Методы метапрограммирования в компьютерном зрении: 7-точечный алгоритм и автокалибровка

О.С.Сидоркина, Д.В.Юрин
Факультет вычислительной математики и кибернетики
Московского государственного университета им. М.В.Ломоносова, Москва, Россия
sidorkina_olga@mail.ru, yurin_d@inbox.ru

Аннотация

Неотъемлемой частью практически всех систем восстановления трехмерных сцен является 7-точечный алгоритм, линейная задача автокалибровки также часто используется в таких системах. Эти задачи включают в себя вычисление коэффициентов полинома (3 или 4 степени) задаваемого как детерминант линейной комбинации двух матриц. Показано, как с помощью методов метапрограммирования на C++, для этой специфичной для области компьютерного зрения технической задачи построить решение с помощью компактного и ясного кода, значительная часть которого выполняется в момент компиляции программы до ее исполнения. Результатом компиляции является близкий к оптимальному ассемблерный код без циклов и условных переходов, реализующий вычисления по явной формуле, построенной компилятором. С детальным пояснением методов и алгоритмов приведен полный и переносимый исходный код, готовый к использованию при решении практических задач.

Keywords: *7-ми точечный алгоритм, автокалибровка, метапрограммирование, C++*

1. ВВЕДЕНИЕ

Традиционным языком программирования в задачах компьютерного зрения и компьютерной графики является C++. Этот выбор, помимо исторических причин, обусловлен высокой вычислительной сложностью характерных задач, когда операции надо выполнять над миллионами (а иногда и миллиардами) пикселей или полигонов. Однако, в настоящее время, наблюдается разрыв между огромными возможностями современных C++ и компиляторов и используемыми техниками программирования, зачастую берущими начало от языка C или «C с классами». Тем не менее, постепенно современные методы программирования проникают и в область компьютерного зрения, иногда – неявно, через используемые библиотеки, например матричных вычислений, таких как uBlas из библиотеки Boost [1], а в ряде случаев появляются разработки, специализированные для задач компьютерного зрения. В качестве таких положительных примеров можно отметить библиотеку VIGRA [2,3] и алгоритмы на основе минимизации энергии на графах [4,5], включенные в настоящее время в библиотеку Boost.

Настоящая работа посвящена некоторым техническим аспектам проблемы восстановления трехмерных сцен, а именно задачам вычисления фундаментальной матрицы по междукадровым точечным соответствиям [6, pp. 281, 291] с помощью 7-точечного алгоритма и автокалибровке с помощью абсолютной дуальной квадрики [6, p. 465]. В обоих случаях математически задача формулируется следующим образом. Ищется решение системы линейных уравнений

$$\mathbf{C}\vec{\mathbf{x}} = 0 \qquad (1)$$

с помощью сингулярного разложения как правые собственные векторы, принадлежащие нуль-пространству матрицы $\mathbf{C}$. Количество уравнений на два меньше, чем число переменных, поэтому в нуль-пространстве оказываются два собственных вектора и общее решение выражается их линейной комбинацией. Компоненты искомого вектора $\vec{\mathbf{x}}$ - суть элементы фундаментальной матрицы $\mathbf{F}$ для первой задачи и элементы матрицы абсолютной дуальной квадрики $\mathbf{Q}_\infty^*$ для второй. Для того, чтобы свести задачу к линейной, в системе уравнений (1) наложены не все ограничения. Матрица $\mathbf{F}$ имеет размер 3x3, определена с точностью до множителя и имеет ранг 2, а матрица $\mathbf{Q}_\infty^*$ имеет размер 4x4, симметрична (это требование учтено при построении системы (1)), и удовлетворяет условию $\det \mathbf{Q}_\infty^* = 0$. Через компоненты вектора x (1) эти матрицы выражаются как

$$\mathbf{F} = \begin{pmatrix} x_0 & x_1 & x_2 \\ x_3 & x_4 & x_5 \\ x_6 & x_7 & x_8 \end{pmatrix} \mathbf{Q}_\infty^* = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_1 & x_4 & x_5 & x_6 \\ x_2 & x_5 & x_7 & x_8 \\ x_3 & x_6 & x_8 & x_9 \end{pmatrix} \qquad (2)$$

Формируя из двух векторов нуль-пространства матрицы $\mathbf{C}$ две матрицы $\mathbf{X}_1$ и $\mathbf{X}_2$ (через $\mathbf{X}_i$ обозначаются матрицы $\mathbf{F}$ или $\mathbf{Q}_\infty^*$ соответственно решаемой задаче), решение, удовлетворяющее всем ограничениям ищется как $\mathbf{X} = (1-\lambda)\mathbf{X}_1 + \lambda\mathbf{X}_2$, где λ удовлетворяет уравнению

$$\det((1-\lambda)\mathbf{X}_1 + \lambda\mathbf{X}_2) = 0 \qquad (3)$$

Это алгебраическое уравнение 3 или 4 степени (для $\mathbf{F}$ или $\mathbf{Q}_\infty^*$ соответственно) и существуют стандартные численные методы решения таких уравнений.

Однако здесь возникает техническая сложность: по известным матрицам $\mathbf{X}_1$ и $\mathbf{X}_2$ вычислить коэффициенты полинома (3). Обычно используется один из двух подходов. 1) написать численный алгоритм, 2) получить явные

выражения для коэффициентов через элементы матриц с помощью программ аналитических вычислений (Maple, Mathematica). Первый подход неудовлетворителен потому, что вычисление этих коэффициентов требуется выполнять часто, например 7-точечный алгоритм работает внутри процедуры RANSAC, кроме того, вычисления надо производить для всех возможных пар изображений, участвующих в трехмерной реконструкции. Поэтому часто следуют второму подходу. Он тоже имеет свои недостатки. Помимо высокой стоимости программ аналитических вычислений, имеется следующая проблема. Выражения для коэффициентов оказываются весьма громоздкими, например для поиска $\mathbf{Q}_{\infty}^{*}$ выражение для каждого коэффициента занимает около листа формата A4. Это создает источник потенциальных ошибок при переносе полученного выражения в код программы, адаптации результата автоматического преобразования в C из программ аналитических вычислений к используемому типу контейнеров для матриц в C++, возможность случайных изменений кода, которые не могут быть обнаружены программистом визуально. Для компилятора же такой код представляет сложность в плане оптимизации, так как он дается как есть и для компилятора закон построения этого кода неизвестен. Здесь уместна аналогия: полином, представленный в виде произведения простых сомножителей очень легко преобразовать к стандартному виду, задача же разложения на простые множители (поиск корней) весьма нетривиальна.

В последующих разделах предлагается третий подход вычисления коэффициентов таких полиномов (3), основанный на технике метапрограммирования в C++ [7]. Суть подхода можно сформулировать как «***заставить компилятор построить явные выражения для коэффициентов полинома во время компиляции***». Значительная часть кода, приводимого далее, отрабатывает *в момент компиляции* и к моменту исполнения программы не существует и, естественно, не выполняется. Так как явные выражения строит компилятор с помощью несложных рекурсивных подстановок шаблонов, структура кода для компилятора прозрачна, что создает предпосылки для максимальной оптимизации. Код на C++, в свою очередь, хотя и несколько затейлив, вполне поддается анализу человеком, в отличие от огромных выражений, получаемых с помощью программ аналитических вычислений.

2. АЛГОРИТМ

Сначала отметим, что уравнение (3) можно переписать в виде

$$\det(\mathbf{A}+\lambda\mathbf{B})=0 \qquad (4)$$

где $\mathbf{A}=\mathbf{X}_1$, $\mathbf{B}=\mathbf{X}_2-\mathbf{X}_1$ и в дальнейшем будем работать именно с полиномом (4). Для вычисления детерминанта будем пользоваться формулой через сумму произведений элементов на четность перестановки индексов.

Будут использоваться два класса из [7], полный код которых приводится ниже для ясности изложения.:

```
class NullType {};
template<int v>  struct Int2Type
    { enum { value = v }; };
```

Класс `NullType` используется в качестве указателя конца списка, а класс `Int2Type` для обеспечения перегрузки функций ***по значению*** аргумента [7].

2.1 Список индексов и операции с ним

Для работы с индексами и их перестановками потребуется список индексов, который реализуется в виде класса, похожего на список типов из [7].

```
template<int i,class T=NullType>
struct IndexList {
  enum {value=i};
  typedef T Next;
};
```

Этот класс содержит собственно значение индекса в виде enum, который по стандарту C++ является константой времени компиляции и не может определяться во время исполнения и указатель на следующий элемент. Класс `IndexList` используется исключительно как тип, элементы этого класса никогда не создаются. Для работы с такими списками требуются ряд алгоритмов времени компиляции, которые будем помещать в **`namespace IL`** .

2.1.1 Создание последовательности индексов

Элементы матриц имеют индексы в диапазоне 0..(N-1). Для создания начальной четной перестановки служит класс

```
template<int i,int n>
struct MakeInRange
{
  typedef IndexList<i,typename
  MakeInRange<i+1,n>::Result > Result;
};
```

единственная задача которого заключается в определении типа **`Result`** путем рекурсивной подстановки самого себя в качестве шаблонного параметра в `IndexList` с возрастающим значением первого аргумента. Для окончания рекурсии служит частичная специализация класса.

```
template<int n>
  struct MakeInRange<n,n>
  {
    typedef NullType Result;
  };
```

Использование этого класса выглядит так:

```
typedef IL::MakeInRange<0,7>::Result MyLst;
```

Тип `MyLst` (а не элемент данных этого типа!) теперь содержит числа 0,1,2,3,4,5,6.

2.1.2 Добавление в конец списка

Алгоритм добавления в конец списка имеет 2 шаблонных параметра: добавляемое число и обрабатываемый список. Результатом является новый тип `Result`.

```
template<int i,class IList=NullType>
struct PushBack;
```

Частичная специализация: если список пустой, создать его:

```
template<int i>
struct PushBack<i,NullType >
{ typedef IndexList<i,NullType> Result;};
```

Частичная специализация: если список не пуст, то создать новый список, сначала помещая туда элементы из входного списка, а потом новое значение с помощью специализации определенной выше.

```
template<int i,int j,class Tail>
struct PushBack<i,IndexList<j,Tail> >
{
  typedef IndexList<j,typename
  PushBack<i,Tail>::Result > Result;
};
```

2.1.3 Вычисление длины списка

Алгоритм состоит в том, что перебираются элементы списка, каждый раз увеличивая счетчик на единицу до тех пор, пока не встретится тип конца списка (класс `NullType`)

```
template<class IList> struct Length;
template<> struct Length<NullType>
{  enum {value=0}; };
template<int i,class T>
struct Length<IndexList<i,T> >
{ enum { value=1+Length<T>::value };};
```

2.1.4 Индексированный доступ

Для доступа до i-го элемента, как в массиве, используется следующий алгоритм: если i≠0, то перейти к следующему элементу, уменьшив i на 1. К сожалению, в рамках вычислений в момент компиляции переопределить operator[] нельзя, так как операторы применяются к данным а не к типам и, следовательно, в момент компиляции недоступны.

```
template<int i,class IList> struct Elem;
```

Когда i-й элемент достигнут, следующая частичная специализация помещает его значение в перечисление `value`.

```
template<int j,class T>
struct Elem<0,IndexList<j,T> >
{ enum { value=j }; };
```

Если i-й элемент еще не достигнут, искать его рекурсивно и результат поместить в перечисление `value`:

```
template<int i,int j,class T>
struct Elem<i,IndexList<j,T> >
{ enum { value=Elem<i-1,T>::value }; };
```

Следующая частичная специализация служит для обработки ошибок, она подставляется тогда, когда индекс i выходит за пределы списка, т.е. достигнут последний элемент. Подробнее вопрос об обработке ошибок рассматривается в следующем разделе.

```
template<int i> struct Elem<i,NullType >
{ enum { value =
  ERROR__in_IndexList_usage<i>
           ::Index_out_of_range };
};
```

Синтаксис доступа до, скажем, 5-го элемента списка `MyLst`, введенного в 2.1.1. выглядит следующим образом:

```
enum{ x = IL::Elem<5, MyLst >::value };
```

2.1.5 Обработка ошибок

Поскольку все вычисления с введенным типом выполняются в момент компиляции, обработка ошибок должна производиться в то же время. Это является безусловным достоинством: либо код не компилируется, либо он не содержит ошибок, в то время как обработка ошибок во время исполнения и занимает время, и находится в зависимости от траектории исполнения программы. Часто ошибка обнаруживается в самый неподходящий момент, например при демонстрации программы заказчику. Возможности языка C++ в плане обработки ошибок вычислений в момент компиляции весьма ограничены, и, если не предпринять специальных мер, можно получить совершенно непонятное и пространное сообщение об ошибке. В задачах метапрограммирования такие сообщения компилятора могут выглядеть особенно ужасно. К сожалению, в данном контексте напрямую использовать введенную в [7] статическую проверку ошибок нельзя и надо искать другое решение. Предлагается для этой цели использовать класс.

```
template<int i>
class ERROR__in_IndexList_usage
{ enum Error {Index_out_of_range=i}; };
```

Тогда, если индексируемый элемент не находится в списке, то в ходе рекурсивных подстановок шаблонов в некоторый момент будет достигнут конец списка и будет применена последняя в разделе 2.1.4. частичная специализация. Компилятор попытается инициализировать поле `enum{value}` класса `Elem` значением закрытого (`private`) поля `Index_out_of_range`, доступ до которого не разрешен. При первичном же анализе эта специализация синтаксически верна: перечисление инициализируется значением некоторого другого перечисления. Результатом компиляции следующего кода

```
typedef IL::MakeInRange<0,3>::Result MyLst;
enum{ x = IL::Elem<4, MyLst >::value };
```

будет сообщение об ошибке типа:

«Нет доступа до закрытого поля `Index_out_of_range` перечисления `Error` класса `IL::ERROR__in_IndexList_usage`», т.е. вполне понятное сообщение, что и требовалось. В частности Visual Studio 2008 генерирует сообщение об ошибке:

```
error C2248: 'Index_out_of_range' : cannot access private
enumerator declared in class 'IL::
ERROR__in_IndexList_usage<i>'.
```

Вполне приемлемо и понятно. Какое именно неправильное значение индекса было подставлено можно понять проанализировав стек подстановок в окне «Output» оболочки Visual Studio. Компилятор GNU генерирует сообщение:

```
gc2009.meta.cpp:315: instantiated from here
gc2009.meta.cpp:23: error:
'IL::ERROR__in_IndexList_usage<1>::Error
IL::ERROR__in_IndexList_usage<1>::Index_out_of_range'
is private
```

2.1.6 Удаление i-го элемента

Алгоритм: двигаться по списку уменьшая счетчик (указанный индекс) и перекладывая элементы списка в новый список, пока не будет достигнут заданный элемент (нулевой счетчик), затем хвост списка добавить в конец нового списка. Обработка ошибки выхода индекса за пределы списка – в точности, как и в разделах 2.1.4., 2.1.5.

```
template<int i,class IList>
struct EraseAt;
template<int i>
struct EraseAt<i,NullType >
{ enum { value =
  ERROR__in_IndexList_usage<i>
           ::Index_out_of_range
  };
  typedef typename NullType Result;
};
```

```
template<int j,class T>
struct EraseAt<0,IndexList<j,T> >
{  typedef T Result; };
template<int i,int j,class T>
struct EraseAt<i,IndexList<j,T> >
{
  typedef IndexList<j,typename
  EraseAt<i-1,T>::Result> Result;
};
```

2.2 Перестановки

Теперь есть инструмент для работы со списками индексов и задача заключается в построении всех возможных перестановок этих индексов для вычисления детерминанта. Для каждой найденной перестановки, строго один раз должен вызываться функтор, который собственно и будет заниматься вычислением детерминанта матрицы. Предлагаемый класс одновременно с построением перестановок индексов вычисляет и четность перестановки. Класс содержит необходимые определения типов и единственную статическую функцию `forEach` и зависит от 5 шаблонных параметров – двух списков индексов, четности перестановки и двух индексов i и k. Начальное состояние предполагается следующим: первый список пуст, второй содержит тривиальную четную перестановку индексов элементов матрицы размера NxN: 0,1,…N-1. Значение нечетности (`isOdd`), разумеется, имеет значение 0- «ложь», индекс k равен длине списка индексов N, а индекс i=0.

Алгоритм состоит в следующем: из списка `IList2` удаляется один элемент (`i` - й) и помещается в конец списка `IList1`. Так как `IList1` и `IList2` типы, то для хранения результата определяются новые типы `Head` и `Tail`. Новая перестановка является конкатенацией двух этих списков. Четность перестановки при этом меняется, что легко вычисляется по номеру перемещенного элемента и помещается в поле `is_Odd` . Указанную операцию надо сделать для каждого возможного индекса i в списке `IList2`. Это – первая строка в коде функции `forEach` ниже. Разумеется, для хвоста списка `Tail` надо также перебирать все возможные перестановки, что делается во второй и последней строке функции `forEach`.

```
template<class IList1,class IList2,
           int isOdd,int k,int i>
 struct permut
 {
  enum { value=IL::Elem<i,IList2>::value };
  typedef typename
        IL::EraseAt<i,IList2>::Result Tail;
  typedef typename
   IL::PushBack<value,IList1>::Result Head;
  enum { is_Odd = (1&(isOdd+i)) };

  template<class Func>
  static void forEach(Func& f)
  {
    permut<IList1,IList2,isOdd,
    IL::Length<IList2>::value,i+1>
    ::forEach(f);

    permut<Head,Tail,is_Odd,
    IL::Length<Tail>::value,0>::forEach(f);
  }
 };
```

Теперь требуется задать условия останова такой статической рекурсии подстановок, что выполняется с помощью частичной специализации шаблона: условие – перебрали все возможные элементы в `IList2`.

```
template<class IList1,class IList2,
           int isOdd,int k>
struct permut<IList1,IList2,isOdd,k,k >
{
   template<class Func>
   static void forEach(Func& f) {}
};
```

В этой двух частичной специализации тело функции `forEach` пустое. Так как для шаблонных классов код генерируется компилятором только в момент инстанцирования и только для тех функций, которые используются, то здесь не будет никаких накладных расходов на «вызов функции», а будет непосредственно подставлено «пустое место». Осталось написать только специализацию, которая обеспечивает инстанцирование функтора, отвечающего собственно за вычисление в момент исполнения программы. Каждая полная перестановка построена в тот момент, когда второй список пуст:

```
template<class IList1,int isOdd>
struct permut<IList1,NullType,isOdd,0,0>
{
   template<class Func>
   static void forEach(Func& f)
   { f.template f<isOdd,IList1>(); }
};
```

Здесь следует обратить внимание на конструкцию со словом **`template`** предпоследней строке, которая может показаться непривычной. Эта конструкция служит для того, чтобы указать компилятору при первичном, *до инстанцирования*, просмотре кода и анализе его синтаксической правильности, что член `f` объекта `f` класса `Func` является шаблоном (шаблонная функция член), а не переменной, которая сравнивается (знак меньше) с параметром `isOdd`. Для компилятора Visual Studio 2008 это не столь существенно, так как он первичный синтаксический анализ выполняет весьма поверхностно. Однако, при использовании компилятора GNU отсутствие здесь слова **`template`** приводит к синтаксической ошибке, так как GNU строго следует спецификациям языка C++.

И, наконец, изолируем разработанный класс со слишком сложным интерфейсом в функцию, которая зависит только от двух параметров: размерность списка индексов (т.е. размер квадратной матрицы в рассматриваемых задачах компьютерного зрения) и функтор, который должен быть применен к каждой перестановке индексов.

```
template<int N,class Func>
inline void ForEachPermutation(Func& f)
{
   typedef typename IL::MakeInRange<0,N>
                      ::Result IList;
   permut<NullType,IList,0,N,0>
           ::forEach(f);
}
```

Реально сгенерированный компилятором код будет состоять из последовательности инстанцирований функтора

```
f.f<isOdd_0,IList_0>();
f.f<isOdd_1,IList_1>();
f.f<isOdd_2,IList_2>();
   ...
```

Для всех возможных перестановок индексов в начальном списке индексов с соответствующими четностями. Чтобы код был вычислительно эффективным, надо еще аккуратно задать функторы, так, чтобы при их последовательном инстанцировании компилятор смог выполнить необходимые оптимизации, опираясь на заданный закон изменения индексов в списках и четностей перестановок..

2.3 Функторы

Сначала рассмотрим более простую задачу вычисления детерминанта матрицы, а затем перейдем собственно к задаче о вычислении коэффициентов полинома. В последующем коде требуются контейнеры для хранения матриц. Тип контейнера не принципиален, требуется только, чтобы размер матриц был известен на момент компиляции. В настоящей реализации для работы с матрицами была выбрана библиотека uBlas из Boost [1]. Выбор был обусловлен как высоким качеством и широкой распространенностью этой библиотеки, так и наличием в ней наряду с классами векторов и матриц произвольного размера vector<>, matrix<> контейнеров для этих же объектов с фиксированным на момент компиляции размером c_vector<>, c_matrix<>. Контейнеры обоих типов используются единообразно и обрабатываются одними и теми же алгоритмами. Важным является также то, что эта библиотека полностью шаблонная, что позволяет достигать максимальной оптимизации кода современными компиляторами. Предполагается, что класс матриц определяет тип `value_type` своих элементов и имеет оператор () для доступа к i,j –му элементу.

Для реализации функторов дополнительно потребуется следующая тривиальная функция

```
template<int isSubtract,typename T>
void AddOrSub(T& result,T value)
{
   AddOrSub(result,value,
        Int2Type<(isSubtract?1:0)>() );
}
template<typename T>
void AddOrSub(T& result,
          T value,Int2Type<1> )
   { result-=value; }
template<typename T>
void AddOrSub(T& result,
          T value,Int2Type<0> )
   { result+=value; }
```

которая, в зависимости от значения своего первого шаблонного параметра, прибавляет или вычитает значение `value` к (из) переменной `result`. Здесь на основе введенного в [7] типа `Int2Type` (см. начало раздела 2) выполняется перегрузка функций по значению, а не по типу аргумента. Так как эта функция шаблонная с весьма простым и понятным кодом, результатом имплементации этого шаблона будет непосредственно требуемый оператор сложения или вычитания для встроенных типов.

2.3.1 Вычисление детерминанта

Код функтора приводится в листинге ниже.

```
template<class Matrix>
class det
{
  typedef typename Matrix::value_type
                        value_type;
  const Matrix& A;

  template<class Il,int i>
  value_type a()
  {
    return A(i,IL::Elem<i,Il>::value);
  }
  template<class Il,int i>
  value_type mul(Int2Type<i>  )
  {
    return a<Il,i>()
         *mul<Il,i-1>(Int2Type<i-1>());
  }
  template<class Il,int i>
  value_type mul(Int2Type<0> )
  { return a<Il,i>(); }
public:
  value_type result;
  det(const Matrix& mat)
     : A(mat) , result(0) {}
  template<int isOdd,class IList>
  void f()
  {
    enum {last=IL::Length<IList>::value-1};
    AddOrSub<isOdd>(result,
     mul<IList,last>(Int2Type<last>()) );
  }
};
```

Конструктор сохраняет в объекте константную ссылку на матрицу и обнуляет переменную `result`, в которой путем суммирования по перестановкам будет накапливаться значение детерминанта. Функция `a()` возвращает элемент матрицы из `i` –й строки и столбца, определяемого текущей перестановкой, заданной через список индексов `Il`. Эти параметры передаются как параметры шаблона и вычисляются в момент компиляции. Функция вычисления перестановок из раздела 2.2. вызовет этот функтор строго 1 раз для каждой перестановки, поэтому его функция `f()` должна соответственно значению четности `isOdd` прибавить или отнять произведение соответствующих элементов матрицы из `result`. Вычисление произведения элементов, соответствующих заданной перестановке осуществляется путем рекурсивной подстановки шаблонов с помощью функций `mul()`.

2.3.2 Вычисление коэффициентов полинома

Функтор вычисления коэффициентов полинома отличается от детерминанта тем, что теперь в каждый сомножитель в функции `mul()` предыдущего раздела является линейной комбинацией элементов двух разных матриц. В этих произведениях надо собирать коэффициенты при одинаковых степенях λ и помещать в соответствующий элемент массива коэффициентов `coef`. Размерность массива должна быть на 1 больше, чем размер матриц. Для инициализации этого массива используется следующая тривиальная функция:

```
template<int N,typename T,typename TT>
void set(T* dst,TT value)
{ set(dst,T(value),Int2Type<N>()); }
template<typename T,int N>
void set(T* dst,T value,Int2Type<N>)
{
   *dst=value;
   set(dst+1,value,Int2Type<N-1>());
}
template<typename T>
void set(T* dst,T value,Int2Type<0>) {}
```

Каждый элемент суммы (4) в формуле для детерминанта распадается на ряд сумм, в результате раскрытия скобок для каждого сомножителя, являющегося линейной комбинацией элементов двух матриц. Реализация этих подсумм должна выполняться в функторе, который требуется разработать. Предлагается для кодирования того, элемент какой матрицы входит в сумму, использовать битовую маску Бит 1 соответствует элементу матрицы при множителе λ, бит 0 – элементу второй матрицы. Количество значащих битов в маске должно быть равно N - размеру матриц. Максимальное значение числа, задаваемого маской тогда будет 2^N-1 и каждый элемент в сумме выражения для детерминанта через перестановки распадается на на 2^N слагаемых. Нетрудно заметить, что каждое такое слагаемое вносит вклад в коэффициент полинома при степени λ, равной *числу единичных бит в маске*. Для вычисления числа ненулевых бит в маске используется алгоритм из [8, стр. 76], который здесь реализуется в форме, позволяющей ему отрабатывать во время компиляции

```
template<unsigned n> struct BitsIn
{
  enum { temp=(n/2)+BitsIn<(n/2)>::temp };
  enum { value=n-unsigned(temp) };
};
template<> struct BitsIn<0>
{
  enum { temp=0 };
  enum { value=0 };
};
```

Поле `value` будет содержать количество единичных бит в числе `n`, являющемся параметром шаблона.

После этих предварительных замечаний теперь можно привести полный код функтора для вычисления коэффициентов искомого полинома (4):

```
template<class Matrix,int N>
class det_poly_coef
{
  typedef typename Matrix::value_type T;
  const Matrix& A;
  const Matrix& B;
  enum { mask = (1<<N)-1 };

  template<class Il,int i>
  T ab(Int2Type<0> )
  { return A(i,IL::Elem<i,Il>::value);}
  template<class Il,int i>
  T ab(Int2Type<1> )
  { return B(i,IL::Elem<i,Il>::value); }
  template<class Il,int i,int is_b>
  T ab()
  { return ab<Il,i>(Int2Type<is_b>()); }

  template<class Il,int i,int m>
  T mul(Int2Type<i> )
  {
    enum { m2 = m/2 };
    return ab<Il,i,m&1>() * mul<Il,
         (i-1),m2>( Int2Type<(i-1)>() );
  }
  template<class Il,int i,int m>
  T mul(Int2Type<0> )
  { return ab<Il,i,m&1>(); }

  template<int m,int isOdd,class IList>
  void one_comp()
  {
   enum {last=IL::Length<IList>::value-1};
   AddOrSub<isOdd>(
     coef[BitsIn<m>::value] ,
     mul<IList,last,m>(
        Int2Type<last>()) );
  }
  template<int m,int isOdd,class IList>
  void comp(Int2Type<1> )
  {
    one_comp<m,isOdd,IList>();
    comp<m-1,isOdd,IList>(
         Int2Type<((m-1)?1:0)>());
  }
  template<int m,int isOdd,class IList>
  void comp(Int2Type<0> )
  {
    one_comp<m,isOdd,IList>();
  }

 public:
  T coef[N+1];
  det_poly_coef(const Matrix& A_,
       const Matrix& B_): A(A_), B(B_)
  { set<N+1>(coef,0); }

  template<int isOdd,class IList>
  void f()
  {comp<mask,isOdd,IList>(Int2Type<1>());}
};
```

В приведенном коде для матриц A и B размером NxN, где N-параметр шаблона, конструктор сохраняет константные ссылки, и, с помощью введенной ранее функции `set()`, инициализирует нулями массив коэффициентов длины N+1. Поле `mask` инициализируется числом 2^N-1, как объяснялось выше. Функция `ab()`, по аналогии с функцией `a()` из раздела 2.3.1., обеспечивает доступ до элемента `i`-ой строки матрицы, соответствующего заданной перестановке `Il`. Дополнительный шаблонный параметр `is_b` определяет, из которой матрицы выбирается элемент: из B или A. Функция `mul()` вычисляет произведение указанных элементов матриц (выбирая элементы i-строки в соответствии с текущей перестановкой `Il` матриц A или B, в соответствии со значением текущей маски m). Функции `mul()` вызывается из функции `one_comp()` , которая в зависимости от четности перестановки добавляет или вычитает полученное произведение к (из) коэффициента полинома, определяемого количеством единичных бит в маске. Функция `comp()` обеспечивает вызов `one_comp()` для всех возможных значений маски `m` в диапазоне 0-`m`. Открытая функция этого функтора `f()` вызывает `comp()`,

задавая максимальное значение маски в соответствии с полем `mask`. Вызов же этой функции для каждой перестановки обеспечивается функцией `ForEachPermutation` из раздела 2.2., для которой этот функтор и предназначен.

Подчеркнем еще раз, что все эти «вызовы функций» не являются «call» как таковыми, а являются ***имплементациями шаблонов***, в ходе которых, ***в момент компиляции,*** производятся необходимые вычисления над целыми числами, которые, в конце концов, преобразуются в константные целочисленные индексы. Эти индексы указывают на элемент в матрице и определяют выбор матрицы, выбор прибавлять или вычитать произведение этих элементов и к которому коэффициенту полинома. Для вычислений в момент исполнения программы остается только перемножить указанные элементы указанных матриц и добавить их к нужным коэффициентам полинома. Никаких «циклов по i,j», рекурсий, «if -else» не возникает. Т.е. вычисление происходит по явной формуле. Более того, не вводится даже промежуточных переменных. Если это будет необходимо, компилятор это сделает сам, но не будет решать проблему заводить ли переменную как указано программистом, или ее можно оптимизировать в регистр. Более того, кроме единственного целого числа N, указывающего размер матриц, все остальные целые числа в получающейся формуле мы заставили сгенерировать сам компилятор в процессе компиляции по указанным в приведенном коде алгоритмам. Поэтому компилятор не только получает эти числа, но и «понимает» откуда они взялись и как взаимосвязаны между собой – он их сам только что вычислял! Все это обеспечивает максимально благоприятные условия для анализа и оптимизации кода компилятором.

3. ИСПОЛЬЗОВАНИЕ

В настоящем разделе приводятся листинги, демонстрирующие способ использования предложенных в предыдущем разделе алгоритмов и классов. Требуется включение заголовков

```
#include <iostream>
#include <boost/numeric/ublas/matrix.hpp>
```

из библиотек STL и uBlas [1], и поместить текст примеров в функцию main().

3.1.1 Вычисление детерминанта

```
using namespace boost::numeric::ublas;
using namespace std;
enum { N=2 };
typedef c_matrix<float,N,N> Matrix;
Matrix a;
a(0,0)=5.0f;  a(0,1)=13.0f;
a(1,0)=11.0f; a(1,1)=7.0f;
det<Matrix> d(a);
ForEachPermutation<N>(d);
cout <<" det=" << d.result << endl;
```

В результате выполнения этой программы будет напечатана строка: `det=-108`

3.1.2 Вычисление коэффициентов полинома

```
const int N=3;
typedef c_matrix<float,N,N> Matrix;
Matrix a;
a(0,0)=9.0f; a(0,1)=8.0f; a(0,2)=4.0f;
a(1,0)=7.0f; a(1,1)=1.0f; a(1,2)=3.0f;
a(2,0)=5.0f; a(2,1)=3.0f; a(2,2)=6.0f;
Matrix b;
b(0,0)=1.0f; b(0,1)=2.0f; b(0,2)=3.0f;
b(1,0)=4.0f; b(1,1)=5.0f; b(1,2)=6.0f;
b(2,0)=7.0f; b(2,1)=8.0f; b(2,2)=0.0f;
det_poly_coef<Matrix,N> d(a,b);
ForEachPermutation<N>(d);
cout << "coef= { " ;
for(int i=0; i<=N; ++i)
  cout << "  " << d.coef[i];
cout << " }" << endl;
```

В результате выполнения этой программы будет напечатана строка:
`coef= {  -179  243  42  27}`
т.е. построен полином: $-179+243*x+42*x^2+27*x^3$

4. ЗАКЛЮЧЕНИЕ

Неотъемлемой частью любой системы восстановления трехмерных сцен по набору изображений является 7-ми точечный алгоритм, частью которого является построение коэффициентов полинома. Этот алгоритм используется весьма интенсивно внутри процедуры RANSAC и для всех пар обрабатываемых изображений. Линейная задача автокалибровки также часто решается в составе систем восстановления трехмерных сцен и тоже включает в себя расчет коэффициентов аналогичного полинома (4).

В работе показано, как с помощью методов метапрограммирования с помощью небольшого объема достаточно прозрачного кода может быть получено универсальное и легко настраиваемое решение этих задач, специфичных для отрасли компьютерного зрения. Получается близкий к оптимальному ассемблерный код. Просмотр дизассемблированного кода показывает отсутствие вызовов функций, циклов, условных и безусловных переходов. Кроме считывания значений элементов матриц, все вычисления проходят в регистрах SSE[†]. Интересно, что компилятор GNU на приведенных тестах из раздела 3 вообще все вычисления выполнил в момент компиляции пользуясь тем, что элементы матриц были инициализованы константами. В реальных ситуациях, конечно, так происходить не будет, так как элементы матриц становятся известными только во время работы программы. Простейший способ посмотреть ассемблерный код для реальных ситуаций – это убрать слово **`inline`** в объявлении функции `ForEachPermutation` (см. конец раздела 2.2).

Приведенный код на C++ полностью соответствует стандартам языка и является переносимым. Приведенные

[†] Для Visual Studio надо указать Enable Enhanced Instruction Set – Streaming SIMD Extensions 2. В компиляцию были включены следующие ключи: /O2 /Oi /Ot /GL /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /EHsc /Gy /arch:SSE2.

При компиляции с помощью компилятора GNU версии 4.3.3 и 4.4.0 (ОС LINUX Ubuntu 9.04) использовалась командная строка: g++ -O4 -march=core2 -mfpmath=sse -g0 -DNDEBUG -finline-functions -S gc2009.meta.cpp

листинги являются ***полными***, будучи собранными (Copy-Paste из текста статьи) вместе, компилируются, верно работают и могут быть легко использованы при решении практических задач. При необходимости, код легко может быть настроен на использование других контейнеров для матриц. Вычисления производятся по явным формулам, которые строит сам компилятор в момент компиляции на основе идеи, обозначенной программистом. Не требуется выводить эти формулы вручную или использовать программы аналитических выражений и переносить оттуда в текст программы на C++ многие страницы неудобопонятных выражений.

Значительная часть приведенных кодов готова к повторному использованию. В частности, на базе введенного типа `IndexList` и алгоритмов работы с ним готовится библиотека высокооптимизированных и легко настраиваемых контейнеров и многомерных итераторов в стиле STL и [2] для единообразной работы с изображениями произвольной размерности (что актуально, например, в медицине) и с произвольным количеством цветовых каналов. Публикация на эту тему появится в ближайшее время.

5. БЛАГОДАРНОСТИ

Работа выполнена при поддержке грантов РФФИ 09-01-92470-МНКС_а, 09-07-92000-ННС_а. Авторы выражают благодарность инженеру фирмы Sensor-IC (Москва, Зеленоград) Осипенко А.С. за плодотворные обсуждения методов сигнализации об ошибках во время компиляции и помощь в тестировании кода на UNIX-платформах.

6. ЛИТЕРАТУРА

[1] *Boost libraries*. http://ww.boost.org

[2] U.Köthe. *STL-Style Generic Programming with Images // C++ Report Magazine 12(1), pp. 24-30, January 2000.* http://kogs-www.informatik.uni-hamburg.de/~koethe/ .

[3] U.Köthe, K.Weihe. *The STL Model in the Geometric Domain // in: M. Jazayeri, R. Loos, D. Musser (Eds.): Generic Programming, Proc. of a Dagstuhl Seminar, Lecture Notes in Computer Science 1766, pp. 232-248, Berlin: Springer, 2000*

[4] Y.Boykov, V.Kolmogorov. *An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision // In IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 9, pp. 1124-1137, Sept. 2004* http://www.csd.uwo.ca/faculty/yuri/Abstracts/pami04-abs.html .

[5] http://www.boost.org/doc/libs/1_39_0/libs/graph/doc/kolmogorov_max_flow.html

[6] R.Hartley, A.Zisserman. *Multiple View Geometry in Computer Vision // Cambridge University Press, 2004. - 672 p.*

[7] А. Александреску. *Современное проектирование на C++ // Серия C++ In-Depth, т.3.:Пер. с англ. -М.:Издательский дом "Вильямс",2002. -336 с. ил. -Парал. тит. англ.*

[8] Г.Уоррен, мл. *Алгоритмические трюки для программистов // Пер. с англ. –М.:Изд. Дом «Вильямс», 2003. -288 с.:ил. –Парал. тит. англ.*

Об авторах

Сидоркина Ольга Станиславовна– аспирантка лаборатории Математических методов обработки изображений факультета Вычислительной математики и кибернетики Московского государственного университета им. М.В.Ломоносова. sidorkina_olga@mail.ru

Юрин Дмитрий Владимирович, к.ф.-м.н., с.н.с. лаборатории Математических методов обработки изображений факультета Вычислительной математики и кибернетики Московского государственного университета им. М.В.Ломоносова. yurin_d@inbox.ru

Metaprogramming Techniques in Computer Vision: 7-point Algorithm and Auto-calibration

Olga S. Sidorkina, Dmitry V. Yurin
Department of Computational Mathematics and Cybernetics
Moscow State University, Moscow, Russia
sidorkina_olga@mail.ru, yurin_d@inbox.ru

7-point algorithm is an essential part of almost all 3D reconstruction systems based on images set. Frequently, a linear auto-calibration algorithm is a part of such systems too. Both algorithms include coefficients calculation of 3-d or 4-th order polynomial defined as determinant of linear combination of two matrices. We show how with metaprogramming techniques in C++, this specific for computer vision problem can be solved with compact and clear code. A significant part of this code executes in the compile time. The result of the code compilation is near optimal assembler code without loops and jumps, implementing calculation with direct formula, constructed by compiler. Ready-to-use complete C++ code is given with detailed methods and algorithms description.

Keywords: *7-point algorithm, auto-calibration, metaprogramming, C++*

About the authors

Olga S. Sidorkina is a PhD student at Laboratory of Mathematical Methods of Image Processing, Chair of Mathematical Physics, Faculty of Computational Mathematics and Cybernetics, Moscow Lomonosov State University. Her contact email is sidorkina_olga@mail.ru

Dmitry V. Yurin, PhD, is a senior scientist at Laboratory of Mathematical Methods of Image Processing, Chair of Mathematical Physics, Faculty of Computational Mathematics and Cybernetics, Moscow Lomonosov State University. His contact email is yurin_d@inbox.ru