Реализация n-мерного BMS-алгоритма и методы обобщённого программирования

А.М. Пеленицын

Аннотация

Описывается созданная программная реализация алгоритма Берлекэмпа–Месси–Сакаты (BMS-алгоритма). Излагается ряд методов обобщённого программирования, основанных на использовании шаблонных механизмов языка C++, как известных ранее, так и новых.

1. Введение. Алгоритм Берлекэмпа–Месси (BM-алгоритм), созданный в конце шестидесятых годов, нашёл многочисленные применения в различных областях математики и информатики, обзор последних наиболее важных результатов содержится в [1]. В связи с декодированием алгебро-геометрических кодов появилась необходимость обобщения BM-алгоритма на «многомерные последовательности».

Впервые такое обобщение предложил С. Саката [2], [3] и его теперь принято называть алгоритмом Берлекэмпа—Месси—Сакаты (BMS-алгоритмом). Приложения этого алгоритма к актуальным задачам помехоустойчивого кодирования обсуждаются в [4]. В [5] представлена реализация кодека для одного класса алгебро-геометрических кодов с использованием двумерного BMS-алгоритма.

В данной работе описывается реализация n-мерного BMS-алгоритма, опирающаяся, в основном, на [6] (также использовались идеи работ [7] и [3]). Детально рассматривается ряд известных, а также предлагаются новые шаблонные техники языка C++, разрабатывающие методику обобщённого программирования [8, гл. 14], [9, п. 7.2].

2. Инструменты реализации. Реализация выполнена на языке программирования C++ [10] с использованием стандартных расширений

TR1 [11] (поддержка программирования с STL), библиотеки NTL [12] (арифметика в конечных полях), части коллекции библиотек Boost [13] (поддержка программирования с STL), библиотеки Loki [14] (поддержка обобщённого программирования). Для трансляции использовался компилятор с языка C++ из свободной коллекции компиляторов GNU Compiler Collection (GCC) версии 4.4, в составе которого имеется, в частности, реализация TR1. Проект размещён в сети интернет на площадке Google Code [15]: исходные коды доступны для просмотра онлайн, кроме того, имеется доступ на чтение в Mercurial-репозиторий.

Результаты работы созданной реализации BMS-алгоритма проверялись с помощью примеров, приведённых в статьях [2] и [3]. Кроме того, для отдельных программных модулей был создан набор тестов, использующих свободную легковесную систему автоматизированного модульного тестирования CUTE (C++ Unit Testing Easier) [16]. Последняя имеет ряд преимуществ [17] перед другими программными пакетами данного класса (к примеру, CppUnit), а также предоставляет возможность тесной интеграции со свободной средой разработки Eclipse IDE [18], использовавшейся в разработке.

Основные программные сущности документированы в исходном коде в формате Doxygen [19], что позволяет извлекать документацию к проекту запуском соответствующей утилиты (возможные выходные форматы: HTML, LaTeX).

Eclipse IDE предоставляет средства интеграции почти всех перечисленных инструментов: связь с Mercurial-репозиторием (посредством плагина MercurialEclipse), развёрнутым на площадке Google Code, компиляция, линковка и отладка средствами GCC (посредством плагина CDT), визуальное отображение результатов тестов CUTE (посредством соответствующего плагина, предлагающегося на сайте CUTE), генерация тегов при добавлении Doxygen-комментариев к модулям программы.

3. Архитектура реализации. Была создана библиотека, реализующая работу с полиномами многих переменных в объёме, необходимом для выполнения BMS-алгоритма, а также сам алгоритм. Библиотека в терминологии C++ является «headers-only», то есть вся ре-

ализация содержится в заголовочных файлах C++, предназначенных для текстового включения в использующие её проекты и, таким образом, существует, прежде всего, в виде открытых исходных кодов. Она включает в себя три основных шаблонных класса, Point<N>, Polynomial <T>, BMSAlgorithm< PolynomialT > и набор вспомогательных шаблонных классов и свободных функций, занимающих в общей сложности около полутора тысяч строк кода (включая документацию внутри исходного кода). Наличие свободных функций является отступлением от чистого объектно-ориентированного дизайна и следует рекомендациям [20, п. 44], касающимся использования языка программирования C++ как мультипарадигменного. Также в соответствии с последним, в данной реализации особое внимание уделено применению обобщённого программирования на основе шаблонов C++. Объектно-ориентированный подход используется лишь в малой степени, как дополнительное средство поддержки модульности; наследование применяется лишь в нескольких местах для удобства реализации идиом обобщённого программирования; не используется (динамический, основанный на виртуальных функциях) полиморфизм.

Шаблон класса Point<N> моделирует точку в N-мерной целочисленной решётке. С точки зрения реализации он представляет собой обёртку над шаблоном tr1 :: array<N, **int**> из библиотеки TR1, который, в свою очередь, является статическим массивом с STL-совместимым интерфейсом [11, п. 6.2]. К отдельным координатам точки можно обращаться с помощью операции взятия индекса (**operator**[]). Важной частью интерфейса Point<N> являются функции, реализующие различные упорядочения на множестве точек решётки, такие как byCoordinateLess и totalLess . Первая (свободная функция) обеспечивает покоординатное сравнение точек (частичное упорядочение), вторая (функция-член) задаёт специальное мономиальное упорядочение [6, с. 7], которое иногда называют градуированным антилексикографическим упорядочением. Кроме того, имеется набор свободных функций, реализующих поиск в наборе точек минимумов и максимумов относительно разных порядков, поиск в наборе точек точки, меньшей заданной, и т. п. Всё это необходимо выполнять

на разных стадиях BMS-алгоритма.

Реализация totalLess в форме функции-члена (в отличие от других функций, связанных с упорядочением) связана с желанием параметризовать тип точки мономиальным упорядочением. Такая параметризация доступна с помощью использования второго параметра шаблона Point< N>, который имеет значение по умолчанию, соответствующее градуированному антилексикографическому упорядочению (по этой причине, а также для краткости, мы не указываем его в тексте статьи). Этот приём будет отдельно описан в следующем пункте.

Шаблон класса Polynomial<T> представляет тип полинома, причём типовый параметр T обозначает тип коэффициентов соответствующего множества полиномов. Технически Polynomial<T> является особого рода контейнером для элементов типа T. Полиномы многих переменных получаются использованием того факта, что, к примеру, полином от двух переменных с коэффициентами типа T неотличим по алгебраическим свойствам от полинома от одной переменной с коэффициентами — полиномами от одной переменной и коэффициентами типа T (см., например, [21, гл. IV, § 1, п. 5]); таким объектам соответствует тип Polynomial< Polynomial<T>>, который можно получить, используя рассматриваемую библиотеку.

Для удобства использования полиномов многих переменных введён отдельный шаблон класса MVPolyType<n, T>, где первый параметр указывает на количество переменных в результирующем типе. Например, следующие две строки кода определяют переменные одинакового типа, представляющего полином от двух переменных с целочисленными коэффициентами.

```
Polynomial< Polynomial<int> > p;
MVPolyType<2, int>::ResultT q;
```

Полиномы можно задавать в специальном строковом представлении, перечисляя коэффициенты полинома в квадратных скобках через пробел в порядке возрастания степеней. Например, ‘‘[2 0 1]’’ соответствует полиному $x^2 + 2$. Так как полином от нескольких переменных это обычный полином с коэффициентами — полиномами от переменных, число кото-

рых меньше на единицу, то строковое представление таких полиномов будет содержать вложенные скобки. Например, ‘‘`[[2 0 2] [1] [0 3]]`’’ соответствует полиному $3x^2y + 2y^2 + x + 2$ (с точностью до возможного переименования переменных).

Для типов полиномов перегружены арифметические операции, которые необходимы для реализации BMS-алгоритма. Это сложение полиномов, умножение полинома на скаляр и умножение полинома на моном (**operator**<<) — таким образом, из основных операций с полиномами не реализовано лишь умножение. Поясним использование операции умножения на моном. Считается, что моном задаётся своей степенью, которая, в свою очередь, представлена типом Point<N>. Вот пример умножения полинома от двух переменных на моном x^3y^2:

```
MVPolyType<2, int>::ResultT p("[[1 2] [3]]");
Point<2> mon; mon[0] = 3; mon[1] = 2;
p <<= mon;
```

Стоит отметить, что так как шаблоны и перегрузка операций гарантируют контроль типов времени компиляции, то замена в данном примере двоек в аргументах шаблонов на два произвольных несовпадающих числа привела бы к ошибке компиляции.

Для полиномов многих переменных нельзя однозначно («наилучшим образом») определить степень. Для выполнения BMS-алгоритма требуется задавать некоторое мономиальное упорядочение на множестве мономов, и определить степень возможно. Однако работа со степенями полиномов, получаемых на каждой итерации алгоритма, ведётся в значительной степени независимо от самих полиномов: на каждой итерации сначала определяется геометрический образ очередного приближения к решению, составленный из степеней будущих полиномов в N-мерной решётке, а затем для каждой степени строится полином. Эти факты привели к такому решению: в данной реализации тип полинома не хранит никакой информации о своей степени (что бы ни подразумевалось под степенью полинома от нескольких полиномов), если такая информация нужна клиенту типа, он должен хранить и обновлять её самостоятельно.

Шаблон класса BMSAlgorithm<PolynomialT> параметризован типом

полиномов (в частности, алгоритм может работать с полиномами различного числа переменных) и имеет довольно простой интерфейс, отражающий назначение и использование BMS-алгоритма. Напомним, что BMS-алгоритм по заданной конечной n-мерной последовательности строит конечное множество полиномов от n переменных, называемое минимальным множеством этой последовательности. С технической точки зрения, n-мерная последовательность ничем не отличается от полинома от n переменных, потому тип соответствующего аргумента алгоритма совпадает с типом полиномов, получаемых на выходе алгоритма. Конструктор класса принимает последовательность и точку, обозначающую конец последовательности, алгоритм обрабатывает все элементы в диапазоне от элемента с мультииндексом $(0, \ldots, 0)$ до этого маркера конца последовательности в порядке, заданном используемым мономиальным упорядочением (напомним, что конкретное упорядочение является частью описания типа точки). Метод данного шаблона класса computeMinimalSet возвращает коллекцию с STL-совместимым интерфейсом, содержащую выход алгоритма. Тип этой коллекции является частью определения шаблона класса, к нему можно обратиться следующим образом: BMSAlgorithm<PolynomialT>::PolynomialCollection.

4. Шаблонные техники. Главным проектным решением библиотеки, опирающимся на особенности шаблонов C++, является рекурсивное определение полиномов многих переменных: Polynomial<...Polynomial<T>...>, где глубина вложенности равна n, моделирует тип полинома от n переменных. Идея типа, параметризованного самим собой, так называемое «рекурсивное инстанцирование», уже стало достаточно известным шаблоном проектирования (см., к примеру, подробный, хотя и несколько более сложный, чем наш, пример, описывающий создание кортежей из значений произвольного набора типов [8, гл. 21]).

При появлении рекурсивной структуры возникает вопрос о способе её обхода. В разных ситуациях это может потребовать более или мене сложных шаблонных техник. Приведём некоторые из них. Заметим, что часть примеров касается перегруженных операций для полиномов, и поскольку в реализации использована «каноническая форма» арифме-

тических операций, то вся реализация помещена в присваивающие версии операций, которые реализованы как функции-члены шаблона класса Polynomial<T> [20, п. 27].

При использовании рекурсии в инстанцировании шаблонного типа обычно предпринимаются попытки для облегчения её записи. В данном случае для этой цели создан шаблон MVPolyType<n, T>. Его определение достаточно лаконично и даёт представление о типичном подходе к работе с рекурсивно инстанцируемыми типами.

```
template<int VarCnt, typename Coef>
struct MVPolyType {
    typedef Polynomial< // recursive "call" to MVPolyType
        typename MVPolyType<VarCnt - 1, Coef>::ResultT> ResultT;
};

template<typename Coef>
struct MVPolyType<1, Coef> {
    // multivariate polynomial from 1 variable
    // is just Polynomial
    typedef Polynomial<Coef> ResultT;
};
```

Как видно, рекурсия проводится по количеству переменных полинома, для полинома одной переменной определена специализация шаблона, которая отвечает за остановку рекурсии. Рекурсия по типу, таким образом, не сильно отличается от простейших алгоритмов, использующих рекурсию по данным.

Аналогичным образом внутри типа полинома Polynomial<T> задаётся константа VAR_CNT, определяющая количество переменных полинома в зависимости от типа T, и синоним типа CoefT для обозначения реального типа коэффициентов полинома. Например, Polynomial<Polynomial<**int**>>::VAR_CNT равно 2 и Polynomial<Polynomial<**int**>>::CoefT обозначает **int**.

Изложенное выше связано с обходом и получением информации о самом типе, имеющем рекурсивное определение. Далее рассматривается работа с объектами такого типа.

Операция умножения полинома на скаляр не требует никаких дополнительных усилий по сравнению со случаем полинома от одной пере-

менной.

```
Polynomial operator*=(CoefT const & c) {
    for (typename StorageT::iterator it = data.begin();
            it != data.end(); ++it) {
        (*it) *= c;
    }
    return *this;
}
```

Тип полинома является особого рода контейнером и реализован на базе одного из стандартных контейнеров, который скрыт за синонимом типа StorageT. Соответствующее поле шаблона класса называется data.

Под умножением на скаляр понимается вызов для полинома операции '*' с объектом, тип которого совпадает с CoefT. Непосредственно умножение производится вызовом той же операции для каждого элемента контейнера data. Если мы имеем дело с полиномом одной переменной, то data должен содержать элементы типа CoefT, для которого, таким образом, должна быть определена операция '*' (тип коэффициентов полинома должен допускать умножение). В ином случае, снова будет вызвана приведённая выше функция, однако изменится тип Polynomial — теперь это будет тип полинома от переменных, число которых меньше на единицу. Ещё раз подчеркнём, что описанная логика полностью реализована в приведённом выше коде, не требуется, к примеру, использовать специализацию шаблонов, чтобы явно останавливать рекурсию.

Сложение двух полиномов так же, как умножение полинома на скаляр, не требует дополнительных модификаций по сравнению с тем, что можно было бы закодировать, не рассчитывая на использование в контексте рекурсивного определения. Можно сделать вывод о том, что такого рода рекурсивный обход одного (умножение на скаляр) или нескольких (сложение, сравнение двух полиномов на равенство) объектов одного рекурсивно инстанцированного типа может быть доволен удобен. Однако при работе с полиномами многих переменных возникает задача рекурсивного обхода объектов двух разных типов одновременно (при этом глубина рекурсии разных типов должна быть одинаковой). Примерами решения такой задачи в реализации данной библиотеки являются

операции умножения полинома на моном и обращения к коэффициенту полинома от многих переменных по его мультииндексу (иначе: по мультистепени монома, при котором стоит требуемый коэффициент). В обоих случаях требуется осуществлять одновременный обход объекта типа Polynomial<...Polynomial<T>...> и объекта типа Point<N>, где число N совпадает с глубиной вложенности типа полинома (а значит, в соответствии с нашими соглашениями — с числом переменных полинома).

Рассмотрим реализацию операции обращения к коэффициенту полинома по его (мульти)индексу. В частном случае, при работе с полиномом от одной переменной, обращаться к коэффициенту хотелось бы по обычному целочисленному индексу — по этой причине реализована версия **operator**[](**int**). Для обращения к коэффициенту полинома от многих переменных предоставляется перегруженная версия **operator**[](Point <VAR_CNT>) (размерность точки должна совпадать с количеством переменных полинома). Обсудим вопрос о том, как должна продвигаться рекурсия.

Пусть задана точка p типа Point<VAR_CNT>. Для полинома от многих переменных нужно обратиться к p[0]-му элементу в контейнере data (это полином от переменных, количество которых меньше на одну) и снова вызвать для него **operator**[](Point<VAR_CNT>) с аргументом, представляющим как бы срез точки p: точку со всеми элементами p, кроме p[0]-го. Для этой цели был создан шаблон класса Slice<Dim, Offset>, который хранит ссылку на исходную точку размерности Dim и поддерживает операцию взятия индекса, причём при обращении к i-ому элементу по индексу возвращается Offset+i-ый элемент исходной точки. Используя этот шаблон в реализации **operator**[](Point<VAR_CNT>) хотелось бы написать что-то наподобие (data[p[0]])[make_slice(p)], где функция make_slice создаёт срез, исключающий первый элемент точки. При этом в интерфейс полинома нужно добавить **operator**[](Slice<Dim, Offset>), в котором будет написано примерно то же, что и в исходной версии этой операции, имеющей в качестве аргумента точку. К сожалению, точно так написать нельзя по причине того, что в таком варианте не получается остановить рекурсию.

Для остановки рекурсии в шаблонных типах обычно используется специализация шаблона (как в примере в MVPolyType). В данном случае нужно было бы создать специализацию **operator**[](Slice<Dim, Dim−1>). Однако существует ограничение языка C++ [8, п. 12.3.3], которое, в частности, запрещает специализировать шаблонные члены шаблонов класса (именно таким является **operator**[](Slice<Dim, Offset>)). Так появляется ещё один уровень косвенности, вместо (data[p[0]])[make_slice(p)] используется вызов отельной свободной функции apply_subscript(data[p[0]], make_slice(p)), которая либо продвигает рекурсию, вызывая **operator**[](Slice<Dim, Offset>) с переданным ей срезом, либо (в своей специализации) останавливает рекурсию, вызывая **operator**[](**int**) с нулевым элементом полученного среза (этот элемент стоит в последней координате исходной точки).

Удобно считать, что при обращении по индексу, выходящему за пределы текущего контейнера, нужно вернуть «нулевое значение». Такое значение получается с помощью шаблона CoefficientTraits. Итоговый код, реализующий операцию обращения по индексу выглядит так:

```
template<typename T, typename S, typename Pt>
T applySubscript(S const & el, Pt const & pt) {
        return el[pt]; // call for operator[](Slice<...>)
}

template<typename T, typename S, int Dim>
T applySubscript(S const & el, Slice<Dim, Dim - 1> const & pt) {
        return el[pt[0]];  // call for operator[](int)
}

template<typename T>
typename Polynomial<T>::CoefT
Polynomial<T>::operator[](Point<VAR_CNT> const & pt) const {
    if (pt[0] < (int)0 || (int)data.size() <= pt[0])
        return CoefficientTraits<CoefT>::addId();
    else
        return applySubscript<Polynomial<T>::CoefT>(data[pt[0]],
            make_slice(pt));
}
```

Мы не приводим здесь код **operator**[](Slice<...>), так как он полностью

повторяет **operator**[](Point<VAR_CNT>), а **operator**[](**int**) отличается только тем, что вместо вызова apply_subscript стоит data[pt] (pt — полученное целое число).

Остановимся теперь на использованных шаблонных техниках, не связанных напрямую с идеей рекурсивной реализации полиномов от нескольких переменных. Упомянутый выше шаблон CoefficientTraits представляет собой пример класса характеристик или, в другом переводе, свойств (type traits) [8, гл. 15], [22, п. 2.10]. В данном случае, этот класс предоставляет необходимую полиному информацию о типе коэффициентов, а именно, значение этого типа, играющее роль аддитивно нейтрального элемента (см. пример выше), мультипликативно нейтрального CoefficientTraits<T>::multId(), и функцию, возвращающую аддитивно обратный элемент по заданному CoefficientTraits<T>::multId(T). Применение характеристик основано на использовании специализаций шаблонов. Если клиент шаблона класса полинома инстанцирует его с типом Bar для коэффициентов, нулевое значение которого хранится в статической константе Bar::ZERO, то клиенту нужно предоставить специализацию CoefficientTraits, функция addId которого возвращает Bar::ZERO. Имеется общее определение CoefficientTraits<T>, которое будет использовано, если клиент не сделает последнего, оно реализует наиболее принятое поведение (например, нулевое значение создаётся с помощью вызова конструктора без аргументов типа T).

При описании техники характеристик типов обычно упускают один важный вопрос, а именно: как избежать дублирования в разных инстанциях шаблона характеристик. Решение этой проблемы предложено в данной библиотеке и оно основывается на использовании списков типов [22, гл. 3] и наследования. Сама проблема в нашем случае возникает из-за того, что библиотека NTL, которая используется для арифметики в конечных полях, имеет четыре класса для полей разного типа: простых и расширенных и характеристики два и характеристик, отличных от двух. При этом все четыре класса имеют почти идентичный интерфейс, потому при создании специализаций CoefficientTraits для них пришлось бы четыре раза повторять один и тот же код. Этот код был выделен в шаблон клас-

са NtlCoefficientTraits<T>, и CoefficientTraits<T> наследуется от него, если T является одним из типом NTL или от «реализации по умолчанию» DefaultCoefficientTraits<T> в противном случае. Проверка принадлежности осуществляется метаоператором Select<**bool** flag, T, U> из библиотеки Loki, внутри которого определён синоним типа Result, совпадающий с типом T, если flag равен **true** и совпадающим с U в противном случае. Таким образом, определение CoefficientTraits выглядит так:

```
template<typename CoefT>
struct CoefficientTraits : public Loki::Select<
            isNtlType<CoefT>::result,
            NtlCoefficientTraits<CoefT>,
            DefaultCoefficientTraits<CoefT>    >::Result {};
```

isNtlType — небольшой шаблон класса, созданный на основе списков типов (реализация которых также взята из Loki):

```
template<typename T>
class isNtlType {
    typedef LOKI_TYPELIST_4(NTL::GF2, NTL::GF2E,
                NTL::ZZ_p, NTL::ZZ_pE) NtlFiniteFieldTypes;

public:
    enum { result = Loki::TL::IndexOf<NtlFiniteFieldTypes, T>::
        value >= 0 };
};
```

Список типов (typelist) это тип, хранящий информацию о других типах и позволяющий выполнять все те операции, которые характерны для списков. В частности, с помощью шаблона IndexOf<TList, T> можно получить индекс типа T в списке TList или -1, если T в нём отсутствует.

Последняя шаблонная техника, которую мы обсудим, обеспечивает параметризацию типа точки используемым в алгоритме мономиальным упорядочением, она основана на классах стратегий (policy classes) [22, гл. 1]. Стратегия — это связанный набор действий, связанных с разрабатываемым типом, ортогональный к остальной функциональности этого типа, нуждающийся в параметризации. Способ упорядочения точек N-мерной целочисленной решётки — хороший претендент на вынесение его кода в стратегию. Общая идея реализации состоит в том, чтобы доба-

вить в разрабатываемый шаблонный тип ещё один шаблонный параметр шаблона и наследоваться от него.

```
template<
    typename T,
    template <typename> class Policy = DefaultPolicy>
class MyClass : public Policy<T> { /* ... */ };
```

В классической работе [22] предлагается публичное наследование, при котором составляющие базового класса попадают в интерфейс производного. В нашей реализации клиенту не предоставляется непосредственный доступ к функциям базового класса, вместо этого их вызов помещён в интерфейсные функции производного класса (в данном случае, типа точки). Например, преобразование данной точки в следующую за ней относительно мономиального упорядочения:

```
template<int Dim, template <typename PointImpl> class
   OrderPolicy>
inline Point<Dim, OrderPolicy>&
Point<Dim, OrderPolicy>::operator++() {
    inc(data);
    return *this;
}
```

Функция inc() определена в шаблоне OrderPolicy, от которого наследуется Point. Аналогично

```
    bool operator<(Point<Dim> const & other) const {
        return totalLess(data, other.data);
    }
```

Функция totalLess() также определена в шаблоне OrderPolicy. Таким образом, шаблону OrderPolicy следует знать о типе поля data точки, последний используется в качестве параметра шаблона OrderPolicy.

```
template<
    int Dim, // point dimension
    template <typename PointImpl> class OrderPolicy
        = GradedAntilexMonomialOrder>
class Point : OrderPolicy< std::tr1::array<int, Dim> > { /* ...
   */ };
```

Список литературы

[1] *Куракин В.Л.* Алгоритм Берлекэмпа—Месси над коммутативными артиновыми кольцами главных идеалов // Фундаментальная и прикладная математика. 1999. Т. 5, вып. 4. С. 1061–1101.

[2] *Sakata S.* Finding a minimal set of linear recurring relations capable of generating a given finite two–dimensional array // J. Symb. Comp. 1988. Vol. 5. Pp. 321–337.

[3] *Sakata S.* Extension of the Berlekamp—Massey algorithm to N dimensions // Inform. and Comput. 1990. Vol. 84. No. 2, P. 207–239.

[4] *Sakata S.* The BMS algorithm and Decoding of AG Codes // In Sala M. et al. (ed.), Gr obner bases, coding, and cryptography. Springer, 2009. P. 143–163.

[5] *Маевский А.Э., Пеленицын А.М.* Реализация программного алгебро-геометрического кодека с применением алгоритма Сакаты // Изв. ЮФУ. Технические науки. 2008. № 8. С. 196–198.

[6] *Cox D.A., Little J.B., O'Shea D.B.* Using Algebraic Geometry, Second Edition. Springer, 2005. 496 p.

[7] *Sakata S.* The BMS algorithm // In Sala M. et al. (ed.), Gröbner bases, coding, and cryptography. Springer, 2009. P. 143–163.

[8] *Вандевурд Д., Джосаттис Н.* Шаблоны C++: справочник разработчика. М.: Вильямс, 2003. 544 с.

[9] *Stroustrup B.* Evolving a language in and for the real world: C++ 1991–2006 // Procs. of the ACM HOPL-III. 2007. Pp. 4-1–4-59.

[10] *International Standards Organization*: Programming Languages – C++. International Standard ISO/IEC 14882:1998.

[11] *International Standards Organization*: C++ Library Extensions. International Standard ISO/IEC TR 19768. URL: `http://www.`

open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf (дата обращения: 03.08.2010).

[12] *Shoup V.* NTL: A Library for doing Number Theory // URL: http://shoup.net/ntl/ (дата обращения: 03.08.2010).

[13] Boost C++ Libraries. URL: http://www.boost.org/ (дата обращения 03.08.2010).

[14] Loki C++ library // URL: http://loki-lib.sourceforge.net/ (дата обращения: 25.08.2010).

[15] *Pelenitsyn A.* Multivariate polynomials for C++ // URL: http://code.google.com/p/cpp-mv-poly/ (дата обращения: 03.08.2010).

[16] CUTE: C++ Unit Testing Easier // URL: http://r2.ifs.hsr.ch/cute (дата обращения: 03.08.2010).

[17] *Sommerlad P.* C++ Unit Testing Easier: CUTE // URL: http://wiki.hsr.ch/PeterSommerlad/wiki.cgi?CuTe (дата обращения: 03.08.2010).

[18] Официальный сайт Eclipse IDE // URL: http://www.eclipse.org/ (дата обращения: 03.08.2010).

[19] Официальный сайт Doxygen // URL: http://www.stack.nl/~dimitri/doxygen/ (дата обращения: 03.08.2010).

[20] *Саттер Г., Александреску А.* Стандарты программирования на C++. М.: Вильямс, 2005. 224 с.

[21] *Бурбаки Н.* Алгебра (Многочлены и поля. Упорядоченные группы). М.: Наука, 1965. 300 с.

[22] *Александреску А.* Современное проектирование на C++. М.: Вильямс, 2002. 336 с.