Arageli user's guide.[1]

Н. А. Гонова А. М. Камаев

24 февраля 2006 г.

[1]Задание для мини-проекта Arageli в ИТЛаб.

Оглавление

Глава 1

Введение

Arageli — это библиотека для точных, символьных, алгебраических вычислений. Она содержит определение таких структур, как вектора, матрицы, полиномы, целые и рациональные числа неограниченной величины и алгоритмы для решения различных задач с их использованием.

Глава 2

Длинные целые числа и обыкновенные дроби

2.1 Создание, ввод и вывод

Одной из структур, определенных в библиотеке, являются целые числа неограниченной длины. Для того, чтобы их использовать, необходимо подключить файл <arageli/big_int.hpp>.

Для их задания существует несколько конструкторов. Самый простой из них — конструктор вида

big_int()

который создает число с нулевым значением. Другой конструктор вида

big_int (**const char** **str*)

создает число по его строковому представлению[1]. Существует ряд дополнительных конструкторов, единственным параметром которых является число любого стандартного целочисленного типа. Соответственно, будет создано длинное целое число со значением, равным значению параметра.

Другим очень нужным типом данных является класс обыкновенных дробей, определенный в библиотеке как шаблонный класс[2]. Для работы с ним нужно подключить файл <arageli/rational.hpp>.

[1] Строковым представлением длинного целого числа является последовательность цифр в его десятичной записи.

[2] В этой главе будут рассматриваться примеры, где числитель и знаменатель имеют либо тип **int**, либо *big_int*

Для задания обыкновенных дробей также существует несколько конструкторов. Первые два из них

```
template<typename T = big_int>
class rational< T > {
   rational();
   rational(const char *str);
   ...
};
```

имеют точно такой же смысл, что и для длинных целых, с той лишь оговоркой, что строковым представлением для обыкновенных дробей является запись вида m/n, где m — числитель, n — знаменатель, либо просто m, если знаменатель равен единице. Как видно из объявления класса обыкновенных дробей, тип числителя и знаменателя по умолчанию — целое число неограниченной величины. Есть еще два очень удобных конструктора

```
template<typename T = big_int>
class rational< T > {
   template<typename T1>
      rational(const T1 &w);
   rational(const T &u, const T &v);
   ...
};
```

Первый из них создает дробь с числителем, равным w и знаменателем, равным единице. Второй создает дробь вида u/v.

Для целых чисел неограниченной длины и для рациональных дробей существует функция *rand*, которая задает псевдо-случайное число неограниченной длины или рациональную дробь из интервала $[0; Max]$, где Max — это единственный параметр данной функции.

Для задания целых псевдо-случайных чисел неограниченной величины в библиотеке есть еще две функции:

```
static big_int  random_with_length(size_t len)

static big_int  random_with_length_or_less(size_t len)
```

Первая возвращает число, количество бит в котором точно равно параметру *len*. Вторая функция возвращает число, количество бит в котором $\leq$ *len*.

Для ввода и вывода объектов этих типов данных можно использовать стандартные потоки ввода и вывода *std::istream* и *std::ostream* соответственно.

В случае, если строковое представление длинного целого числа задано неправильно, то в ходе выполнения программы будет сгенерировано исключение *Arageli::big_int::incorrect_string*.

Рассмотрим некоторые функции для обыкновенных дробей, подробнее о которых будет рассказано в следующем разделе. При помощи функций *numerator*() и *denominator*() можно получить доступ к числителю и знаменателю соответственно. Функция *normalize*() сокращает в случае необходимости обыкновенную дробь, а функция *is_normal*() проверяет, является ли дробь несократимой.

Рассмотрим пример:

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    //простое создание переменной
    big_int zero;
    cout << "Конструктор по умолчанию создал zero = "
        << zero << endl;

    //задаем число строкой
    big_int big_number =
        "10110011100011110000111110000011111000000";
    cout << "Really big int: " << big_number << endl;

    //ввод с клавиатуры аналогичен заданию строкой
    big_int number_from_input;
    cout << "Введите длинное целое число: ";
    cin >> number_from_input;

    //создаем рациональное число
    rational<> rat_zero;
    cout << "Для рациональных чисел zero = "
        << rat_zero << endl;
    cout << "Его числитель равен "
        << rat_zero.numerator() << endl;
    cout << "Его знаменатель равен "
        << rat_zero.denominator() << endl;

    //рациональное число можно задать строкой
    rational<> fraction = "705/32768";
```

```
    //и ввод с клавиатуры как всегда прост
    rational<> rational_number_from_input;
    cout << "Введите любое рациональное число: ";
    cin >> rational_number_from_input;

    //целые числа тоже рациональны
    rational<> rational_integer_number = number_from_input;
    cout << "Это рациональное число - целое: "
        << rational_integer_number << endl;

    //можно указать и числитель и знаменатель
    rational<> number_pi(22,7);
    cout << "Pi приближенно равно " << number_pi << endl;
    //рациональное число можно привести к действительному
    double floating_point_pi = number_pi;
    cout << "Это грубое приближение числа Pi: "
        << setprecision(14) << floating_point_pi << endl;
    //pi = 3.14159265358979323846264338327950288419 7
    rational<> pi(355,113);
    cout << "Более точно Pi равно " << pi << " ~ "
        << setprecision(14) << double(pi) << endl;

    //если изменять числитель или знаменатель, то
    //дробь может быть необходимо сократить
    fraction.numerator() += number_from_input;
    cout << "Сокращение дроби: " <<          fraction << " = ";
    fraction.normalize();
    cout << fraction << endl;

    return 0;
}
```

Результаты работы программы:

```
Конструктор по умолчанию создал zero = 0
Really big int: 10110011100011110000111110000011111100000 0
Введите длинное целое число: 9099
Для рациональных чисел zero = 0
Его числитель равен 0
Его знаменатель равен 1
Введите любое рациональное число: -916/929
Это рациональное число - целое: 9099
Pi приближенно равно 22/7
Это грубое приближение числа Pi: 3.1428571428571
Более точно Pi равно 355/113 ~ 3.141592920354
Сокращение дроби: 9804/32768 = 2451/8192
```

2.2 Элементарные операции

Для того, чтобы производить простейшие арифметические операции над объектами рассматриваемых в данной главе типов данных, можно использовать обычные знаки математических формул, такие как:

- + — сложение или унарный плюс;
- − — вычитание или унарный минус;
- * — умножение;
- / — деление; для длинных целых чисел — целая часть от деления[3];
- % — для длинных целых чисел — остаток от деления.

У каждого из этих операторов есть соответствующая составная форма, образованная вместе с оператором присваивания: +=, −=, *=, /=, %=.

Для обыкновенных дробей и длинных целых чисел определены стандартные операторы сравнения, такие как <, >, <=, >=, ==, !=. При этом сравниваться могут числа необязательно одного типа данных.

Также для длинных целых чисел есть префиксные и постфиксные операторы ++ и −−, которые имеют такой же смысл, как и, например, для чисел типа **int**. Для возведения в натуральную степень целых чисел неограниченной величины в библиотеке в файле <arageli/powerest.hpp> есть функция *power*. Дополнительно в этом файле содержатся функции *square* для возведения числа в квадрат, а также функции, прототипы которых выглядят следующим образом:

```
template<class T1, class T2, class Q, class R>
void divide(const T1 &a, const T2 &b, Q &q, R &r)

template<class T1, class T2, class Q, class R>
void prdivide(const T1 &a, const T2 &b, Q &q, R &r)
```

Оба этих метода вычисляют Q и R, т.е. соответственно целую часть и остаток от деления a на b. Остаток соответствует правилам деления для встроенных типов. Разница между этими двумя методами состоит в том, что при использовании второй функции получаемый остаток, т.е. R, всегда положителен. Таким образом, при необходимости получить сразу и целую часть, и остаток от деления лучше воспользоваться одной из этих функций, т.к. это будет быстрее, чем поочередное использование операторов / и %.

[3]При делении на нуль генерируется исключение *Arageli::divizion_by_zero*.

Рациональную дробь можно перевести в десятичную при помощи **operator double**. Для длинных целых чисел есть операторы приведения типа почти к каждому встроенному численному типу данных, такому как **int**, **double** и т.д.

При помощи функций *numerator*() и *denominator*() можно получить доступ к числителю и знаменателю соответственно для обыкновенных дробей. Функция *normalize*() сокращает в случае необходимости обыкновенную дробь, а функция *is_normal*() проверяет, является ли дробь несократимой. Функция *is_integer*() проверяет, является ли дробь целым числом. Функция *inverse*() «переворачивает» обыкновенную дробь, т.е. меняет местами числитель и знаменатель.

При работе с длинным целым числом, можно работать непосредственно и с его битами. Функция *length*() возвращает длину длинного целого числа в битах. Доступ к конкретному биту осуществляется посредством квадратных скобок. При этом можно обращаться к битам младше старшего или к старшему (с индексом $length - 1$); изменять же можно только младшие биты (т.е. длина числа меняться не должна). Используя операторы << и >> можно осуществлять побитовый сдвиг влево или вправо длинных целых чисел на указанное количество разрядов.

Проверку на четность/нечетность длинного целого числа можно выполнить посредством функций *is_even*() и *is_odd*() соответственно.

Для обыкновенных дробей и для длинных целых чисел существует ряд одинаковых базовых функций:

- *cmp* — сравнивает два числа (принимает на вход два числа; возвращает $+1$, если первое больше второго; 0, если они равны; -1, если первое меньше второго);
- *sign*() — определяет знак числа;
- *is_null*() — определяет, является ли число нулем;
- *is_unit*() — определяет, является ли число единицей;
- *is_opposite_unit*() — определяет, является ли число минус единицей;
- *abs* — возвращает абсолютное значение числа;
- *swap* — меняет два числа местами.

Рекомендуется использовать именно первые пять функций вместо явных сравнений.

Рассмотрим пример:

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    //операции с big_int осуществляются так же
    //как и со встроенными типами и, кроме того,
    //можно смешивать в одном выражении big_int,
    //rational<> и встроенные типы.
    big_int day_seconds = (big_int(60) * 60) * 24;
    cout << "Один день это " << day_seconds <<" секунд."<< endl;
    big_int year_seconds = day_seconds * 365;
    cout << "Один год это " << year_seconds <<" секунд."<< endl;
    big_int leap_year_seconds = year_seconds + day_seconds;
    cout << "Но високосный год это " << leap_year_seconds
        << " секунд." << endl;

    if((big_int(12345678987654321) % big_int(111)).is_null())
    {
        cout << "12345678987654321 делится на 111" << endl;
        cout << "Результат деления: "
            << big_int(12345678987654321) / big_int(111) << endl;
    }
    else
        cout << "12345678987654321 не делится на 111" << endl;

    big_int first_big_number  = power(big_int(23),32);
    big_int second_big_number = power(big_int(32),23);
    if(first_big_number < second_big_number)
        cout << "Мы выяснили, что 23^32 < 32^23." << endl;
    else
        if(first_big_number > second_big_number)
            cout << "Мы выяснили, что 32^23 < 23^32." << endl;
        else
            if(first_big_number == second_big_number)
                cout << "Too good to be true!" << endl;
            else
                cout << "It's very lovely!" << endl;

    cout << "2^64 = " << (big_int(1) << 64) << endl;
```

```
    //и с rational все так же просто
    rational<> sqrt_2 = "1/2";
    for(int i = 0; i < 10; i++)
        sqrt_2 = 1 / (2 + sqrt_2);
    /*а можно и так:
        (sqrt_2 += 2).inverse();
    */
    sqrt_2 += 1;
    cout << "sqrt(2) ~ " << sqrt_2 << endl;
    rational<> two = square(sqrt_2);
    if(two.is_integer())
        cout << "It couldn't be further from the truth.";
    cout << "(" << sqrt_2 << ")^2 = " << two << endl;
    cout << two << " ~ "
         << setprecision(16) << double(two) << endl;

    return 0;
}
```

Результаты работы программы:

```
Один день это 86400 секунд.
Один год это 31536000 секунд.
Но високосный год это 31622400 секунд.
12345678987654321 делится на 111
Результат деления: 111222333222111
Мы выяснили, что 32^23 < 23^32.
2^64 = 18446744073709551616
sqrt(2) ~ 19601/13860
(19601/13860)^2 = 384199201/192099600
384199201/192099600 ~ 2.000000005205633
```

2.3 Базовые целочисленные алгоритмы

В библиотеке есть большое количество функций, необходимых при работе с длинными целыми числами.

Рассмотрим некоторые из них подробнее:

- **template**<**typename** *T*>
  T intsqrt (**const** *T* &*a*);
  Находит наибольшее целое, квадрат которого не превосходит *a*;

- **template<typename** *T*>
  T inverse_mod (**const** *T* &*a*, **const** *T* &*n*);
  Находит обратное к a по модулю *n*;

- **template<typename** *T*>
  T factorial (**const** *T* &*a*);
  Вычисляет факториал от числа *a*.

Для того, чтобы использовать эти функции, необходимо подключить файл <arageli/intalg.hpp>.

В случае необходимости вычислять НОК или НОД для двух чисел, нужно использовать функции *lcm* и *gcd* соответственно. Или, для вычисления НОД, можно использовать функцию *euclidean*, которая, работая по алгоритму Евклида, как раз возвращает НОД двух чисел-параметров. Если Вы будете использовать вместо этой функции функцию *euclidean_bezout*, то кроме НОД Вы также найдете коэффициенты Безу.

Все эти функции находятся в файле <arageli/gcd.hpp>.

Для того, чтобы определить, является ли число простым/составным, в библиотеке есть методы *is_prime* и *is_composite*. Для нахождения следующего или предыдущего простого числа для заданного используйте функции *next_prime* и *prev_prime* соответственно. В том случае, если необходимо разложить число на простые множители, можно использовать функцию *factorize*, параметром которой является факторизуемое число, а результатом работы будет вектор, элементами которого и будут являться простые множители этого числа.

Все эти функции для получения результата работают по наиболее подходящему к данной ситуации алгоритму. Для получения доступа к ним нужно подключить файл <arageli/prime.hpp>.

Рассмотрим пример:

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;
```

```
int main(int argc, char *argv[])
{
    //вычислить факториал и корень очень просто:
    big_int f = factorial(big_int(16));
    cout << "16! = " << f << endl;
    big_int sf = intsqrt(f);
    cout << "Целая часть от sqrt(16!) = " << sf << endl;
    cout << sf << "^2 = " << sf*sf << endl;

    //найдем первое составное число Ферма:
    big_int pow = 1;
    big_int fermas_number;
    cout << "Числа Ферма: " << endl;
    while(is_prime(fermas_number = (big_int(1) << pow) + 1))
    {
        cout << fermas_number << " - простое" << endl;
        pow <<= 1;
    }
    cout << fermas_number << " - составное" << endl;
    //найдем его разложение на простые множители
    vector<big_int> factorization;
    factorize(fermas_number,factorization);
    cout << fermas_number << " = ";
    output_list(cout,factorization,"","","*");

    return 0;
}
```

Результаты работы программы:

```
16! = 20922789888000
Целая часть от sqrt(16!) = 4574143
4574143^2 = 20922784184449
Числа Ферма:
3 - простое
5 - простое
17 - простое
257 - простое
65537 - простое
4294967297 - составное
4294967297 = 641*6700417
```

Глава 3

Матрицы и векторы

3.1 Создание, ввод и вывод матриц и векторов

В Arageli есть возможность создавать матрицы и векторы с произвольными коэффициентами. Далее в этой главе будут рассматриваться примеры с целыми и рациональными коэффициентами, в следующей главе, посвященной полиномам, будут также рассмотрены матрицы с полиномиальными коэффициентами и полиномы с коэффициентами из матриц.

Для того чтобы использовать матрицы, необходимо в проект включить файл <arageli/matrix.hpp>, а для использования векторов файл <arageli/vector.hpp>.

Для создания матриц и векторов существует несколько конструкторов. Самые простые из них это

```
template<typename T, bool REFCNT = true>
class matrix< T, REFCNT >
  vector();
  ...
};
```

и

```
template<typename T, bool REFCNT = true>
class matrix< T, REFCNT > {
  matrix();
  ...
};
```

для векторов и матриц соответственно. Они создают объекты нулевого размера (для матриц это 0×0).

При помощи более сложных конструкторов можно создавать матрицы и векторы конкретного размера и уже инициализированные каким-либо значением (по умолчанию это нулевой коэффициент). Для каждого конструктора есть соответствующая ему функция *assign*, которая выполняет аналогичную задачу, но может быть вызвана в любой момент.

Например, следующая запись

matrix<**int**> *A*(3, 2, *diag*);

определяет целочисленную матрицу $A = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$. В данном примере последний параметр конструктора определяет тип матрицы (здесь диагональная), второй параметр — число, которое будет расположено по диагонали, а первый — количество строк и столбцов в ней.

В библиотеке существует очень удобный способ задания векторов и матриц при помощи строк (в том числе и напрямую через стандартный поток ввода *std::istream*). При этом векторы задаются в круглых скобках, где элементы разделяются запятыми; матрицы также задаются в круглых скобках, построчно, где каждая строка задается так же, как и вектор (т.е. в круглых скобках, элементы внутри разделены запятыми), а строки между собой также разделяются запятыми.

Для вывода матриц и векторов предусмотрено несколько механизмов:

1. Простой вывод вида *s*<<*A*, где *s* — стандартный поток вывода *std::ostream*, *A* - матрица или вектор;

2. Специализированная команда для более наглядного вывода матриц и векторов *output_aligned* (см. для более подробной информации документацию по этой функции);

3. Вывод в форме для дальнейшего использования в системе LaTeX для которого нужно использовать функцию *output_latex* (также для более подробной информации см. документацию по этой функции).

Размеры матриц и векторов можно определить при помощи следующих функцций:

- *ncols*() — определяет количество столбцов в матрице;
- *nrows*() — определяет количество строк в матрице;
- *size*() — определяет количество элементов в векторе и матрице (для матрицы это будет произведение количества строк на количество столбцов в ней).

Рассмотрим на простом примере, как все это сделать.

/*

Следующая программа выводит матрицу A различными способами, печатает количество столбцов и строк в ней. Затем выводит вектор b и печатает его размер.

Исходные данные: матрица с рациональными коэффициентами

$$A = \begin{pmatrix} 1/2 & 2/3 & 3/6 \\ -5/7 & 6 & 7/5 \\ 8/11 & -9/2 & 11 \end{pmatrix}$$

вектор с рациональными коэффициентами

$$b = \begin{pmatrix} 2/3 \\ -1/5 \\ 4 \end{pmatrix}$$

*/

```
#include <arageli.hpp>
using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    matrix< rational<> > A;
    vector<rational<> > b;

    A = "((1/2, 2/3, 3/6), (-5/7, 6, 7/5), (8/11, -9/2, 11))";
    b = "(2/3, -1/5, 4)";

    cout<<"\nA (operator <<) = "<<A;

    cout<<"\nA (output_aligned) = \n";
    output_aligned(cout, A, "| ", " |", " ");

    cout<<"\nA (output_latex) = \n";
    output_latex(cout, A);

    cout<<"\nCols in A = "<<A.ncols();
    cout<<"\nRows in A = "<<A.nrows();
    cout<<"\nb = "<<b;
    cout<<"\nSize of b = "<<b.size();

    return 0;
}
```

Результаты работы программы:

```
A (operator <<) = ((1/2, 2/3, 1/2), (-5/7, 6, 7/5),
(8/11, -9/2, 11))
A (output_aligned) =
|| 1/2  2/3  1/2 ||
|| -5/7  6   7/5 ||
|| 8/11 -9/2 11  ||

A (output_latex) =
$\left(\begin{tabular}{ccc}$1/2$ & $2/3$ & $1/2$ \\[3pt]
$-5/7$ & $6$ & $7/5$\\[3pt] $8/11$ & $-9/2$ & $11$
\end{tabular}\right)$
Cols in A = 3
Rows in A = 3
b = (4, -1/5, 2/3)
Size of b = 3
```

3.2 Арифметические операции с матрицами и векторами

В библиотеке используя обычные знаки математических формул можно вычислять арифметические выражения с матрицами и векторами. Также у каждого из этих операторов есть соответствующая составная форма, образованная вместе с оператором присваивания.

Обратите внимание на то, что если в результате всех операций получается вектор, то результат необходимо присвоить элементу типа вектор, а не типа матрица (для преобразования вектора к матрице можно использовать функции по вставке строк или столбцов в матрицу, рассматриваемые в следующем разделе).

В библиотеке для матриц и векторов определены стандартные операторы сравнения, такие как $<$, $>$, $<=$, $>=$, $==$, $!=$.

Опишем, как сравниваются вектора. Вектора a и b с размерами $s1$ и $s2$ соответственно находятся в отношении $a<b$, если:

1. При одновременном поэлементном просмотре этих векторов от начала к концу первая пара различающихся элементов находится под номером i, и $a[i]<b[i]$;

2. если различающихся элементов при просмотре не было найдено (т.е. пр просмотре закончился один из векторов), то $s1<s2$.

Таким образом, сравнение векторов — это лексикографическое сравнение.

Соответственно, матрицы A и B, где в матрице A содержится $m1$ строк и $n1$ столбцов, а в матрице B — $m2$ строк и $n2$ столбцов, находятся в отношении $A<B$, если

1. $m1<m2$, либо
2. $m1=m2$ и $n1<n22$, либо
3. $m1=m2$, $n1=n2$, и матрица A, развернутая по строкам, лексикографически меньше матрицы B, представленной аналогичным образом.

Также реализованы функции для покомпонентного сравнения векторов, их описание содержится в следующем разделе.

Рассмотрим пример.

/*

Следующая программа перемножает матрицы A и B, результат умножает на вектор c, после чего из полученного вектора отнимается вектор d, умноженный на число $k2$, и весь результат делится на число $k1$.

Исходные данные: матрица с рациональными коэффициентами

$$A = \begin{pmatrix} -1/2 & 3/4 \\ -2/3 & 5 \\ 1/7 & -5/2 \end{pmatrix}$$

матрица с рациональными коэффициентами

$$B = \begin{pmatrix} 3/4 & 1/6 & -7/8 \\ 5/2 & 2/5 & -9/10 \end{pmatrix}$$

вектор с рациональными коэффициентами

$$c = \begin{pmatrix} 1/4 \\ -4/15 \\ 5 \end{pmatrix}$$

вектор с рациональными коэффициентами

$$d = \begin{pmatrix} -2/3 \\ -1 \\ 4 \end{pmatrix}$$

рациональное число

$$k1 = 1/120$$

целое число

$$k2 = -2$$

*/

```
#include <arageli.hpp>
using namespace std;
using namespace Arageli;

typedef rational<> R;

int main(int argc, char *argv[])
{
    matrix< R > A, B;
    vector< R > c,d, res;
    R k1;
    int k2;

    A = "((-1/2, 3/4), (-2/3, 5), (1/7, -5/2))";
    B = "((3/4, 1/6, -7/8), (5/2, 2/5, -9/10))";

    c = "(1/4, -4/15, 5)";
    d = "(-2/3, -1, 4)";
    k1 = R(1,120);
    k2 = -2;

    res = ( (A*B)*c - d*k2 ) / k1;
    cout<<"\nResult: \n";
    output_aligned(cout, res);

    return 0;
}
```

Результаты работы программы:

```
Result:
||-7933/30||
||-20614/9||
||43721/21||
```

3.3 Операции с векторами

3.3.1 Операции с элементами векторов

Для того чтобы обратиться к конкретному элементу вектора, нужно использовать квадратные скобки, т.е. запись вида

b[*k*]

где *b* — это вектор с произвольными коэффициентами, *k* — индекс.

Обратите внимание, что нумерация элементов векторов в Arageli начинается с нуля.

Рассмотрим функции, при помощи которых можно добавлять, удалять и переставлять местами элементы векторов:

- *iterator insert* (*size_type pos*, **const** *T* &*val*)
  Эта функция вставляет в вектор элемент со значением *val* на позицию *pos*.

- *iterator insert* (*size_type pos*, *size_type n*, **const** *T* &*val*)
  Эта функция вставляет в вектор *n* элементов со значением *val* начиная с позиции *pos*.

- **void** *push_back* (**const** *T* &*val*)
  Вставляет элемент со значением *val* в конец вектора.

- **void** *push_front* (**const** *T* &*val*)
  Вставляет элемент со значением *val* в начало вектора.

- *iterator erase* (*size_type pos*)
  Эта функция удаляет элемент вектора, который находится на позиции *pos*.

- **void** *erase* (*size_type pos*, *size_type n*)
  Эта функция удаляет из вектора *n* элементов, начиная с позиции *pos*.

- **template**<**typename** *T2*>
  void *remove* (**const** *T2* &*v*)
  Удаляет из вектора все элементы со значением, равным *v*.

- **void** *swap_els* (*size_type xpos*, *size_type ypos*)
  Меняет местами элементы вектора с индексами *xpos* и *ypos*.

Доступ к элементам векторов также может осуществляться через итераторы. Дополнительную информацию об этом можно найти в документации к библиотеке.

Рассмотрим простой пример:

/*

Следующая программа вставляет в вектор a, в котором первоначально содержится только один элемент x, по одному числу в начало и в конец, потом еще два числа, начиная со второй позиции, которые равны x. После этого среднее число в векторе изменяется, а все элементы, равные x, удаляются из a. Затем первый и последний элементы вектора a меняются местами. Печатаются промежуточные и итоговый результаты.

Исходные данные: вектор нулевого размера с целыми коэффициентами $a = (3)$.

*/

```
#include <arageli.hpp>
using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    vector<int> a = "(3)";
    int x = a[0];

    cout<<"\na = "<<a;
    cout<<"\nx = "<<x;

    a.push_front(1);
    a.push_back(5);
    a.insert(1,2,x);
    cout<<"\na (after inserting) = "<<a;

    int m_index = (a.size())/2;
    a[ m_index ] = -1;
    cout<<"\na (after changing the middle element) = "<<a;

    a.remove(x);
    cout<<"\na (after removing all elements, "
      "that are equal to x) = "<<a;

    a.swap_els(0, 2);
    cout<<"\na (after swapping the first and "
      "the last elements) = "<<a;
```

```
    return 0;
}
```

Результаты работы программы:

```
a = (3)
x = 3
a (after inserting) = (1, 3, 3, 3, 5)
a (after changing the middle element) = (1, 3, -1, 3, 5)
a (after removing all elements, that are equal to x) =
(1, -1, 5)
a (after swapping the first and the last elements) = (5, -1, 1)
```

3.3.2 Покомпонентное сравнение векторов

Как уже говорилось выше, в библиотеке для векторов кроме лексикографического сравнения, реализовано также покомпонентное сравнение, причем последнее можно применять лишь в том случае, если размеры векторов совпадают.

Ниже приведены имена функций, обеспечивающих покомпонентное сравнение векторов (из названия видна связь с соответствующими функциями для скаляров):

- *each_cmp*
- *each_sign*
- *each_is_positive*
- *each_is_negative*
- *each_less*
- *each_greater*
- *each_lessequal*
- *each_greater_equal*
- *each_equal*
- *each_not_equal*.

Все эти методы возвращают вектор, каждый элемент которого — значение соответствующей функции, примененной к соответствующей паре элементов векторов-параметров (или к одному соответствующему элементу исходного вектора, если соответствующая функция для скаляров имеет только один параметр).

Например, для функции *each_cmp*, примененной к векторам a и b, результатом будет вектор c, каждый элемент которого вычислен по формуле $c[i] = cmp(a[i], b[i])$. Для функции *each_is_positive*, примененной к вектору a, результатом будет вектор c, где $c[i] = is\_positive(a)$.

Если нужно узнать, удовлетворяют ли все элементы данного вектора некоторому условию или находятся ли все соответствующие пары двух данных векторов в определенном отношении друг с другом, то следует воспользоваться одной из следующих специальных функций:

- *all_is_positive*
- *all_is_negative*
- *all_less*
- *all_greater*
- *all_less_equal*
- *all_greater_equal*
- *all_equal*
- *all_not_equal*.

Все они возвращают **true**, если условие выполняется сразу для всех пар векторов-параметров или для всех элементов одного вектора (также в зависимости от того, сколько аргументов принимает соответствующая функция для скаляров), и **false** в противном случае.

3.3.3 Функции для нахождения НОК и НОД

В библиотеке существуют операции, позволяющие находить НОК и НОД как для элементов одного вектора, так и для двух векторов (функции *gcd* и *lcm* соответственно). В том случае, когда НОК и НОД находятся для двух векторов результатом будет также вектор, т.к. НОК и НОД находятся для элементов векторов попарно. Вообще в Arageli любая бинарная операция над векторами выполняется покомпонентно.

Для выполнения этих операций необходимо подключить файл <arageli/gcd.hpp>.

Рассмотрим простой пример.

/*

Следующая программа сначала вычисляет НОД для элементов вектора a, затем вычисляет НОК для элементов вектора b, после чего вычисляет НОК и НОД поэлементно для векторов a и b. Все результаты поочередно выводятся.

Исходные данные: вектор с целыми коэффициентами

$$a = \begin{pmatrix} 4 \\ 6 \\ 16 \\ 8 \end{pmatrix}$$

вектор с целыми коэффициентами

$$b = \begin{pmatrix} 2 \\ 3 \\ 8 \\ 6 \end{pmatrix}$$

*/

```
#include <arageli.hpp>
using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    vector<int> a,b;
    a = "(4, 6, 16, 8)";
    b = "(2, 3, 8, 6)";

    cout<<"\na = "<<a;
    cout<<"\nb = "<<b;

    cout<<"\nGCD for elements in a = "<<gcd(a);
    cout<<"\nLCM for elements in b = "<<lcm(b);

    cout<<"\nGCD for a and b = "<<gcd(a,b);
    cout<<"\nLCM for a and b = "<<lcm(a,b);

    return 0;
}
```

Результаты работы программы:

```
a = (8, 16, 6, 4)
b = (6, 8, 3, 2)
GCD for elements in a = 2
LCM for elements in b = 24
GCD for a and b = (2, 3, 8, 2)
LCM for a and b = (4, 6, 16, 24)
```

3.4 Операции с матрицами

3.4.1 Операции со строками и столбцами матриц

С матрицами можно выполнять различные операции над строками, такие как вставка, удаление, умножение и деление на коэффициент (естественно соответствующий типу коэффициентов самой матрицы), перестановка строк местами, добавление к одной строке другой, умноженной на какой-то коэффициент и т.д.

Рассмотрим некоторые из этих функций подробнее:

- **void** *insert_row* (*size_type pos*, **const** *T &val*)
  Эта функция вставляет в матрицу новую строку, состоящую из элементов *val*, начиная с позиции *pos* (при этом остальные строки сдвигаются вниз).

- **template<typename** *T1*, **bool** *REFCNT1*>
  void *insert_row* (*size_type pos*,
  const *vector< T1, REFCNT1 > &vals*)
  Вставляет в матрицу новую строку, состоящую из элементов вектора *vals*, начиная с позиции *pos*.

- **void** *insert_rows* (*size_type pos*, *size_type n*, **const** *T &val*)
  Эта функция вставляет в матрицу *n* новых строк, состоящих из элементов *val*, начиная с позиции *pos*.

- **template<typename** *T1*, **bool** *REFCNT1*>
  void *insert_rows* (*size_type pos*, *size_type n*,
  const *vector< T1, REFCNT1 > &vals*)
  Вставляет в матрицу *n* новых строк, каждая из которых состоит из элементов вектора *vals*, начиная с позиции *pos*.

- **void** *erase_row* (*size_type pos*)
  Удаляет в матрице строку, которая находится на позиции *pos2*.

- **void** *erase_rows* (*size_type pos*, *size_type n*)
  Удаляет в матрице *n* строк, начиная с позиции *pos*.

- **void** *swap_rows* (*size_type xpos*, *size_type ypos*)
  Меняет две строки, которые находятся на позициях *xpos* и *ypos*, местами.

- **template**<**typename** *T1*>
  void *mult_row* (*size_type i*, **const** *T1 &x*)
  Умножает *i*-ю строку на *x*.

- **template**<**typename** *T1*>
  void *div_row* (*size_type i*, **const** *T1 &x*)
  Делит *i*-ю строку на *x*.

- **void** *add_rows* (*size_type dstpos*, *size_type srcpos*)
  Прибавляет к строке с индексом *dstpos* строку с индексом *srcpos*.

- **void** *sub_rows* (*size_type dstpos*, *size_type srcpos*)
  Вычитает из строки с индексом *dstpos* строку с индексом *srcpos*.

- **template**<**typename** *T2*>
  void *addmult_rows* (*size_type dstpos*, *size_type srcpos*,
  const *T2 &y*)
  Прибавляет к строке с индексом *dstpos* строку с индексом *srcpos*, умноженную на *y*.

- *vector*<*T*, **true**> *copy_row* (*size_type i*) **const**
  Возвращает копию строки с индексом *i*.

Все операции, доступные для строк, доступны также и для столбцов, причем прототипы этих функций выглядят одинаково, нужно лишь заменить в имени функции *row* на *col*.

Стоит отметить, что нумерация строк и столбцов для матриц в Arageli начинается с нуля.

Для обращения к конкретному элементу матрицы используются круглые скобки, т.е. запись вида

$$A(i,j)$$

где A — это матрица с произвольными коэффициентами, i, j — номера строки и столбца соответственно, на пересечении которых расположен желаемый элемент.

Обратите внимание на ситуацию, когда производятся операции со строкой (столбцом) матрицы и элементом из той же самой строки (столбца). Например, такое может быть при делении столбца на один из элементов этого столбца (см. следующий пример). В этом случае вместо самого элемента надо указывать то, что возвращает функция *safe_reference*, единственным параметром которой должен являться сам этот элемент. Похожая ситуация обстоит и с полиномами.

Рассмотрим пример на применение части выше описанных функций.

/*

Следующая программа выводит на экран матрицу A, затем первую и вторую строки в ней. После чего к первой строке прибавляется вторая, умноженная на k, и результат выводится на экран. Затем первый столбец матрицы делится на элемент матрицы, расположенный в этом столбце во второй строчке, печатается результат. После чего, начиная со второго столбца, удаляются два столбца в матрице, печатается результат. Затем на место второго столбца вставляется вектор b, печатается результат. И наконец итоговый результат — в получившейся матрице переставляются местами первая и третья строки.

Исходные данные: матрица с рациональными коэффициентами

$$A = \begin{pmatrix} -2/5 & -1/5 & 3/5 & 1/2 \\ 2/5 & -1/4 & 2/6 & 1/3 \\ -5/2 & 5/6 & -6/7 & 1/4 \end{pmatrix}$$

вектор с рациональными коэффициентами

$$b = \begin{pmatrix} -2/3 \\ 5/2 \\ -1/6 \end{pmatrix}$$

целое число

$$k = 5$$

*/

```
#include <arageli.hpp>
using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    matrix< rational<> > A;
    vector<rational<> > b;
```

```
    int k;

    A = "((-2/5, -1/5, 3/5, 1/2), (2/5, -1/4, 2/6, 1/3),"
    "(-5/2, 5/6, -6/7, 1/4))";
    b = "(-2/3, 5/2, -1/6)";
    k = 5;

    cout<<"\nA = \n";
    output_aligned(cout, A);

    cout<<"\nThe first row in A = "<<A.copy_row(0);
    cout<<"\nThe second row in A = "<<A.copy_row(1);

    A.addmult_rows(0, 1, k);
    cout<<"\n\nA[0] = A[1] * k (rows)";
    cout<<"\nResult = \n";
    output_aligned(cout, A);

    A.div_col(1, safe_reference(A(2, 1)));
    cout<<"\n\nA[1] = A[1] / A(2,1) (cols)";
    cout<<"\nResult = \n";
    output_aligned(cout, A);

    A.erase_cols(2, 2);
    cout<<"\n\nA after erasing two columns"
                          "from the second one = \n";
    output_aligned(cout, A);

    A.insert_col(2, b);
    cout<<"\n\nVector b = "<<b;
    cout<<"\nA after inserting b = \n";
    output_aligned(cout, A);

    A.swap_rows(0,2);
    cout<<"\n\nA after swapping the first and"
                               "the third rows = \n";
    output_aligned(cout, A);

    return 0;
}
```

Результаты работы программы:

```
A =
||-2/5 -1/5 3/5  1/2||
||2/5  -1/4 1/3  1/3||
||-5/2 5/6  -6/7 1/4||
```

```
The first row in A = (-2/5, -1/5, 3/5, 1/2)
The second row in A = (2/5, -1/4, 1/3, 1/3)

A[0] = A[1] * k (rows)
Result =
||8/5  -29/20 34/15 13/6||
||2/5   -1/4   1/3  1/3 ||
||-5/2  5/6   -6/7  1/4 ||


A[1] = A[1] / A(2,1) (cols)
Result =
||8/5  -87/50 34/15 13/6||
||2/5  -3/10   1/3  1/3 ||
||-5/2   1    -6/7  1/4 ||


A after erasing two first columns =
||8/5  -87/50||
||2/5  -3/10 ||
||-5/2   1   ||


Vector b = (-1/6, 5/2, -2/3, )
A after inserting b =
||8/5  -87/50 -2/3||
||2/5  -3/10  5/2 ||
||-5/2   1    -1/6||


A after swapping the first and the third rows =
||-5/2   1    -1/6||
||2/5  -3/10  5/2 ||
||8/5  -87/50 -2/3||
```

3.4.2 Выполнение базовых операций с матрицами

В Arageli существуют специальные функции для выполнений над матрицами таких базовых операций, как вычисление определителя, нахождение обратной матрицы и т.д.

Также есть специальные функции-члены, позволяющие определить

вид матрицы:

- *is_empty*() — определяет, является ли матрица пустой, т.е. имеет ли она нулевые размеры;
- *is_null*() — определяет, является ли матрица нулевой;
- *is_unit*() — определяет, является ли матрица единичной;
- *is_opposite_unit*() — определяет, является ли матрица отрицательной единичной матрицей;
- *is_squarte*() — определяет, является ли матрица квадратной;
- *swap* — меняет две матрицы местами;

Кроме функции *inverse*, которая находит обратную к данной матрицу, есть также функция *each_inverse*, результатом работы которой является матрица, составленная из обратных для первоначальной матрицы элементов.

Для того чтобы вычислить определитель у матрицы с целочисленными коэффициентами, необходимо вместо обычной функции *det* использовать функцию *det_int*. При вычислении ранга матрицы ситуация аналогичная (вместо функции *rank* нужно использовать *rank_int*). Для использования всех этих четырех функций необходимо подключить файл <arageli/gauss.hpp>.

Рассмотрим их применение на простом примере.

/*

Следующая программа определяет, является ли матрица A квадратной, предварительно напечатав ее. Если да, то находит и выводит d — определитель этой матрицы. Если он не равен 0, *то находит и выводит на экран обратную к данной матрицу A_inv и осуществляет проверку результата, т.е. является ли произведение матриц A и A_inv единичной матрицей.*

Исходные данные: матрица с рациональными коэффициентами

$$A = \begin{pmatrix} 2/3 & -3/5 & 1 \\ -4/7 & 5/8 & 1/12 \\ -3 & 5 & -6/7 \end{pmatrix}$$

*/

```
#include <arageli.hpp>
using namespace std;
using namespace Arageli;
```

```
int main(int argc, char *argv[])
{
    matrix< rational<> > A, A_inv;
    rational<> d = 0;

    A = "((2/3, -3/5, 1) , (-4/7, 5/8, 1/12) , (-3, 5, -6/7))";

    cout<<"\nA = "<<A;

    if (A.is_squarte()) {
        d = det(A);
        cout<<"\ndet(A) = "<<d;
    }

    if (d != 0) {
        A_inv = inverse(A);

        cout<<"\nInverted A = \n";
        output_aligned(cout, A_inv, "|| ", " ||", " ");
        cout<<"\nThe result is "<<boolalpha<<
                                    (A*A_inv).is_unit();
    }

    return 0;
}
```

Результаты работы программы:

```
A = ((2/3, -3/5, 1), (-4/7, 5/8, 1/12), (-3, 5, -6/7))
det(A) = -4139/3528
Inverted A =
|| 3360/4139 -79128/20695 11907/20695 ||
|| 2610/4139 -8568/4139   2212/4139   ||
|| 3465/4139 27048/20695  -1302/20695 ||
The result is true
```

Заметим, что если матрица вырожденная, то при вызове функции *inverse* генерируется исключение *Arageli::matrix_is_singular*.

С точки зрения скорости выполнения данный пример составлен не наилучшим способом, т.к. и при вычислении определителя, и при нахождении обратной матрицы вызывается одна и та же функция *rref*. Эту функцию вызывает довольно много методов, что, безусловно, следует учитывать.

Рассмотрим тот же пример, но с использованием этой функции:

```
#include <arageli.hpp>
using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{

    matrix< rational<> > A, A_inv, B;
    vector< rational<> > basis;
    rational<> d = 0;

    A = "((2/3, -3/5, 1) , (-4/7, 5/8, 1/12) , (-3, 5, -6/7))";

    cout<<"\nA = "<<A;

    rref (A, B, A_inv, basis, d);

    cout<<"\ndet(A) = "<<d<<"\n";
    output_aligned(cout, A_inv, "| | ", " | |", " ");
    cout<<"\nThe result is "<<boolalpha<<
                                    (A*A_inv).is_unit();

    return 0;
}
```

Результаты работы этой программы полностью совпадают с результатами работы программы, текст которой приведен выше.

Для целочисленных матриц необходимо вместо функции *rref* использовать функцию *rref_int*.

3.5 Решение системы линейных уравнений

Для того чтобы решить квадратную систему линейных уравнений, в библиотеке есть функция *solve_linsys*.

Если матрица вырожденная, то при вызове функции *solve_linsys* генерируется исключение *Arageli::matrix_is_singular*.

Для того чтобы использовать эту функцию, необходимо подключить файл <arageli/gauss.hpp>.

```
/*
```

Следующая программа решает систему линейных уравнений вида $A * x = b$ и выводит результат — вектор x. Если матрица A является вырожденной матрицей, то программа сообщает об этом.

Исходные данные: матрица с рациональными коэффициентами

$$A = \begin{pmatrix} -1/2 & 2/3 & 3/6 \\ 5/7 & -6 & 7/5 \\ -8/11 & 9/2 & -11 \end{pmatrix}$$

вектор с рациональными коэффициентами

$$b = \begin{pmatrix} 2/3 \\ -1/5 \\ 4 \end{pmatrix}$$

*/

```
#include <arageli.hpp>
using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    matrix< rational<> > A;
    vector<rational<> > b,x;

    A = "((-1/2, 2/3, 3/6), (5/7, -6, 7/5), (-8/11, 9/2, -11))";
    b = "(2/3, -1/5, 4)";

    cout<<"A = \n";
    output_aligned(cout, A);
    cout<<"\nb = \n";
    output_aligned(cout, b);

    try {
        x = solve_linsys(A, b);

        cout<<"\n\nx = \n";
        output_aligned(cout, x);
        cout<<"\nThe result is "<<std::boolalpha<<(A*x == b);
    }

    catch(matrix_is_singular) {
        cout<<"\n\nError! Matrix is singular!";
    }

    return 0;
}
```

Результаты работы программы:

```
A =
||-1/2  2/3 1/2||
||5/7   -6  7/5||
||-8/11 9/2 -11||

b =
||4||
||-1/5||
||2/3 ||

x =
||-41469/119498||
||-17591/59749 ||
||-247709/119498||
The result is true
```

Глава 4

Разреженные полиномы

Библиотека предоставляет широкие возможности для работы с разреженными полиномами. Следующие примеры демонстрируют различные варианты их использования и возможности библиотеки по работе с ними.

4.1 Способы создания полиномов в Arageli

Разреженный полином в Arageli это шаблон класса *sparse_polynom*, определённый в <arageli/sparse_polynom.hpp>, который представляет полином в виде списка ненулевых мономов. Параметры этого шаблона определяют типы коэффициентов (*Coefficient*) и степеней (*Degree*) полинома:

```
template<typename Coefficient,
        typename Degree = int,
        bool REFERENCE_COUNTER = true>
class sparse_polynom;
```

Далее мы будем задавать только первый параметр *sparse_polynom*, оставляя два последних со значениями по умолчанию. Например, определения

```
sparse_polynom<int> a;
sparse_polynom<rational<> > b;
```

вводят две переменные *a* и *b*, как полиномы с целыми и рациональными коэффициентами соответственно.

Для полиномов в Arageli реализован ряд конструкторов, обеспечивающий различные механизмы реализации:

1. *sparse_polynom*();
   — создает нулевой (не содержащий ни одного монома) полином;

2. *sparse_polynom*(**const char*** *str*);
   — инициализирует полином его символьной записью[1];

3. *sparse_polynom*(**const** *Coefficient*& *x*);
   — создает полином, состоящий из одного монома с нулевым показателем степени;

4. *sparse_polynom*(**const** *Coefficient*& *x*, **const** *Degree*& *p*);
   — создает полином, состоящий из одного монома;

5. **template** <**typename** *Coefficient1*, **typename** *Degree1*>
   sparse_polynom(**const** *monom*<*Coefficient1*, *Degree1*>& *x*);
   — создает полином, равный заданному моному;

6. **template** <**typename** *Coefficient1*, **typename** *Degree1*,
   bool *REFCNT1*>
   sparse_polynom
   (**const** *sparse_polynom*<*Coefficient1*,*Degree1*,*REFCNT1*>& *x*);
   — создает полином как копию другого полинома, при необходимости осуществляя приведение типов коэффициентов и степеней к собственным;

Эти конструкторы обеспечивают функциональность, достаточную для создания и инициализации полиномов любым корректным значением. Следующий пример демонстрирует их использование в программе:

Листинг *Sparse_Polynom_Creating.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    //простое создание полинома
    //с целочисленными коэффициентами:
    sparse_polynom<int> S = "2*x^2+5*x-7+3*x";
    cout << "Полином с целочисленными коэффициентами S = "<< S
        << endl << endl;
```

[1]Правила построения символьной записи полинома описаны в следующем разделе

```
//создание полинома
//с коэффициентами неограниченного размера:
sparse_polynom<big_int>
    B = "123456789101112131415161718192 0*x^777"
          "+112233445566778899";
cout << "Полином с коэффициентами типа big_int B = "<< B
     << endl << endl;

//создание полинома с вещественными коеффициентами:
sparse_polynom<double>
    D = "1.12345*x-1e25*x^2+1234.5678*x^3-0.000002334";
cout << "Полином с вещественными коэффициентами D = "<< D
     << endl << endl;

//создание полинома с коэфициентами рационального типа:
sparse_polynom<rational<> >
    R = "1234/56781*x^321+7/9*x+3*x^2-4/256";
cout << "Полином с рациональными коэффициентами R = "<< R
     << endl << endl;

//возможно создание полинома с матричными коэффициентами:
//обратите внимание на дополнительные скобки;
//правила записи матричных элементов
//описаны в предыдущем разделе
sparse_polynom<matrix<int> > M =
    "(((1,2),(3,4))*x^55-((1,2),(4,5))*x+((5,-1),(4,0)))";
cout << "Полином с матричными коэффициентами M = "<< M
     << endl << endl;

//построение полинома из отдельных мономов:
big_int num = 1, den = 1;
int degree = 0;
sparse_polynom<rational<> > F;
for(int i = 0; i < 6; i++)
{
    F += sparse_polynom<rational<> >::
         monom(rational<>(num,den),degree);
    degree++;
    num += den;
    den += num;
}
cout << "Полином, построенный из отдельных мономов F = "
     << F << endl << endl;
```

```
    //преобразование полиномов с коэффициентами разных типов:
    //конструктор приведения типа гарантирует
    //нормальную форму представления полинома
    sparse_polynom<big_int> BD = D;
    cout << "Полином, полученный преобразованием "
        "из другого полинома BD = "<< BD << endl << endl;
    //при преобразовании рациональных чисел к числам
    //с плавающей запятой возможна потеря точности
    sparse_polynom<double> FR = R;
    cout << "Полином, полученный преобразованием "
        "из другого полинома FR = "<< FR << endl << endl;

    return 0;
}
```

Результат работы программы листинга *Sparse_Polynom_Creating.cpp* Демонстрирует создание *sparse_polynom* из встроенных типов и типов Arageli.

```
Полином с целочисленными коэффициентами S = 2*x^2+8*x-7

Полином с коэффициентами типа big_int
B = 12345678910111213141516171819 20*x^777+112233445566778899

Полином с вещественными коэффициентами
D = 1234.57*x^3-1e+025*x^2+1.12345*x-2.334e-006

Полином с рациональными коэффициентами
R = 1234/56781*x^321+3*x^2+7/9*x-1/64

Полином с матричными коэффициентами
M = ((1, 2), (3, 4))*x^55+((-1, -2), (-4, -5))*x+
((5, -1), (4, 0))

Полином, построенный из отдельных мономов
F = 89/144*x^5+34/55*x^4+13/21*x^3+5/8*x^2+2/3*x+1

Полином, полученный преобразованием из другого полинома
BD = 1234*x^3-1000000000000000905969664*x^2+x

Полином, полученный преобразованием из другого полинома
FR = 0.0217326*x^321+3*x^2+0.777778*x-0.015625
```

4.2 Ввод и вывод полиномов в Arageli

В Arageli предусмотрены встроеные средства для консольного ввода/вывода в стандартные потоки *cin* и *cout*, осуществляемого операторами >> и << соответственно. Полиномы записываются в максимально приближенной к математической форме записи. Правила записи коэффициентов полинома и показателей степени соответствуют формату, который по умолчанию использует система ввода/вывода для соответствующих типов. Кроме рассмотренных в этом разделе в качестве примеров допустимого ввода можно рассматривать все строковые инициализации в коде примеров, т.к. конструкторы и система ввода/вывода используют общий механизм построения полинома по строковой записи.

Для корректного создания полиномов необходимо придерживаться следующих правил:

- полином должен записываться слитно (без пробелов);
- переменная обозначается символом *x*;
- степень отделяется от переменной символом ^;
- символ * между коэффициентом и переменной обязателен;
- знаки + и − не требуют дополнительных скобок;
- порядок мономов в записи произволен;
- в записи одного полинома могут быть мономы с одинаковыми показателями степени;
- коэффициенты записываются согласно правилам принятым в *C++*, а в случае, если это типы Arageli, записываются согласно правилам ввода этих типов принятым в библиотеке (узнать о них можно в соответствующих разделах данного документа)
- Необходимо обратить внимание (см. первый пример) на тот аспект, что нужно брать запись полинома в скобки, если первый коэффициент начинается в скобках.

Таким образом, привычной математической записи x^3+3x^2+3x+1 будет соответствовать строка: *x*^3+3**x*^2+3**x*+1.

Полином в библиотеке хранится в нормальной форме, и при создании из строкового представления он всегда приводится к ней. Т.е.

- мономы упорядочиваются по степеням;
- приводятся подобные мономы;
- исключаются нулевые мономы;

Следующий пример еще раз поясняет особенности ввода:

Листинг *Sparse_Polynom_io.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    sparse_polynom<int> S;//создаем полином
    cout << "Пожалуйста, введите полином "
           "с целочисленными коэффициентами: S = ";

    cin >> S;//ввод полинома с клавиатуры
    cout << "S был успешно прочитан" << endl;
    //на выводе видим полином, записанный в нормальной форме
    cout << "Стандартный вывод полинома: S = " << S << endl;

    return 0;
}
```

Результат работы программы листинга *Sparse_Polynom_io.cpp*

Демонстрирует использование операторов ввода/вывода для полиномов.

```
Пожалуйста, введите полином с целочисленными коэффициентами:
S = 5*x^2-7*x^6+5+x^8-3*x^2+0*x
S был успешно прочитан
Стандартный вывод полинома: S = x^8-7*x^6+2*x^2+5
```

4.3 Арифметические операции

В основе большинства алгоритмов, использующих полиномы, лежат простые арифметические операции. В Arageli эти операции записываются при помощи принятых в *C++* символов:

- $+$ (сложение),
- $-$ (вычитание),

- * (умножение),
- / (деление нацело),
- % (остаток от деления).

Каждая операция имеет парную, совмещенную с присваиванием (например +=). Кроме операций между двумя полиномами, возможны операции, включающие одним из аргументов моном или константу (имеющие типы соответствующие праметрам шаблона данного полинома). Допустимыми являются любые разумные с точки зрения пользователя библиотеки выражения, не содержащие деления на нуль. Поведение программы при попытке деления на нулевой элемент не определено и полностью зависит от типа коэффициентов.

Листинг *Sparse_Polynom_Operations.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    //создание полиномов
    sparse_polynom<rational<> > S = "2/5*x^2+5/7*x-7/8";
    sparse_polynom<rational<> > V = "20/53*x^3-1/9*x-1/8";
    sparse_polynom<rational<> > Q = "1/15*x^4+5/7*x^3+7*x";
    cout << "Исходный sparse_polynom S = " << S << endl;
    cout << "Исходный sparse_polynom V = " << V << endl;
    cout << "Исходный sparse_polynom Q = " << Q << endl << endl;

    //арифметические операции с полиномами:
    cout << "Операции с полиномами использовать очень легко: "
       "Q + S * V = " << Q + S * V << endl << endl;
    cout << "Использование операций деления: Q = (" << Q / S
      << ") * (" << S << ") + ("
      << Q % S << ")" << endl << endl;

    typedef sparse_polynom<rational<> >::monom ratmonom;
    // обратите внимание: нет гарантии, что
    // sparse_polynom<T>::monom
    // и monom<T> это один и тот же тип данных
    // (в нашем случае T = rational<> )
```

```
        //арифметические операции с мономами и константами
        //также осуществляются по правилам алгебры:
        cout << "x = 0 - корень полинома Q. Разделим Q на x: Q = "
            << (Q /= ratmonom(1,1)) << endl << endl;
        cout << "Приведем ведущие коэффициенты в V и Q к 1:" << endl;
        cout << "V = " << (V /= rational<>(20,53)) << endl;
        cout << "Q = " << (Q /= rational<>(1,15)) << endl << endl;
        cout << "Теперь Q - V = " << Q - V << endl;

        return 0;
    }
```

Результат работы программы листинга *Sparse_Polynom_Operations.cpp*

Демонстрирует использование арифметических операций с полиномами.

```
Исходный sparse_polynom S = 2/5*x^2+5/7*x-7/8
Исходный sparse_polynom V = 20/53*x^3-1/9*x-1/8
Исходный sparse_polynom Q = 1/15*x^4+5/7*x^3+7*x

Операции с полиномами использовать очень легко: Q + S * V =
8/53*x^5+1871/5565*x^4+11341/33390*x^3-163/1260*x^2+883/126*x
+7/64

Использование операций деления: Q = (1/6*x^2+125/84*x-3595/
1568) * (2/5*x^2+5/7*x-7/8) + (10228/1029*x-3595/1792)

x = 0 - корень полинома Q. Разделим Q на x:
Q = 1/15*x^3+5/7*x^2+7

Приведем ведущие коэффициенты в V и Q к 1:
V = x^3-53/180*x-53/160
Q = x^3+75/7*x^2+105

Теперь Q - V = 75/7*x^2+53/180*x+16853/160
```

4.4 Получение основных параметров полиномов

Основные характеристики и свойства полиномов можно получить простым вызовом соответствующего метода (или аналогичной функцией Arageli).

Метод **bool** *is_normal*() проверяет, хранится ли полином в нормальной форме или она нарушена. Ненормальное состояние для полинома сделано для того, что бы можно было свободно манипулировать мономами в полиноме, производя различные символьные операции, например, дифференцирование и интегрирование. В этом случае полином используется просто как список мономов и остальные операции для него не доступны. Поэтому для осуществления вычислений с полиномами необходимо приводить их к нормальной форме. Это делает метод **void** *normalize*(), но нет необходимости вызывать его, если не осуществлялось прямое изменение представления полинома, т.к. все методы и функции Arageli, предназначенные для работы с полиномами, не нарушают их нормальную форму.

Для проверки простых условий предусмотрены специальные методы. С точки зрения производительности предпочтительно использование этих методов, а не явная запись условий.

Метод *is_null*() проверяет полином на равенство 0 т.е. *p.is_null*() означает $p = 0$.
Метод *is_x*(), проверяет равенство x т.е. *p.is_x*() означает $p =$ `"x"`.
Метод *is_const*() возвращает **true**, если полином не содержит мономов с x.

Существует возможность быстрого доступа к ведущему моному. Метод *degree*() возвращает степень полинома; *leading_coef* () позволяет получить ведущий коэффициент, а *leading_monom*() – непосредственно сам моном.

Метод *size*() возвращает количество мономов в полиноме.

Метод *subs*(*Coefficient*& *x*) служит для вычисления значения полинома в точке.

Данный пример демонстрирует использование описанных выше методов:

Листинг *Sparse_Polynom_Metrics.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    sparse_polynom<rational<> > S =
        "2/55*x^55+5/567*x^28-56/997*x^5+1/18122005*x^2+567";
    cout << "Исходный sparse_polynom is S = " << S
         << endl << endl;
```

```
//проверим является ли полином,
//приведенным к нормальной форме
if(S.is_normal())
    cout << "S приведен к нормальной форме" << endl;
else
{
    cout << "S хранитсе не в нормальной форме!!!!" << endl;
    //необходима нормализация
    //для корректной работы большинства функций
    S.normalize();
}

//проверка полинома на равенство нулю
if(S.is_null())
    cout << "S = 0 !!!!!!" << endl;
else
    cout << "S - не нулевой полином" << endl;

//проверка на равенство иксу (S == "x")
if(S.is_x())
    cout << "S = x !!!???" << endl;
else
    cout << "S не является иксом" << endl;

//проверка: состоит ли полином
//только из свободного члена (S == const)
if(S.is_const())
    cout << "S - просто рациональное число" << endl;
else
    cout << "S не является обычным числом" << endl;

//получение старшего коэффициента (2 способа)
cout << "Старший коэффициент S равен "
     << S.leading_coef() << " = "
     << S.leading_monom().coef() << endl;

//получение степени полинома (2 способа)
cout << "Степень полинома S равна "
     << S.degree() << " = "
     << S.leading_monom().degree() << endl;

//вычисление значения полинома в точке
cout << "Значение S в точке x = 1/2: S(1/2) = "
    << S.subs(rational<>(1,2)) << endl;
cout << "Значение S в точке x = 0: S(0) = "
    << S.subs(0) << endl;
```

```
    return 0;
}
```

Результат работы программы листинга *Sparse_Polynom_Metrics.cpp*

Демонстрирует методы, предназначенные для работы основными характеристиками полинома.

```
Исходный sparse_polynom is S =
2/55*x^55+5/567*x^28-56/997*x^5+1/18122005*x^2+567

S приведен к нормальной форме
S - не нулевой полином
S не является иксом
S не является обычным числом
Старший коэффициент S равен 2/55 = 2/55
Степень полинома S равна 55 = 55
Значение S в точке x = 1/2: S(1/2) =
1046371599110366637744052777965 53/184545826870304546417100718080
Значение S в точке x = 0: S(0) = 567
```

4.5 Доступ к представлению полинома

Часто бывает необходимо обработать полином не целиком, а поэлементно. Arageli позволяет свободно манипулировать отдельными мономами в представлении полинома. Необходимо лишь приводить полином к нормальной форме при помощи метода *normalize*(). Для полиномов реализован механизм итераторов аналогичный итераторам *STL*, представляющий полином как линейный список мономов. Существует 3 вида итераторов, каждый в двух формах (константной и неконстантной). Эти итераторы позволяют работать со списком мономов и отдельно со списками коэффициентов и степеней. Для добавления в полином или исключения из него мономов служат методы

```
monom_iterator insert(monom_iterator pos,
                      const Arageli::monom<F1, I1>& x);
    //вставляет моном перед позицией, указанной итератором

monom_iterator erase(monom_iterator pos);
    //удаляет моном в указанной итератором позиции
```

Следующий пример показывает как можно использовать этот механизм:

Листинг *Sparse_Polynom_Tracing.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    typedef sparse_polynom<big_int> poly;
    //введем короткие имена
    //для удобства использования итераторов
    typedef poly::coef_iterator coefs;
    typedef poly::degree_iterator degrees;
    typedef poly::monom_const_iterator monoms;

    poly S =
        "213*x^3443+532*x^4432-744*x^44-4235*x^15+292*x+34254";
    cout << "Исходный полином S = " << S << endl << endl;

    //поиск наименьшего коэффициента полинома:
    big_int min_coeff = S.leading_coef();
    for(coefs ci = S.coefs_begin(), cj = S.coefs_end();
                                        ci != cj; ++ci)
        if(min_coeff > *ci) min_coeff = *ci;
    cout << "Минимальный коэффициент S равен " << min_coeff
        << endl << endl;

    //вычислим сумму всех степеней полинома:
    //итераторы предполагают использование
    //обобщенных алгоритмов,
    //поэтому выгодно использовать алгоритм из STL
    int dsum =
        std::accumulate(S.degrees_begin(), S.degrees_end(), 0);
    cout << "Сумма степеней всех мономов полинома S составляет "
        << dsum << endl << endl;

    //выберем из полинома все мономы с нечетными степенями
    poly oddS;
    for(monoms mi = S.monoms_begin(), mj = S.monoms_end();
                                        mi != mj; ++mi)
        if(is_odd(mi->degree()))
            oddS.insert(oddS.monoms_end(), *mi);
    cout << "Полином только с нечетными показателями: oddS = "
        << oddS << endl << endl;
```

```
    //представим степени, как числа по модулю 5
    for(degrees di = S.degrees_begin(), dj = S.degrees_end();
                                          di != dj; ++di)
        *di %= 5;
    //у нас могли появиться мономы с одинаковыми степенями
    //необходимо восстановить нормальную форму представления:
    S.normalize();
    cout << "S = " << S << endl << endl;

    return 0;
}
```

Результат работы программы листинга *Sparse_Polynom_Tracing.cpp*

Демонстрирует использование итераторов для объектов *sparse_polynom*.

```
Исходный полином
S = 532*x^4432+213*x^3443-744*x^44-4235*x^15+292*x+34254

Минимальный коэффициент S равен -4235

Сумма степеней всех мономов полинома S составляет 7935

Полином только с нечетными показателями:
oddS = 213*x^3443-4235*x^15+292*x

S = -744*x^4+213*x^3+532*x^2+292*x+30019
```

4.6 Элементарные операции

Алгоритмы, работающие с полиномами, часто требуют выполнения таких элементарных операций как вычисление противоположного, перестановка двух полиномов местами, сравнение. Для таких операций Arageli предоставляет общий интерфейс для всех типов, и полиномы не являются исключением.

```
int cmp(sparse_polynom<T1> first, sparse_polynom<T2> second);
    //осуществляет сравнение полиномов
void swap(sparse_polynom<T> first, sparse_polynom<T> second);
    //выполняет перестановку двух полиномов местами
sparse_polynom<T> opposite(sparse_polynom<T> p);
    //вычисляет противоположный полином
```

Данный пример демонстрирует простоту их использования:

Листинг *Sparse_Polynom_Functions1.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    sparse_polynom<rational<> >
        P = "1/2*x^8-47/12*x^7-359/84*x^5+349/126*x^4+94/9*x^3+"
                                            "55/14*x^2+10*x",
        Q = "-99/13*x^7+834/13*x^6-141/4*x^5+22171/390*x^4-"
                                          "5699/180*x^3+2*x";
    cout << "Исходный sparse_polynom P = " << P << endl;
    cout << "Исходный sparse_polynom Q = " << Q << endl << endl;

    //функция сравнения полиномов:
    //осуществляет лексикографическое сравнение
    //возвращает +1, -1 или 0 в случае равенства
    int P_vs_Q = cmp(P,Q);
    if(P_vs_Q < 0)
        cout << "P меньше чем Q" << endl;
    if(P_vs_Q > 0)
        cout << "P больше чем Q" << endl;
    if(P_vs_Q == 0)
        cout << "P равен Q" << endl;
    //при сравнении полинома с собой будет равенство:
    int P_vs_P = cmp(P,P);
    if(P_vs_P < 0)
        cout << "P меньше чем P" << endl;
    if(P_vs_P > 0)
        cout << "P больше чем P" << endl;
    if(P_vs_P == 0)
        cout << "P равен P" << endl;
    cout << endl;
```

```
        //для быстрого обмена местами двух полиномов используется
        //функция swap; ее использование позволяет обойтись
        //без создания дополнительной переменной и использует
        //внутренние механизмы библиотеки для минимизации затрат
        //на перемещение данных
        cout << "До перестановки P и Q:" << endl
            << "P = " << P << endl
            << "Q = " << Q << endl << endl;
        swap(P,Q);
        cout << "После перестановки P и Q местами:" << endl
            << "P = " << P << endl
            << "Q = " << Q << endl << endl;

        //получение противоположного полинома:
        //(возвращает полином равный -P )
        cout << "Противоположный для P полином: "
            << opposite(P) << endl;

        return 0;
    }
```

Результат работы программы листинга *Sparse_Polynom_Functions1.cpp*

Демонстрирует применение функций *cmp*, *swap*, *opposite* к полиномам.

```
Исходный sparse_polynom
P = 1/2*x^8-47/12*x^7-359/84*x^5+349/126*x^4+94/9*x^3
+55/14*x^2+10*x
Исходный sparse_polynom Q =
-99/13*x^7+834/13*x^6-141/4*x^5+22171/390*x^4-5699/180*x^3+2*x

P больше чем Q
P равен P

До перестановки P и Q:
P = 1/2*x^8-47/12*x^7-359/84*x^5+349/126*x^4+94/9*x^3
+55/14*x^2+10*x
Q =
-99/13*x^7+834/13*x^6-141/4*x^5+22171/390*x^4-5699/180*x^3+2*x

После перестановки P и Q местами:
P =
-99/13*x^7+834/13*x^6-141/4*x^5+22171/390*x^4-5699/180*x^3+2*x
Q = 1/2*x^8-47/12*x^7-359/84*x^5+349/126*x^4+94/9*x^3
+55/14*x^2+10*x
```

```
Противоположный для P полином:
99/13*x^7-834/13*x^6+141/4*x^5-22171/390*x^4+5699/180*x^3-2*x
```

4.7 Базовые агоритмы

Arageli реализует базовые алгоритмы алгебры. В отношении полиномов имеют особое значение операция символьного дифференцирования (*diff*), вычисление наибольшего общего делителя (*gcd*), поиск наименьшего общего кратного (*lcm*) и, как их обобщение, поиск коэффициентов Безу для двух полиномов (*euclidean_bezout*). Кроме того, функция *is_coprime* позволяет проверить полиномы на взаимную простоту.
Следующий пример показывает их применение к полиномам:

Листинг *Sparse_Polynom_Functions2.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    sparse_polynom<rational<> >
        P = "x^6+x^4-x^2-1",
        Q = "x^3-2*x^2-x+2";
    cout << "Исходный sparse_polynom P = " << P << endl;
    cout << "Исходный sparse_polynom Q = " << Q << endl << endl;

    //можно проверить, являются ли полиномы взаимно-простыми:
    //(т.е. НОД(P,Q) == const)
    if(is_coprime(P,Q))
        cout << "Полиномы P и Q взаимно-просты!" << endl;

    //вычисление производной, осуществляется
    //стандартная операция символьного дифференцирования:
    sparse_polynom<rational<> > dP = diff(P);
    cout << "Первая производная P: dP/dx = " << dP << endl;
    cout << "Вторая производная P: d2P/dx2 = "
        << diff(dP) << endl << endl;

    //вычисление наибольшего общего делителя:
    cout << "Наибольший общий делитель полиномов P и Q: pq = "
        << gcd(P,Q) << endl << endl;
```

```
    //вычисление наименьшего общего кратного:
    cout << "Наименьшее общее кратное полиномов P и Q: "
        << lcm(P,Q) << endl << endl;

    //вычисление коэффициентов Безу:
    sparse_polynom<rational<> > U,V, pq;
    pq = euclidean_bezout(P,Q,U,V);
    cout << "gcd(P,Q) = P * U + Q * V где" << endl
         << "gcd(P,Q) = " << pq << endl
         << "U = " << U << endl
         << "V = " << V << endl << endl;

    return 0;
}
```

Результат работы программы листинга *Sparse_Polynom_Functions2.cpp*

Демонстрирует применение функций *diff*, *gcd*, *lcm*, *euclidean* к полиномам.

```
Исходный sparse_polynom P = x^6+x^4-x^2-1
Исходный sparse_polynom Q = x^3-2*x^2-x+2

Первая производная P: dP/dx = 6*x^5+4*x^3-2*x
Вторая производная P: d2P/dx2 = 30*x^4+12*x^2-2

Наибольший общий делитель полиномов P и Q: pq = 25*x^2-25

Наименьшее общее кратное полиномов P и Q: 1/25*x^7-
2/25*x^6+1/25*x^5-2/25*x^4-1/25*x^3+2/25*x^2-1/25*x+2/25

gcd(P,Q) = P * U + Q * V где
gcd(P,Q) = 25*x^2-25
U = 1
V = -x^3-2*x^2-6*x-12
```

4.8 Пример использования полиномов: построение интерполяционного многочлена

До сих пор рассматриваемые примеры демонстрировали только возможности предоставляемые Arageli безотносительно каких-либо конкретных задач. Этот пример показывает использование полиномиальной арифметики на примере решения задачи построения интерполяционного многочлена Лагранжа:

Листинг *Sparse_Polynom_Interpolation.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;

sparse_polynom<rational<> >
    Lagrang(rational<> *x, rational<> *y, int size)
{
    rational<> tmpDenom;//Знаменатель
    sparse_polynom<rational<> > poly,       //Результат
                                tmpPolyNumer(1),//Числитель
                                mono("x");

    //построение заготовки для числителя
    for(int i = 0; i < size; i++)
        tmpPolyNumer *= mono - x[i];


    for(int j = 0; j < size; j++)
    {
        //вычисление знаменателя
        tmpDenom = 1;
        for(int k = 0; k < size; k++)
            if(k != j)
                tmpDenom *= x[j]-x[k];
        //прибавление следующего слагаемого
        poly +=
            (tmpPolyNumer / (mono - x[j])) * (y[j] / tmpDenom);
    }

    return poly;
}

int main(int argc, char *argv[])
{
    rational<> x[] = {0,1,2,3,4,5,6};
    rational<> y[] = {rational<>(1,3),-1,0,6,7,-3,-7};
    sparse_polynom<rational<> > L = Lagrang(x,y,7);
    cout << "Интерполяционный многочлен Лагранжа L = "
        << L << endl;
    return 0;
}
```

Результат работы программы листинга *Sparse_Polynom_Interpolation.cpp*

Демонстрирует использование полиномов при постоении интерполяционного многочлена Лагранжа.

```
Интерполяционный многочлен Лагранжа
L = 7/2160*x^6+13/144*x^5-709/432*x^4+1115/144*x^3
-12991/1080*x^2+9/2*x+1/3
```

4.9 Пример использования полиномов: поиск рациональных корней многочлена

Следующий пример демонстрирует взаимодействие между различными частями библиотеки. Здесь в ходе вычислений используются класс *vector* и функции модуля <*prime.hpp*>:

Листинг *Sparse_Polynom_Roots.cpp*

```
#include "Arageli.hpp"

using namespace std;
using namespace Arageli;
using Arageli::vector;

bool findoneroot(sparse_polynom<rational<> > &poly,
                 big_int &num, big_int &den)
{
    //разложение на простые множители
    vector<big_int> num_factorization, den_factorization;
    factorize(num < 0 ? -num : num, num_factorization);
    factorize(den < 0 ? -den : den, den_factorization);

    //генерация всех вариантов рац.корней
    //и их подстановка в полином
    big_int den_variants = power(2, den_factorization.size()),
            num_variants = power(2, num_factorization.size());
    typedef vector<big_int>::iterator pfactor;

    for(big_int den_mask = 0; den_mask <= den_variants;
                                          ++den_mask)
        for(big_int num_mask = 0; num_mask <= num_variants;
                                          ++num_mask)
        {
            //построение числителя
            big_int j = num_mask;
            big_int trial_num = 1;
```

```
            for(pfactor factor = num_factorization.begin(),
                end = num_factorization.end();
                        factor != end && j != 0; ++factor)
            {
                if(j.is_odd()) trial_num *= *factor;
                j >>= 1;
            }

            //построение знаменателя
            big_int i = den_mask;
            big_int trial_den = 1;
            for(pfactor factor = den_factorization.begin(),
                end = den_factorization.end();
                        factor != end && i != 0; ++factor)
            {
                if(i.is_odd()) trial_den *= *factor;
                i >>= 1;
            }

            //подстановка полученных пробных корней
            if(poly.subs(rational<>(trial_num,trial_den)) == 0)
            {
                num = trial_num;
                den = trial_den;
                return true;
            }

            if(poly.subs(rational<>(-trial_num,trial_den)) == 0)
            {
                num = -trial_num;
                den = trial_den;
                return true;
            }
        }
    return false;
}

void findroots(sparse_polynom<rational<> > poly,
               vector<rational<> > &ret)
{
    ret.resize(0);//пока найденных корней нет
```

```
    //отделяем нулевые корни
    while(poly.subs(rational<>(0)) == 0)
    {
        ret.push_back(rational<>(0,1));
        poly /= sparse_polynom<rational<> >(rational<>(1,1),1);
    }

    if(poly.is_const()) return;//у константы корней тоже нет

    //приведение всех коэффициентов к общему знаменателю
    typedef sparse_polynom<rational<> >::monom_iterator pmonom;
    big_int nok = 1;
    for(pmonom i = poly.monoms_begin(), j = poly.monoms_end();
                                                i != j; ++i)
        nok *= i->coef().denominator()
                / gcd(nok,i->coef().denominator());
    for(pmonom i = poly.monoms_begin(), j = poly.monoms_end();
                                                i != j; ++i)
        i->coef() *= nok;

    big_int num = poly.monoms_begin()->coef().numerator();
    big_int den = (--poly.monoms_end())->coef().numerator();

    //отделение одного корня
    while(!poly.is_const() && findoneroot(poly,num,den))
    {
        //запоминаем найденный корень
        ret.push_back(rational<>(num,den));

        //понижаем степень полинома
        sparse_polynom<rational<> > tmp(rational<>(den,1),1);
        tmp += rational<>(-num,1);
        poly /= tmp;
        //числитель и знаменатель для поиска следующего корня
        num = poly.monoms_begin()->coef().numerator();
        den = (--poly.monoms_end())->coef().numerator();
    }
}

int main(int argc, char *argv[])
{
    sparse_polynom<rational<> > P =
        "x^7+167/15*x^6-221/20*x^5-91/12*x^4+27/10*x^3+6/5*x^2";
    cout << "Полином P = " << P << endl;

    vector<rational<> > roots;
    findroots(P,roots);//вычисление корней
```

```
    cout << "Имеет рациональные корни: " << roots << endl;
    return 0;
}
```

Результат работы программы листинга *Sparse_Polynom_Roots.cpp*

Демонстрирует возможности библиотеки на примере поиска рациональных корней полинома.

```
Полином P =
x^7+167/15*x^6-221/20*x^5-91/12*x^4+27/10*x^3+6/5*x^2
Имеет рациональные корни: (0, 0, -12, 1/2, -1/2, -1/3, 6/5)
```