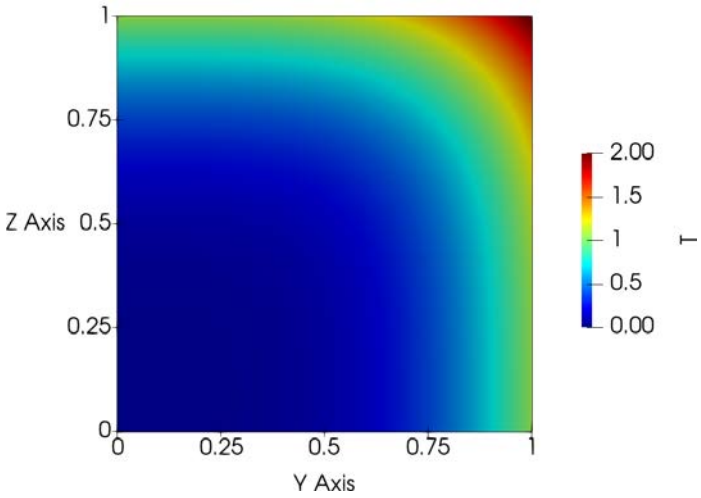

Рис. 5. Решение уравнения Пуассона в плоскости YZ, $x = 0.5$

FreeFEM++ анализирует файл “.edp”, содержащий постановку задачи и собирает матрицу во время выполнения. Коды, которые могут решать широкий класс задач, определяемых входными данными, полученными во время выполнения, обычно полагаются на затратный по времени динамический полиморфизм и сложную логику построенную на ветвлениях. FEniCS использует метод компиляции линковки во время выполнения вместе с генерацией кода C на Python. Оптимизированный код для решения задачи компилируется в динамическую библиотеку во время выполнения программы и загружается программой на ходу. Этот метод чрезвычайно сложно реализовать и поддерживать, но его производительность и универсальность являются одними из самых высоких. FEniCS собирает матрицу в 40 раз быстрее, чем FreeFEM++ для этой задачи. FEMEngine основан на шаблонах C++, поэтому программа компилируется для одной решаемой задачи. FEMEngine собирает матрицу примерно в 8 раз быстрее, чем FreeFEM++, и в 5 раз медленнее чем операции сборки матрицы FEniCS. Шаблонный код для генерации локальной матрицы жесткости основан на оптимизации, обеспечиваемой компилятором C++, которая уже достаточно эффективна и ее трудно улучшить.

Выводы

Разработана библиотека FEMEngine на основе шаблонного метапрограммирования C++ для конечно-элементного анализа.

Шаблонное метапрограммирование наряду с функциональным подходом имеет большой потенциал для разработки кода конечных элементов. Эти методы программирования позволяют писать надежный, универсальный и эффективный код.

Проведено сравнение FEMEngine и FreeFEM++ и достигнуто 8-кратное преимущество по времени построения глобальной матрицы. Сравнение с кодом FEniCS показывает, что существует потенциал для оптимизации узких мест текущего алгоритма сборки матрицы.

Литература

1. Hecht F. New development in freefem++ // J. Numer. Math. 2012. Vol. 20, № 3–4. P. 251–265.

2. Logg A., Mardal K. A., Wells G. N. Automated Solution of Differential Equations by the Finite Element Method. – Springer, 2012.

EFFECTIVE METAPROGRAMMING OF FEM ALGORITHMS ON C++17

A. M. Gurin, A. N. Baykin

Lavrentiev Institute of Hydrodynamics SB RAS, Novosibirsk

The paper is devoted to the use of functional and metaprogramming for the development of a finite element solver. The capabilities of modern C++ allow you to implement finite element method straightforwardly through metafunctions acting on functions. This method generalizes the assembly of a local matrix from the weak form of the equation. The approach is tested on solving the Poisson equation and compared with known open-source codes.

Key words: finite element method, functional programming, metaprogramming.