Arageli User's Guide

Nina Gonova Andrey Kamaev Sergey S. Lyalin Nikolai Yu. Zolotykh

August 31, 2006

Contents

Chapter 1

Introduction

Arageli is a C++ template library for doing symbolic (*i.e.* exact or algebraic) computation. It contains implementations of whole numbers of arbitrary length, rational numbers (fractions), vectors, matrices, polynomials and different algorithms involving these objects.

Chapter 2

Big Integers and Fractions

2.1 Creation, input, output

ARAGELI contains the implemetation of *big integers*, *i.e.* whole numbers of arbitrary length. In order to use them one has to include the header file <arageli/big_int.hpp>. These numbers are represented by class *big_int*.

There are several constructors to create *big_int*. The simplest one is

big_int()

It creates big integer equaled to 0. Constructor

big_int (*const char* **str*)

creates big integer using its string decimal representation. There is a set of supplementary constructors with the only parameter that may be a number of any standard C++ integer class. In this cases the big integer with the appropriate value will be created.

If a string representation of a big number is incorrect, exception *Arageli*::*big_int*::*incorrect_string* will be ejected.

ARAGELI contains the implementation of *rational numbers*, *i.e.* fractions of two integers: numerator and denomerator. They are represented as the templated class *rational*<*T*>. The parameter of the template is the type of numerator and denomerator. By default, they are big integers. In order to work with rational numbers it's necessary to include the header file <arageli/rational.hpp>.

There are several constructors for rational numbers. Constructors

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

have the same meaning as the appropriate constructors for big integers. The decimal string representation for a rational number may have the form m/n,

with m being numerator, n being denominator, or simply m if the denominator is 1.

Consider other constructors:

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

The former creates the fraction with numerator equaled to w and the unit denominator. The later creates the rational number u/v.

For input/output of big integers and rational numbers one can use standard streams *std*::*istream* и *std*::*ostream*. Operators >> and << are implemented.

Functions *numerator*() and *denominator*() return what you expect. Function *normalize*() reduces the fraction by dividing its numerator and demominator by their GCD. Function *is_normal*() checks whether the fraction can be reduced or not. Normally you needn't to call *normalize*() because ARAGELI always reduces all fractions. The only case when you usually have to call *normalize*() is after you have changed numerator or/and denominator directly by means of functions *numerator*() and *denominator*().

Consider the following example.

Listing BigIntRationalCreation.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    // Default constructor
    big_int zero;
    cout << "By default, constructor creates zero = "
        << zero << endl;

    // Define the big_int by a string
    big_int big_number =
        "10110011100011110000111110000011111000000";
    cout << "Really big integer: " << big_number << endl;

    // Input from a keyboard is analogous to defining by a string
    big_int number_from_input;
    cout << "Input a big integer: ";
    cin >> number_from_input;
    cout << number_from_input << endl;

    // Now create a rational number
```

```
    rational<> rat_zero;
    cout << "For rational numbers zero = "
        << rat_zero << endl;
    cout << "Its numerator is "
        << rat_zero.numerator() << endl;
    cout << "Its demonimator is "
        << rat_zero.denominator() << endl;

    // Rational number can be defined by a string
    rational<> fraction = "705/32768";

    // Input from a keabord is easy
    rational<> rational_number_from_input;
    cout << "Input any rational number: ";
    cin >> rational_number_from_input;
    cout << rational_number_from_input << endl;

    // You can assign integer numbers to rational variables
    rational<> rational_integer_number = number_from_input;
    cout << "The following rational number is whole: "
        << rational_integer_number << endl;

    // You can specify numerator and denominator
    rational<> number_pi(22, 7);
    cout << "Pi approximately is " << number_pi << endl;
    // You can assign integer and rational numbers to floating point variables
    double floating_point_pi = number_pi;
    cout << "The following is rough approximation of pi: "
        << setprecision(14) << floating_point_pi << endl;
    // pi = 3.14159265358979323846264338327950288419 7
    rational<> pi(355, 113);
    cout << "More precisely Pi is " << pi << " ~ "
        << setprecision(14) << double(pi) << endl;

    // You can get an access to numerator and denominator
        // of a rational number and you can change them.
        // After that perhaps the fraction must be reduced
    fraction.numerator() += number_from_input;
    cout << "Reducing the fraction: " << fraction << " = ";
        fraction.normalize();
    cout << fraction << endl;

    return 0;
}
```

The results of the program follow.

```
By default, constructor creates zero = 0
Really big integer: 10110011100011110000111110000011111000000
Input a big integer: 27082006
```

```
For rational numbers zero = 0
Its numerator is 0
Its demonimator is 1
Input any rational number: 442/721
The following rational number is whole: 27082006
Pi approximately is 22/7
The following is rough approximation of pi: 3.1428571428571
More precisely Pi is 355/113 ~ 3.141592920354
Reducing the fraction: 27082711/32768 = 27082711/32768
```

Some other functions involving big integers and rational numbers will be discussed below.

2.2 Arithmetical and other operations

Operators +, −, *, / do with big integers and rational numbers what you expect:

- + performs addition; also, it is a unary plus;
- − performs subtraction or negation;
- * performs multiplication;
- / performs division; for two big integer operands it returns the integer part of the result;
- % finds the residue after the division of two big integers.

In dividing negative integers the agreements of the standard C++ (not mathematical rules) are used, for example, (-10)/3 = -3, 10/(−3) = −3, (−10)%3 = -1, 10%(-3) = -1 etc.

An attept to divide be zero leads to ejecting the exception *Arageli*::*divizion_by_zero*.

Each of mentioned operators has its combined form: +=, −=, *=, /=, %=.

Standard comparing operators <, >, <=, >=, ==, != are defined.

We remark that all the arithmetical and comparing operators mentioned above can involve operands of different types.

For big integers you can also use prefix and postfix operators ++, −− that have the same meaning what they have for standard integer types such as *int*.

For raising to a power there is function *power* defined in <arageli/powerest.hpp>. Also this file contatins the definition of function *square* for squaring and the following functions:

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)

The former method computs a quotient q and a remainder r after the division a/b such that operators / and % would return. The later method computes a quotient q and a remainder r after the division a/b such that $a = bq + r$, $0 \geq r \leq |b|$ (this corresponds to usual mathematical agreements).

Big integers can be converted to standard numerical types by means of operators such as *int*, *double* etc. Rational number can be converted to decimal fraction by means of *operator double*.

As we have already mentioned functions *numerator*() and *denominator*() give us an access to numerator and denominator corresponingly. Function *normalize*() normalizes a fraction. Function *is_normal*() checks whether the fraction is reducible or not. Function *is_integer*() checks whether the fraction is integer number. Function *inverse*() swaps the numerator and denominator.

You can operate directly with bits of a big integer. Function *length*() returns its length in bits. An access to a particular bit is carried out by means of brackets []. The lowest bit has the index 0. The highest one has the index $length - 1$. You can modify bits lower than the highest one. Operators << and >> shift big integers on the specified number of bits.

Functions *is_even*() and *is_odd*() check whether the number is even or odd.

For rational numbers and big integers the following functions are implemented.

- *cmp* compares two numbers; returns $+1$ if the first argument is more than the second one; 0 if they are equaled to each other; -1 if the second argument is less than the second one;
- *sign*() returns the sign of the number;
- *is_null*() checks whether the number is 0;
- *is_unit*() checks whether the number is 1;
- *is_opposite_unit*() checks whether the number is -1;
- *abs* returns the absolute value of the number;
- *swap* swaps two numbers.

For generating pseudo-random integers you can use the following functions

```
static big_int   random_with_length(size_t len)
static big_int   random_with_length_or_less(size_t len)
```

The former function returns the number whose length (in bits) is presisely *len*. The later function returns a random number with length at least *len*.

Consider the following example.

Listing BigIntRationalOperations.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    // Operations with big_int can be performed
    // as with standard C++ types.
        // You can mix in one operation operands of different types
```

```
    big_int day_seconds = (big_int(60) * 60) * 24;
    cout << "A day contains " << day_seconds <<" seconds"<< endl;
    big_int year_seconds = day_seconds * 365;
    cout << "A year contains " << year_seconds <<" seconds"<< endl;
    big_int leap_year_seconds = year_seconds + day_seconds;
    cout << "But a leap year contains " << leap_year_seconds
        << " seconds" << endl;

    if((big_int(12345678987654321) % big_int(111)).is_null())
    {
        cout << "12345678987654321 is divisible by 111" << endl;
        cout << "The quatient is "
            << big_int(12345678987654321) / big_int(111) << endl;
    }
    else
        cout << "12345678987654321 is not divisible over 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 << "We stated that 23^32 < 32^23" << endl;
    else
        if(first_big_number > second_big_number)
            cout << "We stated that 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;

    // Working with rationals is yeasy too
    rational<> sqrt_2 = "1/2";
    for(int i = 0; i < 10; i++)
        sqrt_2 = 1 / (2 + sqrt_2);
    // You can also use (sqrt_2 += 2).inverse();
    sqrt_2 += 1;
    cout << "sqrt(2) is approx. " << 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 << " is approx. "
         << setprecision(16) << double(two) << endl;

    return 0;
}
```

The results of the running of the program follows:

```
A day contains 86400 seconds
A year contains 31536000 seconds
But a leap year contains 31622400 seconds
12345678987654321 is divisible by 111
The quatient is 111222333222111
We stated that 32^23 < 23^32
2^64 = 18446744073709551616
sqrt(2) is approx. 19601/13860
(19601/13860)^2 = 384199201/192099600
384199201/192099600 is approx. 2.000000005205633
```

2.3 Basic algorithms involving integers

Let's consider some more functions involving big integers:

- *template<typename T>*
  T intsqrt (const T &a);
  finds the greatest integer whose square does not exceed a;

- *template<typename T>*
  T inverse_mod (const T &a, const T &n);
  finds the inverse to a modulo n;

- *template<typename T>*
  T factorial (const T &a);
  finds factorial of a.

If you want to use these functions you have to include file <arageli/intalg.hpp>.

Function *lcm* and *gcd* returns the least common multiplier and the greates common divider corresponingly for two integers. Function *euclid_bezout* also returns the Bezout's coefficients. These functions is described in <arageli/gcd.hpp>.

In order to know whether a number is prime or composite you can use functions *is_prime* and *is_composite*. Functions *next_prime* and *prev_prime* return the next and previous prime munbers correspondingly. Function *factorize* finds all prime factors of the only argument. The result will be returned in a vector containing all prime factors. All these functions choose the approprite algorithm. To use them one have to include <arageli/prime.hpp>.

Consider the following example:

Listing BigIntRationalFunctions.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

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

```
    // Compute a factorial and a root
    big_int f = factorial(big_int(16));
    cout << "16! = " << f << endl;
    big_int sf = intsqrt(f);
    cout << "Integer part of sqrt(16!) = " << sf << endl;
    cout << sf << "^2 = " << sf*sf << endl;

    // Find the first composite Fermat's number
    big_int pow = 1;
    big_int fermas_number;
    cout << "Fermat's numbers: " << endl;
    while(is_prime(fermas_number = (big_int(1) << pow) + 1))
    {
        cout << fermas_number << " is prime" << endl;
        pow <<= 1;
    }
    cout << fermas_number << " is composite" << endl;

    // Find its fractorisation
    vector<big_int> factorization;
    factorize(fermas_number, factorization);
    cout << fermas_number << " = ";
    output_list(cout, factorization, "", "", "*");

    return 0;
}
```

The results follow:

```
16! = 20922789888000
Integer part of sqrt(16!) = 4574143
4574143^2 = 20922784184449
Fermat's numbers:
3 is prime
5 is prime
17 is prime
257 is prime
65537 is prime
4294967297 is composite
4294967297 = 641*6700417
```

Chapter 3

Vectors and Matrices

3.1 Creation, input, output

In ARAGELI there is a possibility create vectors and matrices with entries of arbitrary type. In this chapter we'll consider examples involving vectors and matrices with integer and rational entries. In the next chapter devoted to polynomials examples involving vectors and matrices with polynomial entries and polynomials with matrix coefficients will be examined.

If you want to use vectors or/and matrices you have to include <arageli/vector.hpp>or/and <arageli/matrix.hpp>correspondingly.

There is a few constructor of vectors and matrices. The simplest of them are

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

and

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

They creates empty objects: a vector with length 0 and a 0×0 matrix correspondingly.

You can use more advanced constructors for creating vectors and matrices of specified sizes and initiated by specified values. By default these values are 0's. For each kind of the constructor there is a corresponding function *assign* that plays the same role but it can be called at every moment.

For example, constructor

```
matrix<int> A(3, 8, diag);
```

create the integer matrix

$$A = \begin{pmatrix} 8 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 8 \end{pmatrix}.$$

In this example the first parameter is the order of the matrix. The last parameter of the constructor defines the kind of a matrix. Here it is diagonal. The second parameter is the value which the matrix will be populated by (in this example only diagonal entries).

Consider an example illustrating different ways to construct vectors and matrices

Listing VectorMatrixCreation.cpp

```
#include <iostream>
#include <iomanip>
#include "arageli/arageli.hpp"

using namespace std;
using namespace Arageli;

int main ()
{
    // Let's create vector with rational entries
    Arageli::vector<rational<> > a = "(21/3, 3, 4)";
    cout << "Vector defined by its string representation = " << endl;
    output_aligned(cout, a);
    cout << endl;

    // Create a vector populated by a value
    Arageli::vector<int> b(3, 777, fromval);
    std::cout << "Vector populated by a value = " << endl;
    output_aligned(cout, b);
    cout << endl;

    // Create a zero vector
    vector<double> c(3);
    std::cout << "A zero vector = " << endl;
    output_aligned(cout, c);
    cout << endl;

    // Create matrix with rational entries
    matrix<rational<> > A = "((21/3, 3, 4), (3335, 6/5, 75), (81, 9, 10/7))";
    cout << "Matrix defined by its string representation = " << endl;
    output_aligned(cout, A);
    cout << endl;

    // Create 3x3 identity matrix
    matrix<rational<> > E(3, eye);
    cout << "Identity matrix = " << endl;
```

```
    output_aligned(cout, E);
    cout << endl;

    // Create 3x3 square matrix populated by a value
    matrix<sparse_polynom<rational<> > >
        B(3, sparse_polynom<rational<> >("3/2*x^2"), fromval);
    std::cout << "Square matrix populated by a value = " << endl;
    output_aligned(cout, B);
    cout << endl;

    // Create 3x4 matrix
    matrix<double> C(3, 4, fromsize);
    std::cout << "Rectangle matrix = " << endl;
    output_aligned(cout, C);
    cout << endl;

    // Create char matrix!
    matrix<char> D(3, '*', fromval);
    std::cout << "Char matrix = " << endl;
    output_aligned(cout, D);

    return 0;
}
```

Results follow:

```
Vector defined by its string representation =
||7||
||3||
||4||

Vector populated by a value =
||777||
||777||
||777||

A zero vector =
||0||
||0||
||0||

Matrix defined by its string representation =
|| 7    3   4  ||
||3335 6/5  75 ||
|| 81   9  10/7||

Identity matrix =
||1 0 0||
||0 1 0||
||0 0 1||
```

```
Square matrix populated by a value =
||3/2*x^2 3/2*x^2 3/2*x^2||
||3/2*x^2 3/2*x^2 3/2*x^2||
||3/2*x^2 3/2*x^2 3/2*x^2||

Rectangle matrix =
||0 0 0 0||
||0 0 0 0||
||0 0 0 0||

Char matrix =
||* * *||
||* * *||
||* * *||
```

For input of vectors and matrices one can use standard streams *std*::*istream*. Operator >> is implemented. Entries of a vector must be enclosed in parentheses and separated by commas. Entries of a matrix also must be enclosed in parentheses and must be written in row-wise order. Each order is written as a vector, *i.e.* it must be in parentheses and entries is separated by commas. Rows also must be separated by commas.

For vectors/matrices output several possibilities are provided.

1. Simple output $s << A$ where s is a output stream; A is a vector or a matrix. The same format as in operator >> is used.
2. Prettier output using function *output_aligned* (for more information see examples below and refences).
3. Output in LaTeXformat (for more information see examples below and refences).

The following functions deal with the sizes of vectors and matrices:

- *ncols*() returns the number of columns in the matrix;
- *nrows*() returns the number of rows in the matrix;
- *size*() for a vector: returns its length; for a matrix: returns the product of the number of columns and the number of rows;
- *length*() for a vector: returns the length of the vector; for a matrix: returns the maximum of the number of columns and the number of rows.

Consider the following example.

Listing VectorMatrixInputOutput.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;
```

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

    // Let's define vector b
    b = "(2/3, -1/5, 4)";

    // Print b
    cout << "b (operator <<) = " << b << endl;

    // Prettier output
    cout << "b (output_aligned) = " << endl;
    output_aligned(cout, b, "|| ", " ||");

    // Output using latex notation
    cout << "b (output_latex) = " << endl;
    output_latex(cout, b);
    cout << endl;

    cout << "Size of b = " << b.size() << endl;
    cout << "Length of b = " << b.length() << endl;

    // Now let's define matrix A
    A = "((1/2, 2/3), (-5/7, 6), (8/11, -9/2))";

    // Print A
    cout << "A (operator <<) = " << A << endl;

    // Prettier output
    cout << "A (output_aligned) = " << endl;
    output_aligned(cout, A, "|| ", " ||", " ");

    // Output using latex notation
    cout << "A (output_latex) = " << endl;
    output_latex(cout, A);
    cout << endl;

    cout << "Cols in A = " << A.ncols() << endl;
    cout << "Rows in A = " << A.nrows() << endl;
    cout << "Size of A = " << A.size() << endl;
    cout << "Length of A = " << A.length() << endl;

    return 0;
}
```

Results follow:

```
b (operator <<) = (2/3, -1/5, 4)
b (output_aligned) =
```

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

3.2 Matrix algebra

If the operands of operators $+$, $-$, * are vectors or matrices then these operators perform matrix operations. If A and B are matrices of the same size then $A+B$ and $A-B$ are their sum and their difference correspondingly. If the number of column in A is equal to the number of rows in B then $A*B$ is their product. For each operators $+$, $-$, * there is the corresponding operator combined with assignment.

If A is a matrix and b is a vector then $A*b$ will be interpreted as multiplication of A to the *column* b, while $b*A$ will be considered as multiplication of the *row* b to matrix A. In the both cases (if the sizes of operands are consistent) the result will be a vector.

Binary operations $+$, $-$, *, /, % for two vectors of the same length are entry-wise. For example, $a+b$ is the vector of the same length as a and b with entries equaled to the sum of corresponding entries in a and b.

Standard comparing operators $<$, $>$, $<=$, $>=$, $==$, $!=$ also are supported. For vectors and matrices they implement lexicographic comparing. As an example consider operator $<$ applying for two vectors a and b. Let a be of length m and b be of length n. We will say that $a < b$ iff

1. in simultaneous element-wise viewing of these vectors from its beginning to the end the first pair of non-coincident entries a_i, b_i satisfies the inequality $a_i < b_i$, or

2. there are no pairs of non-coincident entries and $m < n$.

Let matrix A have m_1 rows and n_1 columns, B have m_2 rows and n_2 columns. We will say that $A < B$ iff

1. $m_1 < m_2$, or

2. $m_1 = m_2$ и $n_1 < n_2$, or

3. $m_1 = m_2$, $n_1 = n_2$, and matrix A unrolled in row-wise order is lexicographic less than B represented.

Consider the following example.

Listing VectorMatrixArithmetic.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

typedef rational<> Q;

int main(int argc, char *argv[])
{
    matrix<Q> A, B;
    vector<Q> c, d, res;
    Q alpha;
    int beta;

    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)";

    alpha = Q(1, 120);
    beta = -2;

    res = ((A*B)*c - d*alpha)/beta;
    cout << "Result:" << endl;
    output_aligned(cout, res);

    return 0;
}
```

Results follow:

```
Result:
|| 3113/7200 ||
|| 3689/432  ||
||-23477/5040||
```

The description of entry-wise comparing is in the next section.

3.3 Entry-wise operations under vectors

3.3.1 Operations with entries of vectors

We recall that binary operations $+$, $-$, *, /, % for vectors of the same length are entry-wise. For example, $a + b$ is the vector of the same length as a and b with entries equaled to the sum of corresponding entries in a and b.

Access to i-th entry of a vector a can be obtained by means of brackets:

```
b[k]
```

We remark that the numeration of entries begins with index 0.

Consider functions which are used for inserting, erasing and swapping entries in a vector.

- *iterator insert(size_type pos, const T &val)*
  inserts new entry *val* at the position *pos*.
- *iterator insert(size_type pos, size_type n, const T &val)*
  inserts *n* new entries with value *val* beginning with the position *pos*.
- *void push_back(const T &val)*
  inserts new entry *val* in the end of the vector.
- *void push_front(const T &val)*
  inserts new entry *val* in the beginning of the vector.
- *iterator erase(size_type pos)*
  deletes an entry at position *pos*.
- *void erase(size_type pos, size_type n)*
  deletes n entries beginning with position *pos*.
- *template<typename T2>*
  void remove (const T2 &v)
  deletes all entries with value *v*.
- *void swap_els (size_type xpos, size_type ypos)*
  swaps entries with indexes *xpos* и *ypos*.

An access to the entries of vectors can be obtained by means of iterators. For detailes see references.

Consider the following example:

Listing VectorEntries.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

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

    cout << "a = " << a << endl;
    cout << "x = " << x << endl;

    // Let's insert new entries
    a.push_front(1);
```

```
    a.push_back(5);
    a.insert(1, 2, x);
    cout << "a (after inserting new entries) = " << a << endl;

    // Now let's change the middle entry
    int m_index = (a.size())/2;
    a[m_index] = -1;
    cout << "a (after changing the middle entry) = " << a << endl;

    // Remove all entries equaled to x
    a.remove(x);
    cout << "a (after removing all entries equaled to x) = "
        << a << endl;

    // Swap the first and last entries
    a.swap_els(0, a.size() - 1);
    cout<<"\na (after swapping the first and the last entries) = "
        << a << endl;

    return 0;
}
```

Results follow:

```
a = (3)
x = 3
a (after inserting new entries) = (1, 3, 3, 3, 5)
a (after changing the middle entry) = (1, 3, -1, 3, 5)
a (after removing all entries equaled to x) = (1, -1, 5)

a (after swapping the first and the last entries) = (5, -1, 1)
```

3.3.2 Entry-wise comparing of vectors

As we have alredy mentioned in ARAGELI in addition to lexicografic comparing functions there are entry-wise comparing functions. These can be used only in the case if the vectors are of the same length.

Below there is a list of functions for entry-wise comparing. Functions names show the relationship of the functions with corresponding functions for scalars.

- *each_cmp*
- *each_sign*
- *each_is_positive*
- *each_is_negative*
- *each_less*
- *each_greater*

- *each_lessequal*
- *each_greater_equal*
- *each_equal*
- *each_not_equal*.

Each of these functions returns a vector whos entries are values returned by the corresponding scalar function applying to the pair of corresponding entries (or to the only entry if the scalar function has one parameter).

For example, $each\_cmp(a,\ b)$ returns a vector c such that $c[i] = cmp(a[i],\ b[i])$, while $each\_is\_positive(a)$ returns vector c such that $c[i] = is\_positive(a)$ etc.

If it is necessary to determine whether all entries of a vector satisfy some condition or all pairs of corresponding entries is in specific relation one can use one of the following fuctions:

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

All these functions return *true* iff the condition is true for all pairs of corresponding entries in two parameters of the function or for all entries of the only parameter.

3.3.3 LCM and GCD for vector entries

In ARAGELI there are functions *lcm* and *gcd* that find LCM and GCD for entries of a vector as well as for all pairs of corresponding entries in two vectors. In the later case the parameters must have the same length and the result is the vector of the same length. We recall that in ARAGELI any binary operation under vectors is entry-wise.

If you want to use *lcm* or/and *gcd* you have to include <arageli/gcd.hpp>.

Consider the following example.

Listing VectorLCMGCD.cpp

```
#include <arageli/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 << "a = " << a << endl;
    cout << "b = " << b << endl;

    cout << "GCD for entries in a = " << gcd(a) << endl;
    cout << "LCM for entries in b = " << lcm(b) << endl;

    cout << "GCD for a and b = " << gcd(a, b) << endl;
    cout << "LCM for a and b = " << lcm(a, b) << endl;

    return 0;
}
```

Results follow:

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

3.4 Matrix operations

3.4.1 Operations involving rows and columns

You can perform various operations under matrices such as inserting/deleting rows/columns, their multiplication/division by a scalar, swapping rows/columns etc.

Consider some of them:

- *void insert_row (size_type pos, const T &val)*
  The function inserts in the matrix a new row populated by value *val* at the position *pos*. All other rows shifts down.

- *template<typename T1, bool REFCNT1>*
  void insert_row (size_type pos,
  const **vector**< *T1, REFCNT1* > *&vals)*
  The fuction inserts in the matrix a new row formed by entries in *vals* at the position *pos*.

- *void insert_rows (size_type pos, size_type n, const T &val)*
  The function inserts in the matrix *n* new rows populated by value *val*, beginning with position *pos*.

- *template<typename T1, bool REFCNT1>*
  void insert_rows (size_type pos, size_type n,
  const **vector**< *T1, REFCNT1 > &vals)*
  The function inserts in the matrix n new rows each of them is formed by enties in *vals*, beginning with position *pos*.

- *void erase_row (size_type pos)*
  The function deletes a row at position *pos*. All other rows shift up.

- *void erase_rows (size_type pos, size_type n)*
  The function deletes n rows, beginning with position *pos*.

- *void swap_rows (size_type xpos, size_type ypos)*
  The function swaps two rows at positions *xpos* and *ypos*.

- *template<typename T1>*
  void mult_row (size_type i, const T1 &x)
  The function multiplyes the i-th row by x.

- *template<typename T1>*
  void div_row (size_type i, const T1 &x)
  The function divides the i-th row by x.

- *void add_rows (size_type i, size_type j)*
  The function adds to the i-th row the j-th row.

- *void sub_rows (size_type i, size_type j)*
  The function subtracts from the i-th row the j-th row.

- *template<typename T2>*
  void addmult_rows (size_type i, size_type j, const T2 &y)
  The function adds to the i-th row the j-th row multiplied by y.

- **vector**<*T, true*> *copy_row (size_type i) const*
  The function returns the copy of the i-th row.

Analogous operations are available also for matrix columns. The functions names can be obtained from the names for row-oriented operations by substituting substring *row* for *col*.

The numeration of rows and columns begins with 0. $A(i, j)$ is an entry of matrix A in i-th row and j-th column.

Consider the operation involving a row/column and a entry in the same row/column. For example we have to divide the i-th row by the entry $A(i, j)$. In this case it is necessary to use *safe_reference*$(A(i, j))$ instead of $A(i, j)$:

A.div_row$(i,$ *safe_reference*$(A(i, j)))$;

Analogous safety measure will be taken for polynomials.

Consider the following example.

Listing MatrixColumnRow.cpp

```
#include <arageli/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 << "A = " << endl;
    output_aligned(cout, A);
    cout << endl;

    cout << "The first row in A = " << A.copy_row(0) << endl;
    cout << "The second row in A = " << A.copy_row(1) << endl;
    cout << "The first column in A = " << A.copy_col(0) << endl;
    cout << "The second column in A = " << A.copy_col(1) << endl;

    cout << endl << "A[0] = " << k << "*A[1] (rows)" << endl;
    A.addmult_rows(0, 1, k);
    cout << "Result = " << endl;
    output_aligned(cout, A);

    cout << endl << "A[1] = A[1] / A(2, 1) (columns)" << endl;
    A.div_col(1, safe_reference(A(2, 1)));
    cout << "Result = \n" << endl;
    output_aligned(cout, A);

    cout << endl << "Erase two columns = " << endl;
    A.erase_cols(2, 2);
    output_aligned(cout, A);
    cout << endl;

    cout << "Vector b = " << b << endl;
    cout << "Insert b into A = " << endl;
    A.insert_col(2, b);
    output_aligned(cout, A);
    cout << endl;

    A.swap_rows(0, 2);
    cout<<"Swap in A the first and the third rows = " << endl;
    output_aligned(cout, A);

    return 0;
```

}

The results follow:

```
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)
The first column in A = (-2/5, 2/5, -5/2)
The second column in A = (-1/5, -1/4, 5/6)

A[0] = 5*A[1] (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) (columns)
Result =

||8/5  -87/50 34/15 13/6||
||2/5  -3/10   1/3  1/3 ||
||-5/2   1    -6/7  1/4 ||

Erase two columns =
||8/5  -87/50||
||2/5  -3/10 ||
||-5/2   1   ||

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

Swap in A 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 Other functions

In Arageli there is a few functions for determinig the matrix shape:

- *is_empty*() checks whether the matrix is empty, *i.e.* the number of columns or/and the number of rows is 0;
- *is_null*() checks whether the matrix is of zeros;

- *is_unit*() checks whether the matrix is identity;
- *is_opposite_unit*() checks whether the matrix is opposite to an identity matrix;
- *is_square*() checks whether the matrix is square;
- *swap* swaps two matrices.

3.5 Linear algebra

File <arageli/gauss.hpp> contains the definition of a few linear algebra functions. Function *rank* returns the rank of a matrix. For a square matrix functions *det* and *inverse* find the determinant and the inverse matrix correspondingly. If a matrix is degenerated then *inverse* ejects exception *Arageli*::*matrix_is_singular*. These function works only with matrices whos entries are from a field, for example, with rational matrices. If you want to find the rank or the determinant of an integer matrix you can use functions *rank_int* and *det_int* correspondingly.

Consider the following example.

Listing MatrixGauss.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

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

    matrix< rational<> > A, A_inv;
    rational<> d;

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

    cout << "A = " << endl;
    output_aligned(cout, A, "|| ", " ||", " ");
    cout << endl;

    if (A.is_square())
    {
        d = det(A);
        cout << "det(A) = " << d << endl << endl;

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

            cout << "Inverse to A = " << endl;
            output_aligned(cout, A_inv, "|| ", " ||", " ");
            cout << endl << "The result is " << boolalpha
```

```
                << (A*A_inv).is_unit() << endl << endl;
        }
    }

    matrix< big_int > B;
    big_int delta;

    B = "((1, 2, 3) , (3, 2, 3) , (0, 1, 3))";

    cout << "B = " << endl;
    output_aligned(cout, B, "|| ", " ||", " ");
    cout << endl;

    if (B.is_square())
    {
        delta = det_int(B);
        cout << "det(B) = " << delta << endl;
    }

    return 0;
}
```

Results follow:

```
A =
|| 2/3  -3/5  1    ||
|| -4/7 5/8  1/12 ||
||  -3   5   -6/7 ||

det(A) = -4139/3528

Inverse to A =
|| 3360/4139 -79128/20695 11907/20695 ||
|| 2610/4139  -8568/4139   2212/4139  ||
|| 3465/4139 27048/20695  -1302/20695 ||

The result is true

B =
|| 1 2 3 ||
|| 3 2 3 ||
|| 0 1 3 ||

det(B) = -6
```

We remark that this example is not written using the best technique because both *det* and *inverse* call the same function *rref*. Function *rref* finds the row reduced echelon form of A.

Listing MatrixRref.cpp

```
#include <arageli/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 << "A = " << endl;
    output_aligned(cout, A, "|| ", " ||", " ");
    cout << endl;

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

    cout << "The row reduced echelon form of A = " << endl;
    output_aligned(cout, B, "|| ", " ||", " ");
    cout << endl;

    cout << "The inverse to A = " << endl;
    output_aligned(cout, A_inv, "|| ", " ||", " ");
    cout << endl;

    cout << "det(A) = " << d << endl << endl;

    cout << "The result is " << boolalpha << (A*A_inv).is_unit();

    return 0;
}
```

Results follow:

```
A =
|| 2/3  -3/5  1    ||
|| -4/7 5/8  1/12 ||
||  -3   5   -6/7 ||

The row reduced echelon form of A =
|| 1 0 0 ||
|| 0 1 0 ||
|| 0 0 1 ||

The inverse to A =
|| 3360/4139 -79128/20695 11907/20695 ||
|| 2610/4139  -8568/4139   2212/4139  ||
|| 3465/4139 27048/20695  -1302/20695 ||
```

```
det(A) = -4139/3528

The result is true
```

For integer matrix one have to use *rref_int* instead of *rref*.

Function *solve_linsys*(*A*, *b*) solves the square system of linear equations $Ax = b$. If A is degenerate the the exception *Arageli*::*matrix_is_singular* is ejected. The function is also defined in <arageli/gauss.hpp>.

Listing MatrixLinearSystem.cpp

```
/*
Solving the system of linear equations $Ax=b$,
$ A = \left(\begin{array}{rrr} -1/2 & 2/3 & 3/6 \\ 5/7 & -6 & 7/5\\ -8/11 & 9/2 & -11\end{array}\r
$ b = \left(\begin{array}{ccc} \phantom{+}2/3 \\ -1/5 \\ \phantom{+}4 \end{array}\right)$
*/

#include <arageli/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 = " << endl;
    output_aligned(cout, A);
    cout << endl;
    cout << "b = " << endl;
    output_aligned(cout, b);
    cout << endl;

    try
    {
        x = solve_linsys(A, b);

        cout << "x = " << endl;
        output_aligned(cout, x);
        cout << endl;
        cout << "The result is " << boolalpha << (A*x == b) << endl;
    }
    catch(matrix_is_singular)
    {
        cout << "Error! Matrix is singular!" << endl;
    }
```

```
    return 0;
}
```

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

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

x =
||-247709/119498||
|| -17591/59749 ||
||-41469/119498 ||

The result is true
```

3.6 Smith's normal diagonal form for integer matrix

Listing MatrixSmith.cpp

```
#include <arageli/arageli.hpp>

// Smith's normal diagonal form

using namespace std;
using namespace Arageli;

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

    matrix< big_int > A, B, P, Q;
    size_t rk;
    big_int d;

    A = "((1, 2, 3), (3, 2, 3), (0, 1, 3))";

    cout << "A = " << endl;
    output_aligned(cout, A, "|| ", " ||", " ");
    cout << endl;

    smith(A, B, P, Q, rk, d);
```

```
    cout << "B = " << endl;
    output_aligned(cout, B, "|| ", " ||", " ");
    cout << endl;

    cout << "P = " << endl;
    output_aligned(cout, P, "|| ", " ||", " ");
    cout << endl;

    cout << "Q = " << endl;
    output_aligned(cout, Q, "|| ", " ||", " ");
    cout << endl;

    cout << "det(A) = " << d << endl;
    cout << "det(B) = " << det_int(B) << endl;
    cout << "det(P) = " << det_int(P) << endl;
    cout << "det(Q) = " << det_int(Q) << endl;
    cout << "B == P*A*Q: it's " << boolalpha << (B == P*A*Q) << endl;

    return 0;
}
```

```
A =
|| 1 2 3 ||
|| 3 2 3 ||
|| 0 1 3 ||

B =
|| 1 0 0 ||
|| 0 1 0 ||
|| 0 0 6 ||

P =
|| 1  0 0 ||
|| 0  0 1 ||
|| -3 1 4 ||

Q =
|| 1 -2 3  ||
|| 0 1  -3 ||
|| 0 0  1  ||

det(A) = -6
det(B) = 6
det(P) = -1
det(Q) = 1
B == P*A*Q: it's true
```

Chapter 4

Sparse Polynomials

4.1 Creation

A polynomial with great amount of zero coefficients is called *sparse*. ARAGELI has great possibilities for dealing with sparse polynomials. In ARAGELI The sparse polynomial is a templated class *sparse_polynom*defined in <arageli/ sparse_polynom.hpp>. This class represents a polynomial as a list of non-zero monomials. Their parameters define the type of coefficients (*Coefficient*) and the tipe of degrees (*Degree*). Also there is a boolean parameter to conrol the system of reference counter:

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

Here we'll define only the first parameter. For two other parameters we will use the default values. For example, definitions

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

introduce variables a and b. Polynomial a has integer coefficients while b has rational polynomials.

Consider polynomial constructors:

- *sparse_polynom*()
  creates zero polynomial (it has no non-zero monomials);
- *sparse_polynom*(*const char** *str*);
  defines a polynomial by meens of its string representation; polynomials are written in the form similar to mathematical notation;
- *sparse_polynom*(*const Coefficient*& *a*);
  creates a polynomial with the only monomial x of zero degree;
- *sparse_polynom*(*const Coefficient*& *a*, *const Degree*& *p*);
  creates a polynomial with the only monomial ax^p;

- *template <typename Coefficient1, typename Degree1>*
  sparse_polynom(const monom<Coefficient1, Degree1>& m);
  creates a polynomial equal to monomial m;

- *template <typename Coefficient1, typename Degree1,*
  bool REFCNT1>
  sparse_polynom
  (const sparse_polynom<Coefficient1,Degree1,REFCNT1>& x);
  creates a polynomial equal to another polynomial;

Consider the following example:

Listing SparsePolynomCreation.cpp

```
#include "arageli/arageli.hpp"

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    // Let's create some polynomials
    sparse_polynom<int> S = "2*x^2+5*x-7+3*x";
    cout << "An integer polynomial = "
        << endl << S << endl << endl;

    sparse_polynom<big_int>
        B = "12345678910111213141516171819 20*x^777"
            "+112233445566778899";
    cout << "A big integer polynomial = "
        << endl << B << endl << endl;

    sparse_polynom<double>
        D = "1.12345*x-1e25*x^2+1234.5678*x^3-0.000002334";
    cout << "A real polynomial = "
        << endl << D << endl << endl;

    sparse_polynom<rational<> >
        R = "1234/56781*x^321+7/9*x+3*x^2-4/256";
    cout << "Rational polynomial = "
        << endl << R << endl << endl;

    // It is posible to create a polynomial with matrix coefficients
    // Pay attention to extra parentheses
    sparse_polynom<matrix<int> > M =
        "(((1,2),(3,4))*x^55-((1,2),(4,5))*x+((5,-1),(4,0)))";
    cout << "Polynomial with matrix coefficients = "
        << endl << M << endl << endl;

    // Now let's create a polynomial from separate monomials
    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 << "A polynomial constructed from separate monomials = "
         << endl << F << endl << endl;

    // Converting types of polynomials
    sparse_polynom<big_int> BD = D;
    cout << "A polynomial converted from another = "
        << endl << BD << endl << endl;

    sparse_polynom<double> FR = R;
    cout << "A polynomial converted from another = "
        << endl << FR << endl << endl;

    return 0;
}
```

```
An integer polynomial =
2*x^2+8*x-7

A big integer polynomial =
123456789101112131415161718192O*x^777+112233445566778899

A real polynomial =
1234.57*x^3-1e+025*x^2+1.12345*x-2.334e-006

Rational polynomial =
1234/56781*x^321+3*x^2+7/9*x-1/64

Polynomial with matrix coefficients =
((1, 2), (3, 4))*x^55+((-1, -2), (-4, -5))*x+((5, -1), (4, 0))

A polynomial constructed from separate monomials =
89/144*x^5+34/55*x^4+13/21*x^3+5/8*x^2+2/3*x+1

A polynomial converted from another =
1234*x^3-1000000000000000000905969664*x^2+x

A polynomial converted from another =
0.0217326*x^321+3*x^2+0.777778*x-0.015625
```

4.2 Input and output of polynomials

Input and output of polynomials from/into streams are supported. Polynomials are written in the form similar to mathematical notation. The rules are the same as in constructing from the string representation:

- polynomial must be written without spaces;
- independent variable is always x;
- the degree is separeted from x with ^;
- symbol * bitween a coefficient and x is necessary;
- the order of monomials is arbitrary;
- you can use monomials with the same degree;
- if coefficients are of standard classes of C++ then they are written conforming to the rules of C++; if they are of ARAGELI classes then they must be written conforming the ARAGELI rules.
- if the first coefficient is in paretheses then the entire polynomial must be in parenthese; otherwise the parenthesis can be omitted.

For example, the string x^3+3*x^2+3*x+1 corresponds to usual mathematical formula $x^3 + 3x^2 + 3x + 1$.

Usually ARAGELI stores any polynomial in a canonical form, *i.e.*

- monomials are in the degree decreasing order;
- similar terms are reduced;
- monomials with zero coefficients are excluded;

Listing SparsePolynomInputOutput.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    sparse_polynom<int> S;
    cout << "Enter a polynomial with integer coefficients "
        << endl << endl;

    cin >> S; // 5*x^2-7*x^6+5+x^8-3*x^2+0*x

    cout << "Standard form: " << S << endl;

    return 0;
}
```

```
Enter a polynomial with integer coefficients

Standard form: x^8-7*x^6+2*x^2+5
```

4.3 Arithmetic operations

In ARAGELI arithmetical operations involving polynomials are written using habitual symbols:

- + performs addition,
- − performs subtraction,
- * performs multiplication,
- / finds the quotient,
- % finds the remainder.

Each of these opeartions has a pair one combined with assignment: +=, −= etc.. In addition to arithmetic operations involving two polynomials there supported operations involving a polynomial and a monomial and operations involving a polynomial and a scalar.

Listing SparsePolynomOperations.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    sparse_polynom<rational<> > f = "1/15*x^4+5/7*x^3+7*x";
    sparse_polynom<rational<> > g = "20/53*x^3-1/9*x-1/8";
    sparse_polynom<rational<> > h = "2/5*x^2+5/7*x-7/8";

    cout << "f = " << f << endl;
    cout << "g = " << g << endl;
    cout << "h = " << h << endl << endl;

    cout << "f + g * h = " << f + g * h << endl << endl;
    cout << "Division: f = q*h + r where" << endl;
    sparse_polynom<rational<> > q = f/h;
    sparse_polynom<rational<> > r = f%h;
        cout << "q = " << q << endl;
        cout << "r = " << r << endl;
        // Check the result
        cout << "It's " << boolalpha << (f == q*h + r) << endl << endl;

    typedef sparse_polynom<rational<> >::monom ratmonom;
```

```
    // Operations involving monomilas and scalars

    cout << "Let's divide f by x: " << endl << "f = "
        << (f /= ratmonom(1, 1)) << endl << endl;

    cout << "Let's divide f and g by their leading coefficients:" << endl;
    cout << "f = " << (f /= rational<>(20, 53)) << endl;
    cout << "g = " << (g /= rational<>(1, 15)) << endl << endl;
    cout << "Now f - g = " << f - g << endl;

    return 0;
}
```

```
f = 1/15*x^4+5/7*x^3+7*x
g = 20/53*x^3-1/9*x-1/8
h = 2/5*x^2+5/7*x-7/8

f + g * h = 8/53*x^5+1871/5565*x^4+11341/33390*x^3-163/1260*x^2+883/126*x+7/64

Division: f = q*h + r where
q = 1/6*x^2+125/84*x-3595/1568
r = 10228/1029*x-3595/1792
It's true

Let's divide f by x:
f = 1/15*x^3+5/7*x^2+7

Let's divide f and g by their leading coefficients:
f = 53/300*x^3+53/28*x^2+371/20
g = 300/53*x^3-5/3*x-15/8

Now f - g = -87191/15900*x^3+53/28*x^2+5/3*x+817/40
```

4.4 Polynomial properties

Function *is_normal*() checks whether the polynomial is in canonical representations or not. Usually any polynomial is always in its canonical form. The only case when this form can be disturbed is after you have manipulated their internal representation (see the next section). In this case you have to call *normalize*().

is_null() checks the condition $p = 0$.

is_x() checks the condition $p = x$ where x is independent variable.

is_const() checks whether the polynomial is scalar or not.

Function *degree*() returns the degree of the polynomial; functions *leading_coef*() returns the leading coefficient; and *leading_monom*() returns the leading monomial; *size*() returns the number of non-zero monomials in the polynomial.

subs(*Coefficient*& x) finds the value of the polynomial in the point x.

***Listing** SparsePolynomProperties.cpp*

```
#include <arageli/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 << "S = " << S << endl << endl;

    if (S.is_normal())
        cout << "S is in canonical form" << endl;
    else
    {
        cout << "S is not in canonical form!" << endl;
        // This would mean that there is a bug in Arageli!
        S.normalize();
    }

    if (S.is_null())
        cout << "S = 0" << endl;
    else
        cout << "S is not zero" << endl;

    if (S.is_x())
        cout << "S = x" << endl;
    else
        cout << "S is not x" << endl;

    if (S.is_const())
        cout << "S is rational number" << endl;
    else
        cout << "S is not a scalar" << endl;

    // Two ways to determine the leading coefficient
    cout << "The leading coefficient is "
         << S.leading_coef() << " = "
         << S.leading_monom().coef() << endl;

    // Two ways to determine the degree of the polynomial
    cout << "The degree of S is "
         << S.degree() << " = "
         << S.leading_monom().degree() << endl;

    // The value of the polynomial at the point
    cout << "S(1/2) = " << S.subs(rational<>(1,2)) << endl;
    cout << "S(0) = " << S.subs(rational<>(0)) << endl;

    return 0;
```

```
}

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

S is in canonical form
S is not zero
S is not x
S is not a scalar
The leading coefficient is 2/55 = 2/55
The degree of S is 55 = 55
S(1/2) = 1046371599110366637744052777965 53/184545826870304546417100718080
S(0) = 567
```

4.5 Manipulating with internal representation

Sometimes we have to get an access to internal representation of a polynomial. ARAGELI allows us to manipulate with separate monomials. We pay attention to that after all such manipulations one must call *normalize*() to set the canonical representation of the polynomial.

The iterators technique analogouse to one in *STL* is supported in ARAGELI. This technique allows us to consider a polynomial as a list of its monomials. There are 3 kinds of iterators. Each of these 3 kinds can have constant form and non-constatn form. These iterators work with the list of monomials and separately with lists of coefficients and degrees. You can insert and erase monomials by means of functions:

monom_iterator insert(*monom_iterator pos*, *const Arageli*::*monom*<*F1*, *I1*>& *x*);

monom_iterator erase(*monom_iterator pos*);

The former inserts the monomial before the position specified by the iterator. The later erases the monomial specified by the iterator.

The following example show how one can use this technique.

Listing SparsePolynomIterator.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    typedef sparse_polynom<big_int> poly;
    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 << "S = " << S << endl << endl;
```

```
    // Let's find the minimal non-zero coefficient
    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 << "Minimal non-zero coefficient in S is " << min_coeff
        << endl << endl;

    // Find the sum of all degrees of S
    // We'll use the STL algorithms
    int dsum = std::accumulate(S.degrees_begin(), S.degrees_end(), 0);
    cout << "The sum of all degrees of S is "
        << dsum << endl << endl;

    // Pick out all monomials with odd degrees
    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 << "The polynomial with only odd degrees = " << oddS << endl << endl;

    // Replace all degrees by their residues modulo 5
    for(degrees di = S.degrees_begin(), dj = S.degrees_end(); di != dj; ++di)
        *di %= 5;

    // Now S can contain monomials this equal degrees
    // We have to reduce S to its canonical form
    S.normalize();
    cout << "All degrees modulo 5 = " << S << endl << endl;

    return 0;
}
```

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

Minimal non-zero coefficient in S is -4235

The sum of all degrees of S is 7935

The polynomial with only odd degrees = 213*x^3443-4235*x^15+292*x

All degrees modulo 5 = -744*x^4+213*x^3+532*x^2+292*x+30019
```

4.6 Other operations

For a set of operations such as computing opposite, swapping, comparing ARAGELI provides a common interface and polynomials not are an exception.

sparse_polynom<T> opposite(sparse_polynom<T> p) returns opposite polynomial.

cmp(sparse_polynom<T1> first, sparse_polynom<T2> second) performs lexicographic comparing.

swap(sparse_polynom<T> first, sparse_polynom<T> second) swaps polynomials.

Consider the following example:

Listing SparsePolynomOtherOperations.cpp

```
#include <arageli/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 << "P = " << P << endl;
    cout << "Q = " << Q << endl << endl;

    int P_vs_Q = cmp(P,Q);
    if (P_vs_Q < 0) cout << "P < Q" << endl;
    if (P_vs_Q > 0) cout << "P > Q" << endl;
    if (P_vs_Q == 0) cout << "P == Q" << endl;

    int P_vs_P = cmp(P,P);
    if (P_vs_P < 0) cout << "P < P" << endl;
    if (P_vs_P > 0) cout << "P > P" << endl;
    if (P_vs_P == 0) cout << "P == P" << endl;
    cout << endl;

    cout << "Before swapping P and Q:" << endl;
    cout << "P = " << P << endl;
    cout << "Q = " << Q << endl << endl;
    swap(P, Q);
    cout << "After swapping:" << endl;
    cout << "P = " << P << endl;
    cout << "Q = " << Q << endl << endl;

    cout << "-P = " << opposite(P) << endl;

    return 0;
}
```

```
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
```

```
P > Q
P == P

Before swapping P and 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

After swapping:
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

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

4.7 Basic algorithms

ARAGELI implements basic algebraic algorithms involving polynomials: *diff* performs the symbolic differentiation, *gcd* and *lcm* find the GCD and LCM of two polynomials correspondingly, *euclid_bezout* finds Bezout's coefficients, *is_coprime* checks whether polynomials are coprime or not.

Listing SparsePolynomAlgorithms.cpp

```
#include <arageli/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 << "P = " << P << endl;
    cout << "Q = " << Q << endl << endl;

    if (is_coprime(P, Q))
        cout << "P and Q are coprime" << endl << endl;
    else
        cout << "P and Q are not coprime" << endl << endl;

    sparse_polynom<rational<> > dP = diff(P);
    cout << "The first derivative dP/dx = " << dP << endl;
    cout << "The second derivative d2P/dx2 = " << diff(dP) << endl << endl;

    cout << "GCD(P, Q) = " << gcd(P, Q) << endl;
    cout << "LCM(P, Q) = " << lcm(P, Q) << endl << endl;

    sparse_polynom<rational<> > U, V, pq;
    pq = euclid_bezout(P, Q, U, V);
```

```
    cout << "U = " << U << endl;
    cout << "V = " << V << endl << endl;
    cout << "Check for gcd(P, Q) == P*U + Q*V: ";
    cout << boolalpha << (pq == P*U + Q*V);

    return 0;
}
```

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

P and Q are not coprime

The first derivative dP/dx = 6*x^5+4*x^3-2*x
The second derivative d2P/dx2 = 30*x^4+12*x^2-2

GCD(P, Q) = x^2-1
LCM(P, Q) = x^7-2*x^6+x^5-2*x^4-x^3+2*x^2-x+2

U = 1/25
V = -1/25*x^3-2/25*x^2-6/25*x-12/25

Check for gcd(P, Q) == P*U + Q*V: true
```

4.8 Smith's normal diagonal form for polynomial matrix

Listing SparsePolynomMatrixSmith.cpp

```
#include <arageli/arageli.hpp>

// Smith's normal diagonal form

using namespace std;
using namespace Arageli;

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

    matrix< sparse_polynom<rational<> > > A, B, P, Q;
    size_t rk;
    sparse_polynom<rational<> > d;

    A = "((x+1,x-1), (x-2,x+1))";

    cout << "A = " << endl;
    output_aligned(cout, A, "|| ", " ||", " ");
    cout << endl;
```

```
    smith(A, B, P, Q, rk, d);

    cout << "B = " << endl;
    output_aligned(cout, B, "|| ", " ||", " ");
    cout << endl;

    cout << "P = " << endl;
    output_aligned(cout, P, "|| ", " ||", " ");
    cout << endl;

    cout << "Q = " << endl;
    output_aligned(cout, Q, "|| ", " ||", " ");
    cout << endl;

    cout << "det(A) = " << d << endl;
    cout << "det(B) = " << det_int(B) << endl;
    cout << "det(P) = " << det_int(P) << endl;
    cout << "det(Q) = " << det_int(Q) << endl;
    cout << "B == P*A*Q: it's " << boolalpha << (B == P*A*Q) << endl;

    return 0;
}
```

```
A =
|| x+1 x-1 ||
|| x-2 x+1 ||

B =
|| 1   0    ||
|| 0 x-1/5 ||

P =
||  0  1/3 ||
|| 3/5 2/5 ||

Q =
|| -1 1/3*x+1/3  ||
|| 1  -1/3*x+2/3 ||

det(A) = 5*x-1
det(B) = x-1/5
det(P) = -1/5
det(Q) = -1
B == P*A*Q: it's true
```

4.9 Example: *matrix polynomial* ↔ *polynomial matrix* conversion

Listing SparsePolynomMatrix.cpp

```
#include <arageli/arageli.hpp>

// $ \left(\begin{array}{rrr} 3 & -5 \\ 0 & 7 \end{array}\right) x^8 + \left(\begin{array}{rrr} 0 & 1
// $ \left(\begin{array}{ccc} 3x^8-1 & -5x^8+x^3+9 \\ 13 & 7x^8-8x^3+2 \end{array}\right)$

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    sparse_polynom<matrix<int> >
        f = "(((3,-5),(0,7))*x^8+((0,1),(0,-8))*x^3+((-1,9),(13,2)))";

    matrix<sparse_polynom<int> >
        x = "((x,0),(0,x))";

    cout << "f(x) = " << f << endl << endl;
    cout << "Let x = " << x << endl << endl;
    cout << "Then the resulting matrix is" << endl;

    output_aligned(cout, f.subs(x), "|| ", " ||", " ");

    return 0;
}
```

```
f(x) = ((3, -5), (0, 7))*x^8+((0, 1), (0, -8))*x^3+((-1, 9), (13, 2))

Let x = ((x, 0), (0, x))

Then the resulting matrix is
|| 3*x^8-1 -5*x^8+x^3+9  ||
||   13    7*x^8-8*x^3+2 ||
```

4.10 Example: interpolating polynomial

Consider the problem of constructing Lagrange interpolating polynomial. We don't think that we've implemented the best way to determine coefficients of an interpolating polynomial. The code must be considered only as illustration of using Arageli functions.

Listing SparsePolynomLagrange.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
```

```
using namespace Arageli;

sparse_polynom<rational<> > Lagrange(rational<> *x, rational<> *y, int n)
{
    rational<> tmpDenom; // denominator
    sparse_polynom<rational<> > poly, // result
      tmpPolyNumer(1), // numerator
      mono("x");

    // a blank for numerator
    for (int i = 0; i < n; i++)
        tmpPolyNumer *= mono - x[i];

    for (int j = 0; j < n; j++)
    {
        // computing a denominator
        tmpDenom = 1;
        for (int k = 0; k < n; k++)
            if (k != j)
                tmpDenom *= x[j]-x[k];

        // next addand
        poly += (tmpPolyNumer / (mono - x[j])) * (y[j] / tmpDenom);
    }

    return poly;
}

int main(int argc, char *argv[])
{
    int n = 7;
    rational<> x[] = {0, 1, 2, 3, 4, 5, 6};
    rational<> y[] = {rational<>(1,3), -1, 0, 6, 7, -3, -7};

    sparse_polynom<rational<> > L = Lagrange(x, y, n);

    cout << "Interpolating polynomial" << endl << "L(x) = " << L << endl << endl;

    cout << "Let's check it" << endl;

    bool ok = true;

    for (int i = 0; i < n; i++)
    {
        rational<> yy = L.subs(x[i]);
        cout << "L(" << x[i] << ") = " << yy << endl;

        if (yy != y[i])
        {
            ok = false;
```

```
            cout << "Error!" << endl;
        }
    }

    if (ok)
        cout << "All is Ok" << endl;


    return 0;
}
```

```
Interpolating polynomial
L(x) = 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

Let's check it
L(0) = 1/3
L(1) = -1
L(2) = 0
L(3) = 6
L(4) = 7
L(5) = -3
L(6) = -7
All is Ok
```

4.11 Example: finding all rational roots

Let's implement the classical method for finding all rational roots of a polynomial due to Kroneker. The method is very slow and the code have to be considered only as an example of using ARAGELI functions.

Listing SparsePolynomKroneker.cpp

```
#include <arageli/arageli.hpp>

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

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

    // All combinations of numerators and denominators
    // Substitute them into poly
    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)
    {
        // Constructin a numerator
        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;
        }

        // Constructing a denomiator
        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;
        }

        // Substitute a trial root into poly
        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); // for the present there are no roots found

    // First find zero roots
```

```
    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;

    // Find common denominator
    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();

    // Find all roots
    while(!poly.is_const() && findoneroot(poly, num, den))
    {
        // Store tye root found
        ret.push_back(rational<>(num, den));

        // Reduce the degree of the polynomial
        sparse_polynom<rational<> > tmp(rational<>(den, 1), 1);
        tmp += rational<>(-num, 1);
        poly /= tmp;

        // Take numerator and denominator for the next root
        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 << "P = " << P << endl << endl;

    vector<rational<> > roots;
    findroots(P,roots);

    cout << "Rational roots: " << roots << endl;
    return 0;
}
```

```
P = x^7+167/15*x^6-221/20*x^5-91/12*x^4+27/10*x^3+6/5*x^2

Rational roots: (0, 0, -12, 1/2, -1/2, -1/3, 6/5)
```

Chapter 5

Modular arithmetic

5.1 Creation

Listing Residue.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
{
    // Working in ${\bf Z}\sb{5} = {\bf Z}/5{\bf Z}$

    residue<int> a = "2 (mod 5)";
    residue<int> b = "3 (mod 5)";

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a + b = " << a + b << endl;
    cout << "a - b = " << a - b << endl;
    cout << "a * b = " << a * b << endl;
    cout << "a / b = " << a / b << endl << endl;
}

{
    // Working in ${\bf Q}/(x^2 + 1){\bf Q}$

    typedef residue<sparse_polynom<rational<> > > T;

    T a = "((1+x) (mod (x^2+1)))";
    T b = "((1-x) (mod (x^2+1)))";

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
```

```
    cout << "a + b = " << a + b << endl;
    cout << "a - b = " << a - b << endl;
    cout << "a * b = " << a * b << endl;
    cout << "a / b = " << a / b << endl << endl;
}

{
    // Working in ${\bf Z}\sb{3}/(x^2 + 1){\bf Z}\sb{3}$

    typedef residue<sparse_polynom<residue<int> > > T;

    T a = "( ((1(mod 3))*x+(1(mod 3))) (mod ((1(mod 3))*x^2+(1(mod 3)))) )";
    T b = "( ((1(mod 3))*x+(2(mod 3))) (mod ((1(mod 3))*x^2+(1(mod 3)))) )";

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a + b = " << a + b << endl;
    cout << "a - b = " << a - b << endl;
    cout << "a * b = " << a * b << endl;
    cout << "a / b = " << a / b << endl << endl;
}

    return 0;
}
```

```
a = 2(mod 5)
b = 3(mod 5)
a + b = 0(mod 5)
a - b = 4(mod 5)
a * b = 1(mod 5)
a / b = 4(mod 5)

a = x+1(mod x^2+1)
b = -x+1(mod x^2+1)
a + b = 2(mod x^2+1)
a - b = 2*x(mod x^2+1)
a * b = 2(mod x^2+1)
a / b = x(mod x^2+1)

a = x+1(mod +3)(mod x^2+1(mod +3))
b = x+2(mod +3)(mod x^2+1(mod +3))
a + b = 2(mod 3)*x(mod x^2+1(mod +3))
a - b = 2(mod 3)(mod x^2+1(mod +3))
a * b = 1(mod 3)(mod x^2+1(mod +3))
a / b = 2(mod 3)*x(mod x^2+1(mod +3))
```

5.2 Linear algebra over finite field

Listing ResidueLinearSystem.cpp

```
#include <arageli/arageli.hpp>

using namespace std;
using namespace Arageli;

int main(int argc, char *argv[])
{
    typedef residue<int> T;

    int mod = 7;

    matrix<T> A = "((3, 1, 2), (1, 2, 3), (4, 3, 2))";
    vector<T> b = "(1, 1, 1)";
    vector<T> x;

    for(matrix<T>::iterator i = A.begin(); i < A.end(); ++i)
    {
        i->module() = mod;
        i->normalize();
    }

    for(matrix<T>::iterator i = b.begin(); i < b.end(); ++i)
    {
        i->module() = mod;
        i->normalize();
    }

    cout << "A = " << endl;
    output_aligned(cout, A);
    cout << endl;

    cout << "b = " << endl;
    output_aligned(cout, b);
    cout << endl;

    try
    {
        x = solve_linsys(A, b);

        cout << "x = " << endl;
        output_aligned(cout, x);
        cout << endl;
        cout << "Check the result: " << boolalpha << (A*x == b) << endl;
    }
    catch(matrix_is_singular)
    {
```

```
        cout << "Error! Matrix is singular!" << endl;
    }

    return 0;
}
```

```
A =
||3(mod 7) 1(mod 7) 2(mod 7)||
||1(mod 7) 2(mod 7) 3(mod 7)||
||4(mod 7) 3(mod 7) 2(mod 7)||

b =
||1(mod 7)||
||1(mod 7)||
||1(mod 7)||

x =
||2(mod 7)||
||6(mod 7)||
||5(mod 7)||

Check the result: true
```