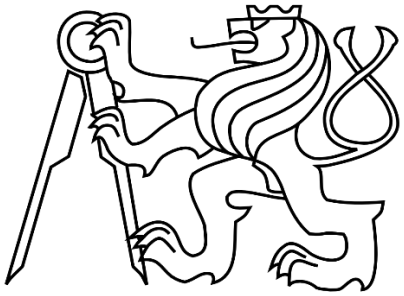
Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Science and Engineering



Master's thesis

Analysis and comparison of methods of Gröbner bases for solving nonlinear polynomial systems in finite fields of characteristic 2

Bc. Ildar Nizamov

Supervisor: Dr. Bestoun S. Ahmed Al-Beywanee, Ph.D.

Study Program: Open Informatics

Field of Study: Software Engineering

May 24, 2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Nizamov** Jméno: **Ildar** Osobní číslo: **474701**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Analysis and comparison of methods of Grobner bases for solving nonlinear polynomial systems in finite fields of characteristic 2.

Název diplomové práce anglicky:

Analysis and comparison of methods of Grobner bases for solving nonlinear polynomial systems in finite fields of characteristic 2.

Pokyny pro vypracování:

Grobner bases method is a universal and powerful tool for solving nonlinear polynomial equations in some algebraic fields. This method is a generalization of Gaussian elimination algorithm in nonlinear case.
This diploma thesis consists of three main parts. In the first part we consider the basic theory of the Grobner bases and their properties. The second part is about the different algorithms for construction of Grobner basis. There we make the comparison and analysis of these algorithms. In the last part we show how to use this methods to solve the system of the nonlinear equations in the case of algebraic finite fields with 2^n elements, which are very useful in Coding Theory and Cryptography. Also, in this part the experimental results for the implemented algorithms are shown.

Seznam doporučené literatury:

1) Cox, David A.; Little, John; O'Shea, Donal (1997). Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra. Springer. ISBN 0-387-94680-2
2) Bruno Buchberger (1965). An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. Ph.D. dissertation, University of Innsbruck. English translation by Michael Abramson in Journal of Symbolic Computation 41 (2006): 471?511. [This is Buchberger's thesis inventing Gröbner bases.]
3) Faug?re, J.-C. (July 2002). ""A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)"" (PDF). Proceedings of the 2002 international symposium on Symbolic and algebraic computation (ISSAC). ACM Press: 75?83. doi:10.1145/780506.780516. ISBN 1-58113-484-3

Jméno a pracoviště vedoucí(ho) diplomové práce:

Dr. Bestoun S. Ahmed Al-Beywanee, Ph.D., katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **16.04.2018** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.09.2019**

Dr. Bestoun S. Ahmed Al-Beywanee, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání	Podpis studenta

Acknowledgments

First of all, I would like to thank my teacher and mentor from KFU side Vladimir Kugurakov for his amazing ability to explain the unexplainable thing in a simple and clear way, and for all his advices and recommendations, which were very helpful for me in the process of writing this thesis. Also, thanks a lot to my family and friends for their support and encouragement.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Kazan, May, 2018 _____

Abstract

Grobner bases method is a universal and powerful tool for solving nonlinear polynomial equations in some algebraic fields. This method is a generalization of Gaussian elimination algorithm in nonlinear case. This diploma thesis consists of three main parts. In the first part, we consider the basic theory of the Grobner bases and their properties. The second part is about the different algorithms for construction of Grobner basis. There we make the comparison and analysis of these algorithms. In the last part, we show how to use these methods to solve the system of the nonlinear equations in the case of algebraic finite fields with 2^n elements, which are very useful in Coding Theory and Cryptography. Also, in this part the experimental results for the implemented algorithms are shown.

Contents

List of figures

List of tables.

Introduction

Preamble.

In present day problems where solution of the systems of algebraic equations is required arise in many fields of exact sciences. For example, we must solve these problems in the cryptography and coding theory with respect to finite fields. It obvious that in the general case solution of such a problem is not a trivial task, although for a variety of such systems there exists a whole lot of various ways to find the solution.

As the most obvious example, we can pay attention to the degree of equations in given system. At this moment, many exact and iterative methods are known for solving systems of linear algebraic equations, including methods of Cramer, Gauss, the sweep method etc. However, the solution of systems of nonlinear equations is much more complicated problem, which is solved using huge amounts of time and memory. Special methods are also invented for such systems, for example, the linearization method and the linearization sets method.

In this paper, we study the Gröbner bases method, first used in the 1980s and received an impressive amount of attention for its universality. Its study was accelerated by the development of electronic computing devices, which made it possible to make quick calculations over polynomials. We will describe, analyze and compare different algorithms for the Gröbner bases calculation and their application to solve such systems. Moreover, the comparison will be done based on the our own implementation of these algorithms, because we consider the finite field F_{2^n}. This type of field is not supported by any free software, which is used for the Gröbner bases calculation. Therefore, our work will also give us the tool to solve the systems in the F_{2^n} and we will be able to consider a behaviour of the algorithms for the Gröbner bases calculation with respect to such fields.

Statement of the task and the basic notation.

This paper considers the problem of solving systems of nonlinear polynomial equations in a finite field F_{2^n}. **The goals of this thesis** are to:

1) Give definition of the Gröbner bases and describe their properties, which can help to solve such systems;
2) Find and describe the algorithm for the Gröbner bases calculation;
3) Consider different ways to optimize this algorithm;
4) Apply this algorithm in the process of solving systems;

5) Make an implementation of all considered algorithms for different algebraic fields.
6) Compare the implementations in practice in the case of the field F_{2^n}.

To begin with, we will define a few basic definitions, which will be used further in this paper.

Definition. A group is a set G with a given binary operation *, for which the following axioms are satisfied:

1) Operation's closure

$$a * b \in G \forall a, b \in G$$

2) Operation's associativity

$$a * (b * c) = (a * b) * c \forall a, b, c \in G$$

3) Existence of the identity element e such that

$$a * e = e * a = a \forall a \in G$$

4) Existence of the inverse element a^{-1} for any a, for which holds

$$a * a^{-1} = a^{-1} * a = e \forall a \in G$$

Definition. The group G is called the commutative group or the abelian group, if holds

$$a * b = b * a \forall a, b \in G$$

Definition. The group is called the finite group, if it has the finite number of elements. This number is called the order of the group.

Definition. A ring is a set R with two given binary operations $+$ and $\cdot$ such that:

1) R is the abelian group with respect to $+$ operation.
2) Associativity of the $\cdot$ operation
3) $\cdot$ operation is distributive with respect to $+$ operation.

$$(a + b) \cdot c = (a \cdot c) + (b \cdot c) \forall a, b, c \in R$$
$$a \cdot (b + c) = (a \cdot b) + (a \cdot c) \forall a, b, c \in R$$

Definition. A field is an algebraic structure , $+$, $\cdot$, for which these axioms are satisfied:

1) , $+$ is the abelian group;
2) , $\cdot$, where $F \quad = F \setminus \{0\}$, is the abelian group;
3) The multiplication and addition operation are connected via the distribution law:

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z) \forall x, y, z \in F$$

Further, we will use the following notation:

1) K is the finite field;
2) $K[x_1, \dots, x_n]$ is the polynomial ring in n variables $x_1, \dots, x_n$ over K;
3) f, g, h, k, p, q are the polynomials from the $K[x_1, \dots, x_n]$;
4) I, F, G are the finite subsets of $K[x_1, \dots, x_n]$;

5) s, t, u are the monomials in the form $x_1^{i_1} \cdot \ldots \cdot x_n^{i_n}$, and we assume that $x_i^0 \equiv 1$ is satisfied;
6) a, b, c, d are the elements of K;
7) i, j, l, m are the natural integers (or 0).

Definition. An ideal is a subset $I \subset K[x_1, \ldots, x_n]$, for which the following holds:

1) $0 \in I$

2) $f \in I, g \in I \Longrightarrow f + g \in I$

3) $f \in I, h \in K[x_1, \ldots, x_n] \Longrightarrow hf \in I$

Definition. Let $f_1, \ldots, f_s$ be the polynomials and $f_i \in K[x_1, \ldots, x_n]$. Then, the set

$$\langle f_1, \ldots, f_s \rangle = \{\sum_{i=1}^{s} h_i f_i : h_i \in K[x_1, \ldots, x_n]\}$$

is called the ideal, generated by $f_1, \ldots, f_s$. These polynomials are called the generating polynomials or the generating set. We will write $f \equiv_F g$, if $f \equiv g (mod \langle F \rangle)$, so that $f - g \in \langle F \rangle$, where F is a subset of $K[x_1, \ldots, x_n]$.

A brief overview of methods for solving the systems of equations.

In present day, in many areas of the theory of coding and cryptography the following methods for solving systems of nonlinear equations are used.

1) Gaussian elimination.

In the case of systems of linear equations, we have a plenty of a various exact and iterative methods for solving them. The most common and the most known method is the Gaussian elimination. The process of solving consists of two parts. During the first stage (called Forward elimination), we transform the system, sequentially eliminating variables from consideration. This continues until we eliminate all variables except one. Now we have the linear equation with one variable. After that, we find the value of this variable. During the second stage (called Back substitution), we solve the system, using the values of variables that are already found to solve other equations in the system.

In the case of system of m equations and m variables, this method make cm^{ω} actions in the F_q field, where c is a constant and ω is a Gaussian reduction value with an upper estimate $\omega \leq 2.376$. The fastest known algorithm called Strassen algorithm has the following values: $c = 7$ and $\omega = log_2 7$.

2) Full search.

This method sequentially looks over all elements of F_q^n until it finds a fulfilling set. The set is called fulfilling if there are no contradictions after substituting this set in the system. In the average case, if $q = 2$ we need to check about 2^{n-1} sets.

3) Full search with prefixes.

This method basically is the depth-first search algorithm, which runs on q-ary tree with height n. So, this method sequentially looks over not all elements of F_q^n, but their prefixes from F_q^k, where k goes from 1 to n. The checking of the prefix v is the following:

a) If we find a contradiction in the system after substitution the prefix fromF_q^k for some k, then the set of variables is not fulfilling. After that, we will check the next prefix, which can be found by the following rule: if the last element of the prefix is the last element of the field, then we remove it from the prefix. Otherwise, we replace it by the next element of the field.

b) If the system after the prefix substitution has no contradictions and it's linear, then we can solve it using methods for solving systems of linear equations, for example, the Gaussian elimination.

c) If the system after the prefix substitution has no contradictions and it's nonlinear, then we check the prefix $v \circ f$ from F_q^{k+1}, where $f \in F_q$ and f is a first element of the field.

f is the initial prefix for this method.

4) Linearization method.

Let $t_1, t_2, \dots, t_r$ denote all different monomials in the system with a degree greater than 1. After that, we substitute them with $y_1, y_2, \dots, y_r$, the values of which are in F_q. We obtained a new linear system. The number of variables in it is less than or equal to $n + r$. Suppose that the system is overdetermined. Therefore, a subset of linearly independent equations exists. If we solve this subset with a Gaussian elimination and substitute obtained values in the original system, we will get a system of monomial equations of the form $u_i = a_i, i = 1, \dots, s$ where $s \leq r, t_i = y_i, i = 1, \dots, r, a_i \in F_q$ and u_i are the monomials. We can easily solve this system.

In the case of F_2, the solving becomes easier. If $a_i = 1$ then all variables, which are contained in u_i, must be equal to 1. Otherwise, at least one of them must be equal to 0.

This method will not work if the resulting system becomes underdetermined after substitution of the monomials. In that case, the system will have $q^{n(L)-m(L)}$ roots where $n(L)$ is a number of variables in the new system and $m(L)$ is a number of linearly independent equations in it. Therefore, we must find a partial solution and substitute it into $t_i = y_i, i = 1, \dots, r$ to

transform the monomial system into the consistent system. This can be done in exponential time.

5) Relinearization. [1]

We use this method if we obtain an underdetermined system after its linearization. We simply add to the system some new nonlinear equations, which are trivial with respect to old variables, but using the new variables. After that we use the linearization (or, recursively, relinearization) method again.

6) Extended linearization. [1]

This method uses the parameter d, where d is a natural integer and $d \geq d_0$. The method uses two steps:

a) Multiplication: the system E_d is built, which is consists of equations in the form $f_i(x)t = b_i t$ for $i = 1, \dots, m$, where tX^k and $k = 0,1, \dots, d - d_0$. The degree of the new system is not greater than d.

b) Linearization: the resulting system E_d is solved using the linearization method.

It's obvious, that $E_{d_0} = E$, so, if the system of equations has degree d_0, then the linearization method and the extended linearization method with $d = d_0$ are equal to each other.

We need to choose a proper value for d. The method works only when the system E_d is overdetermined.

7) Method of the linearization sets

If we determine some variables in the system, it will become linear. Subset of variables Z, which contains these variable, is called a linearization set of the system of equations. Hence, we can construct a linearization set and sequentially check each possible set of values of variables included in the linearization set.

Each time we obtain a new linear system. If this system is consistent, then we can solve it with the Gaussian elimination and find the values of the remaining variables.

This method's complexity is q^p where $p = |Z|$. Therefore, the most efficient algorithm use the smallest linearization set.

8) Buchberger's method

The Gröbner bases are slightly useful for the tasks, which are related with ideals of the polynomial rings. Bruno Buchberger introduced the definition of such bases in his Ph.D. thesis [5]. Mr. Buchberger also introduced a way of easy (but not very efficient) calculation of the Gröbner bases. Afterwards, this method was optimized by many scientists, including Jean-Charles Faugere ([12],[13]), Till Stegers ([15]) and John Perry ([2],[11]).

Chapter 1. The Gröbner bases.

Monomial ordering in $K[x_1, \dots, x_n]$

Let K denote an arbitrary field and let $R = K[x_1, \dots, x_n]$ denote a polynomial ring over a field K. Polynomial is a sum $c_1M_1 + c_2M_2 + \cdots + c_mM_m$ where the c_i are the elements of K and the M_i are monomials. Every monomial is a product $x^\alpha = x_1^{\alpha_1}x_2^{\alpha_2} \dots x_n^{\alpha_n}$ where the α_i are nonnegative integers.

In order to define the Gröbner bases, we need to determine an ordering $\prec$ for the monomials, which is satisfies to following conditions:

$$t(t1 \Longrightarrow 1 \prec t);$$

$$s, t, u(s < t \Longrightarrow su \prec tu).$$

These orderings are called admissible orderings.

Once an admissible monomial ordering is fixed, the terms of a polynomial (the products of a monomials with theirs nonzero coefficients) are naturally ordered by decreasing monomials (with respect to this ordering). This makes the representation of a polynomial as an ordered list of pairs coefficient–exponent vector a canonical representation of the polynomials.

Note that there is a one-to-one correspondence between the monomial $x^\alpha = x_1^{\alpha_1}x_2^{\alpha_2} \dots x_n^{\alpha_n}$ and its exponent vector $\alpha = (\alpha| \quad |1, \alpha_2, \dots, \alpha_n)$. So, $\alpha \prec \beta$ means $x^\alpha \prec x^\beta$.

Admissible monomial ordering examples:

Lexicographical ordering (lex): Let $\alpha = (\alpha_1, \dots, \alpha_n)$, $\beta = (\beta_1, \dots, \beta_n)$. We say that $\alpha \quad {}_{lex}\beta$, if the most left nonzero element of vector $\alpha - \beta$ is greater than 0.

Examples:

1)(2,3,4) ${}_{lex}$(1,2,3), because $\alpha - \beta = (1,1,1)$.

2)(4,5,6) ${}_{lex}$(4,0,3), because $\alpha - \beta = (0,5,3)$.

Inverse lexicographical ordering (invlex): Let $\alpha = (\alpha_1, \dots, \alpha_n)$, $\beta = (\beta_1, \dots, \beta_n)$. We say that $\alpha \quad {}_{invlex}\beta$, if the most right nonzero element of vector $\alpha - \beta$ is greater than 0.

Examples:

1)(2,3,4) ${}_{invlex}$(1,2,3), because $\alpha - \beta = (1,1,1)$.

2)(4,5,6) ${}_{invlex}$(4,3,6), because $\alpha - \beta = (0,2,0)$.

Graded lexicographical ordering (grlex): Let $\alpha = (\alpha_1, \dots, \alpha_n)$, $\beta = (\beta_1, \dots, \beta_n)$. We say that $\alpha \quad {}_{grlex}\beta$, if

$$|\alpha| = \sum_{i=1}^{n} \alpha_i > |\beta| = \sum_{i=1}^{n} \beta_i \vee |\alpha| = |\beta| \wedge \alpha \quad {}_{lex}\beta.$$

Examples:

1)(2,3,8) $_{grlex}$(5,2,3), because |(2,3,8)| = 13 > |(5,2,3)| = 10.

2)(4,6,5) $_{grlex}$(4,5,6), because |(4,5,6)| = |(4,6,5)|, but (4,5,6) $_{lex}$(4,6,5)

Graded reverse lexicographical ordering (grevlex): Let $\alpha = (\alpha_1, \dots, \alpha_n)$, $\beta = (\beta_1, \dots, \beta_n)$. We say that α $_{grevlex}\beta$, if

$$|\alpha| = \sum_{i=1}^{n} \alpha_i > |\beta| = \sum_{i=1}^{n} \beta_i \vee |\alpha| = |\beta| \wedge \alpha \ _{lex}\beta$$

and the most right nonzero element of vector $\alpha - \beta$ is smaller than 0.

Examples:

1)(2,3,8) $_{grevlex}$(5,2,3), because |(2,3,8)| = 13 > |(5,2,3)| = 10.

2)(4,6,5) $_{grevlex}$(4,5,6), because |(4,5,6)| = |(4,6,5)|, but $\alpha - \beta = (0,1,-1)$.

We note that the lex-, grlex- and grevlex-orderings equally order all unit vectors of the $x_1, \dots, x_n$ variables:

$$(1,0,\dots,0) \ _{lex}(0,1,\dots,0) \ _{lex} \dots \ _{lex}(0,0,\dots,1),$$

$$(1,0,\dots,0) \ _{grlex}(0,1,\dots,0) \ _{grlex} \dots \ _{grlex}(0,0,\dots,1),$$

$$(1,0,\dots,0) \ _{grevlex}(0,1,\dots,0) \ _{grevlex} \dots \ _{grevlex}(0,0,\dots,1).$$

Example 1.1. Let's write the polynomial $f = 4xy^2z + 4z^2 - 5x^3 + 7x^2z^2 \subset K[x,y,z]$ with respect to every noted monomial ordering:

1) Lex-ordering

$$f = -5x^3 + 7x^2z^2 + 4xy^2z + 4z^2$$

2) Invlex-ordering

$$f = 4z^2 + 7x^2z^2 + 4xy^2z - 5x^3$$

3) Grlex-ordering

$$f = 7x^2z^2 + 4xy^2z - 5x^3 + 4z^2$$

4) Grevlex-ordering

$$f = 4xy^2z + 7x^2z^2 - 5x^3 + 4z^2$$

Let f denote a polynomial with the fixed monomial ordering and

$$f = \sum_{\alpha} a_\alpha x^\alpha$$

Now we can define the following properties of polynomials.

1) Multidegree of the polynomial f is

$$multideg(f) = \max_{\alpha}(\alpha : a_\alpha \neq 0)$$

where the maximum is taken according to the current monomial ordering.

2) Leading coefficient of the polynomial f is

$$LC(f) = a_{multideg(f)} \in K$$

3) Leading monomial of the polynomial f is

$$LM(f) = x^{multideg(f)}$$

4) Leading term of the polynomial f is

$$(f) = LC(f)LM(f)$$

Division algorithm in $K[x_1, \dots, x_n]$

In fact, the algorithm, which divides the polynomial $f \in K[x_1, \dots, x_n]$ by the collection of polynomials $f_1, f_2, \dots, f_s$ from the ring $K[x_1, \dots, x_n]$, coincides with the one variable case: we need to remove the leading term of the polynomial f, which is defined by the monomial ordering. We can do this in the following way: we multiply one of the polynomials f_i by the proper monomial and subtract the result from f.

Theorem 1.1 (Division algorithm in the ring $K[x_1, \dots, x_n]$). Let > be the admissible monomial ordering and let $F = (f_1, \dots, f_s)$ be the ordered collection of s polynomials from the ring $K[x_1, \dots, x_n]$. Then every polynomial $f \in K[x_1, \dots, x_n]$ can be represented in the following form

$$f = a_1 f_1 + \cdots + a_s f_s + r,$$

where $a_i, r \in K[x_1, \dots, x_n]$ and either $r = 0$, or r is a linear combination of monomials (with the coefficients in$K$), which are not divisible by any of the leading terms $(f_1), \dots, < (f_s)$ of the monomials from the collection F. In this case r is called the reminder of the division of the polynomial f to the collection of polynomials F. Also, if $a_i f_i \neq 0$, then

$$multideg(f) \geq multideg(a_i f_i).$$

Here is the algorithm's pseudocode:

Input: $f_1, \dots, f_s, f$

Output: $a_1, \dots, a_s, r$

$$a_1 = 0; \dots; a_s = 0; r = 0;$$

$$p = f;$$

WHILE $p \neq 0$ DO

$i = 1$;

havedivision false;

WHILE $i \leq s$ AND havedivision false DO

IF (f_i) divides (p) THEN

$a_i = a_i + \frac{(p)}{(f_i)}$;

$p = p - \left(\frac{(p)}{(f_i)}\right) f_i$;

havedivision true;

ELSE

$i = i + 1$;

IF havedivision false THEN

$r = r + (p)$;

$p = p - (p)$;

The proof of this theorem is available in [8].

Example 1.2. Let's divide the polynomial $f = x^2y + xy^2 + y^2$ by $f_1 = xy - 1$ and $f_2 = y^2 - 1$ using the lex-ordering, where $x > y$. In the case of choice between f_1 and f_2, we will use f_1 firstly. At the beginning$p = f$.

Step 1. $(p) = x^2y$ and it is divisible only by $(f_1) = xy$. The result of division is x, and we add it to a_1. After subtraction of $xf_1 = x^2y - x$ from p we obtain $p = xy^2 + x + y^2$.

Step 2. $(p) = xy^2$and it is divisible by both $(f_1) = xy$ and $(f_2) = y^2$. We choose the first polynomial. The result of division is y, and we add it to a_1. So, $a_1 = x + y$. After subtraction of $yf_1 = xy^2 - y$ from p we obtain $p = x + y^2 + y$.

Step 3. $(p) = x$ and it is not divisible by any of the leading terms. But p is not the reminder, because y^2 is divisible by (f_2). So, we add x to r and substract it from p After subtraction we obtain $p = y^2 + y$.

Step 4. $(p) = y^2$ and it is divisible only by $(f_2) = y^2$. The result of division is 1, and we add it to a_2. After subtraction of $f_2 = y^2 - 1$ from p we obtain $p = y + 1$.

Step 5. Both of the terms of p are not divisible by any of the leading terms. So, we add them to the remainder r.

The algorithm is successfully terminated, Therefore, we can represent f in the following form:

$$f = x^2y + xy^2 + y^2 = (x + y)(xy - 1) + 1(y^2 - 1) + x + y + 1$$

We see that all monomials of the remainder are not divisible by any of the leading terms of the divisors. Also, we note that the values of $a_1, \dots, a_{s,}r$ depend on the order of the divisors in F. If we change the order, we obtain

$$f = x^2y + xy^2 + y^2 = (x + 1)(y^2 - 1) + x(xy - 1) + 2x - 1$$

Further, we will see that there are the collections of polynomials, for which these dependencies are absent with respect to the remainder.

Monomial ideals.

Let's begin this chapter with the definition of the monomial ideal.

Definition 1.1. The ideal $I \subset K[x_1, \dots, x_n]$ is a monomial ideal if I consists of all finite sums of the form $\sum_{\alpha \in A} h_\alpha x^\alpha$ where the A is a subset of $Z_{\geq 0}^n$ and each polynomial $h_\alpha \in K[x_1, \dots, x_n]$. Let denote such ideals as $\langle x^\alpha : \alpha \in A \rangle$.

The example of such ideal is the $\langle x^4y^2, x^3y^4, x^2y^5 \rangle \subset K[x, y]$.

Now we characterize all monomials belonging to a given monomial ideal with two lemmas.

Lemma 1.1. Let $I = \langle x^\alpha : \alpha \in A \rangle$ be a monomial ideal. Then the monomial x^β belongs to I if and only if some monomial $x^\alpha, \alpha \in A$ divides x^β.

Proof.

If the monomial x^β is divisible by some monomial $x^\alpha, \alpha \in A$, then x^β belongs to I by definition. Let's prove the inverse statement. Let x^β belong to I. In this case $x^\beta = \sum_{i=1}^{s} h_i x^{\alpha(i)}$, where $h_i \in K[x_1, \dots, x_n]$ and $\alpha(i)$ belongs to A. Every h_i is a linear combination of monomials, so every term in the right part of the equation is divisible by some monomial $x^{\alpha(i)}$. Therefore, x^β have this property too and can be found as a term in some $h_i x^{\alpha(i)}$. ■

Lemma 1.2. Let I be a monomial ideal and $f \in K[x_1, \dots, x_n]$. Then the following conditions are equivalent:

a) f belongs to I.

b) Every term of f belongs to I.

c) f can be represented as a linear combination of monomials from I.

Proof.

The sequence of proofs $(c) \Rightarrow (b) \Rightarrow (a)$ is obvious. The proof of $(a) \Rightarrow (c)$ is conducted exactly like the proof of the second part of Lemma 1.1.■

We obtained that a monomial ideal is uniquely determined by its monomials.

Corollary 1.1. Two monomial ideals equal to each other if and only if sets of monomials, which are belongs to corresponding ideals, are equal.

We obtain the first main result from Lemma 1.2 and Corollary 1.1.

Theorem 1.2 (Dickson's Lemma). Every monomial ideal $I = \langle x^\alpha : \alpha \in A \rangle \subset K[x_1, \dots, x_n]$ can be represented as $I = \langle x^{\alpha(1)}, \dots, x^{\alpha(s)} \rangle$, where $\alpha(1), \dots, \alpha(s) \in A$. In other words, I has got a finite basis.

The proof of Dickson's Lemma is given in [8].

Therefore, every monomial ideal in $K[x_1, \dots, x_n]$ is finitely generated.

Definition 1.2. Let $I \subset K[x_1, \dots, x_n]$ be a nonzero ideal.

1) Let denote the set of all leading terms of elements from I as (I), i.e.

$$(I) = \{cx^{\alpha} : \exists f \in I((f) = cx^{\alpha})\}$$

2) Let denote the ideal generated by elements from (I) as $\langle (I) \rangle$.

Note 1.1. An equality $\langle (f_1), \dots, < (f_s) \rangle = \langle (I) \rangle$ is not always fulfilled despite the fact that I generated by $f_1, \dots, f_s$. Nevertheless, an inclusion $\langle (f_1), \dots, < (f_s) \rangle \subset \langle (I) \rangle$ is always fulfilled.

Example 1.3:

Let $I = \langle f_1, f_2 \rangle$, where $f_1 = x^3 - 2xy$, $f_2 = x^2y - 2y^2 + x$. In the case of the grlex-ordering

$$x^2 = x(x^2y - 2y^2 + x) - y(x^3 - 2xy)$$

Therefore, x^2 belongs to the ideal I, so $x^2 = (x^2)$ belongs to the ideal $\langle (I) \rangle$. But, neither $(f_1) = x^3$ nor $(f_2) = x^2y$ are not divisible by x^2. Hence, x^2 doesn't belong to the ideal $\langle (f_1), < (f_2) \rangle$ by Lemma 1.

The Gröbner bases and their properties.

In order to define the concept of the Gröbner bases, we use the following assertion.

Proposition 1.1. Let $I \subset K[x_1, \dots, x_n]$ be an ideal. Then:

1) $\langle (I) \rangle$ is a monomial ideal;
2) $\exists g_1, \dots, g_s \in I : \langle (g_1), \dots, < (g_s) \rangle = \langle (I) \rangle$

Proof.

Let every polynomial $g \in I - \{0\}$ have the leading monomial $LM(g)$. These leading monomials generate a monomial ideal $\langle LM(g) : g \in I - \{0\} \rangle$. $LM(g)$ differs from (g) only with some nonzero coefficient. Therefore, the new monomial ideal is equal to $\langle (I) \rangle$.

$\langle (I) \rangle$ is generated by the monomials $LM(g), g \in I - \{0\}$, so there is a finite collection of polynomials $g_1, \dots, g_s \in I$, for which $\langle LM(g_1), \dots, LM(g_s) \rangle = \langle (I) \rangle$. Again, $LM(g)$ differs from (g) only with some nonzero coefficient. Hence, $\langle (I) \rangle = \langle (g_1), \dots, < (g_s) \rangle$. ■

Using this statement and the division algorithm, which we defined earlier, we prove the second main result.

Theorem 1.3 (Hilbert's basis theorem). Any ideal $I \subset K[x_1, \dots, x_n]$ is finitely generated, i.e. $\exists g_1, \dots, g_s \in I : I = \langle (g_1), \dots, < (g_s) \rangle$.

The proof of this theorem is given in [8]. The subset of elements $g_1, \dots, g_s$ from the proposition 1.1 are exactly the Gröbner basis of an ideal I, the central element of this paper.

Definition 1.3. Let some monomial ordering be defined. The Gröbner basis of an ideal I is a finite subset $G = \{g_1, \dots, g_s\}$ of elements from I, where

$$\langle (g_1), \dots, < (g_s) \rangle = \langle (I) \rangle.$$

The following corollary follows from the proof of the Hilbert's basis theorem.

Corollary 1.2. Let some monomial ordering be defined. Then each ideal $I \subset K[x_1, \dots, x_n]$, which does not include zeros, has got a Gröbner basis G, and G is the basis of I.

Now we prove one of the most important properties of the Gröbner bases.

Proposition 1.2. Let $G = \{g_1, \dots, g_s\}$ be a Gröbner basis of an ideal $I \subset K[x_1, \dots, x_n]$ and let $f \in K[x_1, \dots, x_n]$. Then there exists a unique polynomial $r \in K[x_1, \dots, x_n]$ that has the following properties:

1) None of the leading terms $(g_i), \dots, < (g_s)$ divides any term of the polynomial r.
2) $\exists g \in I: f = g + r$.

Therefore, r is a reminder of dividing a polynomial f by G. Moreover, r does not depend on the order of the divisors in G. In that case r is called a normal form of f.

Proof.

We can write f in the following form using the division algorithm:

$$f = a_1 g_1 + \cdots + a_s g_s + r,$$

where r satisfies the first condition. Let $g = a_1 g_1 + \cdots + a_s g_s$. The polynomial g belongs to I, so the second condition is satisfied too.

Suppose, there is a different polynomials r' and g' such that the both conditions hold and $f = g' + r'$. In this case, $r - r' = g - g' \in I$. So, if $r \neq r'$ then

$$(r - r') \in \langle (g_1), \dots, < (g_s) \rangle = \langle (I) \rangle.$$

From the Lemma 1.1 we obtain that $(r - r')$ divides some monomial (g_i), which is contradiction to the first condition. Hence, $r = r'$.■

Corollary 1.3. Let $G = \{g_1, \dots, g_s\}$ be a Gröbner basis of an ideal $I \subset K[x_1, \dots, x_n]$ and let $f \in K[x_1, \dots, x_n]$. Then f belongs to I if and only if the reminder of dividing a polynomial f by G is equal to 0.

Proof.

Let reminder be equal to 0. In this case, f belongs to I, obviously. Let's proof the inverse statement. Let f belong to I. We can write f in the form $f = f + 0$, which fully satisfies both conditions from Proposition 1.2. Also, the zero polynomial is the only possible reminder of dividing a polynomial f by G.■

Hence, we can solve the membership of an element in an ideal problem. It is only necessary to find the remainder of dividing the polynomial by the Gröbner basis. Now we need to find a way to build these bases.

Definition 1.4. Let $\acute{f}^F$ denote a remainder of dividing the polynomial f by an ordered collection of polynomials F. This collection can be considered as unordered, if F is the Gröbner basis.

Example 1.4:

Let $F = (x^2y - y^2, x^4y^2 - y^2)$. We will use the lexicographical monomial ordering. In that case $x^{\acute{5}}y^F = xy^3$.

Definition 1.5. Let $f, g \in K[x_1, \dots, x_n]$ be a nonzero elements, where $multideg(f) = \alpha, multideg(g) = \beta$. Let $\gamma = (\gamma_1, \dots, \gamma_n)$ where $\gamma_i = max(\alpha_i, \beta_i)$ for any i. Then the x^γ is a least common multiple of $LM(f)$ and $LM(g)$. We denote it as $x^\gamma = LCM\big(LM(f), LM(g)\big)$.

The S-polynomial of f and g is the following composition:

$$S(f, g) = \frac{x^\gamma}{(f)} \cdot f - \frac{x^\gamma}{(g)} \cdot g$$

It is easy to note that this composition removes the leading terms of polynomials.

Lemma 1.3. Let's consider the sum $\sum_{i=1}^{n} c_i f_i$, where multidegree of f_i is equal to δ, and c_i belongs to K for all i. If the multidegree of this sum is smaller than δ, then this sum is the linear combination of the S-polynomials $S\big(f_j, f_l\big), 1 \le j, l \le s$ with coefficients in K. Moreover, the multidegree of each S-polynomial $S\big(f_j, f_l\big)$ is smaller than δ.

The proof of this Lemma is given in [8]. We obtain that if the combination of polynomials, where every of them has an equal multidegree, removes its own leading terms, then it's relates with the removals in their S-polynomials. In order to identify the Gröbner bases, Buchberger introduced the following criteria based on the last lemma in [6].

Theorem 1.4 (Buchberger's criteria). Let I be a polynomial ideal. Then the basis $G = \{g_1, \dots, g_s\}$ of an ideal I is the Gröbner basis of I if and only if for any pairs $i \neq j$ the reminder of dividing the S-polynomial $S\big(g_i, g_j\big)$ by G is equal to 0.

This criteria is called the S-pairs criteria. It helps us to make an algorithm for the Gröbner bases computation.

Buchberger's algorithm.

The idea of the Buchberger's algorithm based on the S-pairs criteria. We can obtain the Gröbner basis of the collection of polynomials F, sequentially adding $S\big(f_i, \acute{f_j}\big|\ \big)^F$ to F.

Theorem 1.5 (Buchberger's algorithm, [6]). Let $I = \langle f_1, \dots, f_s \rangle$ where I does not contain zeros. Then the Gröbner basis of I is built by the following algorithm in a finite number of steps:

In: $F = (f_1, \dots, f_s)$

Out: the Gröbner basis $G = \{g_1, \dots, g_t\}$ of an ideal I where $F \subset G$

$$G = F$$

REPEAT

$G' = G$

FOR each pair $\{p, q\}, p \neq q$ в G' DO

$S = S(\acute{p}, q)^{G'}$

IF $S \neq 0$ THEN $G = G \cup \{S\}$

UNTIL $G = G'$

Example 1.5. Let $I = \langle f_1, f_2 \rangle = \langle x^3 - 2xy, x^2y - 2y^2 + x \rangle$. We use the graded degree ordering with $x > y$. Previously, we showed that its generating elements are not the Gröbner basis of the ideal. Let's expand this collection. At the beginning $G = \{f_1, f_2\}$.

Step 1. $S(f_1, f_2) = yf_1 - xf_2 = -x^2$, $S(\acute{f_1}, f_2)^G = -x^2 \neq 0$. So, $G = \{f_1, f_2, f_3\}$, where $f_3 = -x^2$.

Step 2. $S(\acute{f_1}, f_2)^G = 0$, $S(f_1, f_3) = f_1 + xf_3 = -2xy$, but $S(\acute{f_1}, f_3)^G = -2xy \neq 0$. So, $G = \{f_1, f_2, f_3, f_4\}$, where $f_4 = -2xy$.

Step 3. $S(\acute{f_1}, f_3)^G = 0$, $S(f_1, f_4) = yf_1 + \frac{1}{2}x^2 f_4 = -2xy^2 = yf_4$. Therefore, $S(\acute{f_1}, f_4)^G = 0$.

Step 4. $S(f_2, f_3) = f_2 + yf_3 = -2y^2 + x$, but $S(\acute{f_2}, f_3)^G = -2y^2 + x \neq 0$. So, $G = \{f_1, f_2, f_3, f_4, f_5\}$, где $f_5 = -2y^2 + x$.

Step 5. It's easy to check that $S\left(\acute{f_i}, f_j\right)^G = 0$ for all $1 \leq i < j \leq 5$.

At the end, we obtain, using the Theorem 4, that for the grlex-ordering the resulting Gröbner basis is:

$$G = \{x^3 - 2xy, x^2y - 2y^2 + x, -x^2, -2xy, -2y^2 + x\}$$

This algorithm, actually, is the simplest version of the Buchberger's algorithm. It does not really helps us in practice. Also, we can see the first improvement of the algorithm. We note that if the remainder $S(\acute{p}, q)^{G'} = 0$, then it will be the same after the expansion of G'.

However, this algorithm builds redundant bases, because they have redundant elements. We can eliminate them, using the following lemma.

Lemma 1.4. Let G be a Gröbner basis of an ideal I, and let the polynomial $p \in I$ where its leading term $(p) \in \langle (G - \{p\}) \rangle$. Then $G - \{p\}$ also is the Gröbner basis of I.

Proof.

It is known that $\langle (G) \rangle = \langle (I) \rangle$. The fact that $(p) \in \langle (G - \{p\}) \rangle$ means that $\langle (G - \{p\}) \rangle = \langle (G) \rangle$. Hence, $G - \{p\}$ is the Gröbner basis by definition.■

Therefore, if we eliminate such polynomials from G and normalize the remaining polynomials, we will obtain the minimal Gröbner basis. But these operations can be done only when the G is a Gröbner basis

Definition 1.6. The minimal Gröbner basis of an ideal I is a Gröbner basis G of I that satisfy the following conditions:

1) All leading coefficients of the basis elements are equal to 1.
2) For any $p \in G$ $(p) \notin \langle (G - \{p\}) \rangle$.

In our example, the resulting minimal Gröbner basis is

$$G = \{x^2, xy, y^2 - \frac{1}{2}x\}$$

Every ideal can have several minimal Gröbner bases. In our example, the minimal Gröbner basis also is

$$G = \{x^2 - axy, xy, y^2 - \frac{1}{2}x\}$$

for any $a \in K$. We must choose the best one. This Gröbner basis is called a reduced Gröbner basis.

Definition 1.7. The reduced Gröbner basis of an ideal I is Grobner basis G of I that satisfy the following conditions:

1) All leading coefficients of the basis elements are equal to 1.
2) For any $p \in G$ none of its monomials belongs to $\langle(G - \{p\})\rangle$.

In our example, the first calculated minimal basis is the reduced Gröbner basis.

Proposition 1.3. Let I be the polynomial ideal, where I doesn't contain zeros, and let the monomial ordering be defined. In that case, there is the only one reduced reduced Gröbner basis.

Proof.

Let's say that the polynomial $g \in G$ is reduced with respect to G, if none of its monomials belongs to $\langle(G - \{g\})\rangle$. We note that if g is reduced with respect to G it is also reduced with respect to any other minimal Gröbner basis G', which contains g and operates with the same set of leading terms as G.

Let$G' = (G - \{g\}) \cup \{\acute{g}^{G-\{g\}}\}$, for some polynomial $g \in G$. This set is the minimal Gröbner basis of an ideal I too, because $\left(\acute{g}^{G-\{g\}}\right) = (g)$ and $\langle(G')\rangle = \langle(G)\rangle$. The polynomial $g' = \acute{g}^{G-\{g\}}$ is also reduced with respect to G'.

If we make this procedure for every element in G the obtained Gröbner basis will be the reduced Gröbner basis, because all of its elements are reduced with respect to it.

Now we have to proof the uniqueness of the reduced basis. Suppose that we have two reduced Gröbner bases G and G'. They share the same set of leading terms ([8]), so for any polynomial $g \in G$ there is an element $g' \in G'$ such that $(g) = (g')$.

Let's consider the polynomial $g - g'$ for any pair g, g'. It belongs to I, so, $g \overset{\acute{}}{-} g'^{G} = 0$. None of the monomials of $g - g'$ belongs to (G) or (G'), because G and G' are reduced and the leading terms of g and g' are gone after subtraction. Hence, $g \overset{\acute{}}{-} g'^{G} = g - g' = 0$. This shows us that $G = G'$. Therefore, the reduced basis is unique.■

Despite the fact that the Buchberger's algorithm is practically unusable, there are much more efficient algorithms, which are based on it. In fact, the most resource-consuming part of the algorithm is the operation of reduction (division) of the S-polynomial by the collection of

polynomials. Therefore, the new algorithms use some criteria to reduce the number of the considered S-polynomials and, consequently, number of reductions.

In order to define such criteria, we need to introduce a new definition.

Definition 1.8. Let $G = \{g_1, \dots, g_s\} \subset K[x_1, \dots, x_n]$ be the set of polynomials, and let some monomial ordering be defined. Then the function f, which belongs to $K[x_1, \dots, x_n]$, is called reducible to zero modulo G ($f \to_G 0$), if it can be represented as

$$f = a_1 g_1 + \cdots a_s g_s$$

where the multidegree of the function is greater or equal than the multidegrees of the nonzero components.

The following theorem holds.

Theorem 1.6. The basis $G = \{g_1, \dots, g_s\}$ of the ideal I is the Gröbner basis if and only if all S-polynomials, constructed from its elements, are reducible to zero modulo G.

The proof of this theorem is obtained during the proof of the Theorem 1.4. Now we obtain that the following fact can be used for the first criterion.

Proposition 1.4. Suppose that we have the finite set $G \subset K[x_1, \dots, x_n]$. Let f, g be the elements of G and their leading terms are coprime. Then the S-polynomial $S(f, g)$ is reducible to 0 modulo G.

Let's introduce another definition, in order to define the second criterion.

Definition 1.9. Let $F = (f_1, \dots, f_s)$ be the collection of polynomials. The syzygy of the leading terms of the collection F is the collection of polynomials $(h_1, \dots, h_s) \subset (K[x_1, \dots, x_n])^s$ such that

$$\sum_{i=1}^{s} h_i < (f_i) = 0$$

Let's denote the subset of $(K[x_1, \dots, x_n])^s$, which contains all syzygies of the leading terms of the collection F as $S(F)$. This subset has a finite basis.

We define the syzygy, which is generated by the S-polynomial $S(f_i, f_j)$, in the following way:

$$S_{ij} = \frac{x^\gamma}{(f_i)} e_i - \frac{x^\gamma}{(f_j)} e_j$$

where e_i is the i-th unit vector from $(K[x_1, \dots, x_n])^s$. It can be shown that such syzygies are also the basis of the $S(F)$, so that any syzygy S from this set can be represented as

$$S = \sum_{i<j} u_{ij} S_{ij}$$

where $u_{ij} \subset K[x_1, \dots, x_n]$.

Using this definition, we obtain the second criterion for the Gröbner bases construction.

Theorem 1.7. The basis$G = \{g_1, \dots, g_s\}$ of the ideal I is the Gröbner basis if and only if the following condition holds for all elements of the basis $S = (h_1, \dots, h_s)$ of $S(G)$:

$$S \cdot G = \sum_{i=1}^{s} h_i g_i \to_G 0$$

We need the following fact to use this criteria.

Proposition 1.5. Suppose that the subset $S \subset \{S_{ij} : 1 \le i < j \le s\}$ is the basis of $S(G)$, where $G = \{g_1, \dots, g_s\}$. Also, suppose that there are three polynomials g_i, g_j и g_l, which belong to G, such that the least common multiple of the leading terms of g_i and g_j is divisible by the leading term of g_l. If S_{il} и S_{jl} belong to S, then $S - \{S_{ij}\}$ is also the basis of $S(G)$.

Now we can build the improved version of Buchberger's algorithm, using these new criteria. Proofs for all mentioned criteria and the following algorithm are available in [8].

Theorem 1.8. The Gröbner basis of the polynomial ideal $I = \langle f_1, \dots, f_s \rangle$ is built by the following algorithm in a finite number of steps:

Input: $F = (f_1, \dots, f_s)$

Output: G, the Gröbner basis of the ideal I

B:=$\{(i, j) \vee 1 \le i < j \le s\}$

G:=F

t:=s

WHILE $B \neq 0$ DO

Take $(i, j) \in B$

IF $LCM\left((f_i), <\left(f_j\right)\right) \neq < (f_i) < \left(f_j\right)$ AND Criteria (f_i, f_j, B) doesn't hold THEN

S:=$S\left(f_i^{'}, f_j\right)^G$

IF $S \neq 0$ THEN

t:=$t + 1$

f_t:=S

G:=$G \cup \{f_t\}$

B:=$B \cup \{(i, t) \vee 1 \le i \le t - 1\}$

B:=$B - \{(i, j)\}$

Criteria (f_i, f_j, B) holds means that there is $l \notin \{i, j\}$, for which the pairs (i, l) и (j, l) don't belong to B and the leading term of f_l divides the least common multiple of the leading terms of f_i и f_j.

This algorithm is much more optimal, but there is still a window for improvement. For example, if we arrange the divisors in the order of increasing the leading terms, there will be some resource saving, because we will decrease the number of comparisons in the reduction

operation. In the 1985, Buchberger suggested to choose pairs (i, j) of polynomials such that the least common multiples of their leading terms were minimal ([6]). This strategy is called the normal choice strategy. Later, it was improved by the sugar strategy. All these strategies are studied in detail from the beginning of the 1980s. Moreover, in the 1985 Buchberger showed how to reduce the G during its expansion ([6]), so we can obtain the reduced bases at once.

The most effective algorithms for the Gröbner bases calculation are the Faugere F4 and F5, which were invented by Jean-Charles Faugere, respectively, in 1999 ([12]) and 2005 ([13]). They significantly reduce the number of considered S-polynomials and reduction operations during the Gröbner bases calculation. In the next section, we will consider the F5 algorithm and its modifications F5R and F5C.

The most common upper estimate for runtime and memory complexity of Buchberger's algorithm was found by Thomas Dube in 1990 [10]. That research was published in a paper called "The Structure of Polynomial Ideals and Gröbner Bases". Dube proved that the degrees of the elements of a Gröbner basis are always bounded by

$$2\left(\frac{d^2}{2} + d\right)^{2^{n-1}}$$

where n is the number of variables, and d is the maximal total degree of the input polynomial equations. This shows us that the Buchberger's algorithm has a double exponential complexity

$$d^{2^{n+o(1)}}$$

On the other hand, there are examples where the basis contains of degree

$$d^{2^{\Omega(n)}}$$

Nevertheless, such examples are extremely rare. Therefore, this estimate also is a lower estimate and cannot be reduced. In other words, even the most powerful algorithms can use the enormous amounts of time and memory for calculations. But this case is extremely rare. In addition, in the case of the generators with the small degree, good estimations for the Gröbner basis elements degree were obtained.

Chapter 2. The algorithms for the Gröbner basis computation, based on signatures.

As previously stated, the most-time consuming operation used by the Buchberger's algorithm is the reduction of the S-polynomial. Moreover, we are interested only in those pairs of polynomials, the S-polynomials of which results into the nonzero polynomial after reduction. Such pairs are called the critical pairs. Since the necessary condition for the Gröbner basis is a reduction to zero for all possible S-polynomials, we see that the most of the computations are useless and redundant. Therefore, we must be able to recognize the reductions of S-polynomials to zero before they are computed and reduced. One of such criterion is already been described in the improved Buchberger's algorithm.

In this section, we will consider the F5 algorithm, developed in 2002 by Jean-Charles Faugere ([13]). The criterion of this algorithm is considered as the one of the most effective. The F5 for its work so-called polynomial signatures.

Signatures and labeled polynomials.

Let $F = (f_1, \dots, f_m)$ be the ordered collection of the polynomials from $K[x_1, \dots, x_n]$ with the defined monomial ordering. Therefore, $F \in K[x_1, \dots, x_n]^m$. Let denote T as the set of all possible monomials.

Let denote F_i as the i-th unit vector in the $K[x_1, \dots, x_n]^m$. The vector $g = (g_1, \dots, g_m)$ is called the syzygy if

$$\sum_{i=1}^{m} g_i f_i = 0$$

On the other hand,

$$g = \sum_{i=1}^{m} g_i F_i$$

Let's define the new ordering in the $K[x_1, \dots, x_n]^m$ in the following way:

$$\sum_{k=i}^{m} g_k F_k \prec \sum_{k=j}^{m} h_k F_k \Leftrightarrow \begin{cases} i > j \wedge h_j \neq 0 \\ i = j \wedge LM(g_i) < LM(h_i) \end{cases}$$

We note that $F_i \prec F_j$ if $i > j$.

For all $g \in K[x_1, \dots, x_n]^m$ there is an index i such that $g = \sum_{k=i}^{m} g_k F_k$ and $g_i \neq 0$. We define the new function $index(g)$, the value of which is equal to this index.

Further, according to the new ordering, $LM(g) = (g_i)F_i$. Let T_i be the set $\{tF_i \vee t \in T\}$. The union of T_i over all i is the set of all indexes of polynomials from the ideal $\langle F \rangle$.

Let t be the monomial. We denote as $W(t)$ the following set:

$$\left\{g \in K[x_1, \dots, x_n]^m \vee LM\left(\sum_{i=1}^{m} g_i f_i\right) = t\right\}$$

The following statements holds:

Statement 2.1. Let $w(t)$ be the minimum element of $W(t)$. If the monomials t_1 и t_2 are different, then $LM\big(w(t_1)\big) \neq LM\big(w(t_2)\big)$.

Statement 2.2. Let define the new function $v(p) = LM\left(w\big(LM(p)\big)\right)$, where p is the polynomial from the ideal $\langle F\rangle$. If the polynomials p_1 и p_2 of the ideal $\langle F\rangle$ have different leading monomials, then $v(p_1) \neq v(p_2)$.

Definition 2.1. The function $v(p) = LM\left(w\big(LM(p)\big)\right)$ is used by F5-family algorithms as the polynomial signature.

Definition 2.2. The labeled polynomial is the pair $r = (tF_i, f)$, where the second element is the original polynomial and first element is the signature of that polynomial.

Example 2.1. Suppose that we have an ideal $I = \langle g_1, g_2\rangle \subset Q[x, y, z]$, where $g_1 = xy - z^4$, $g_2 = y^2 - z$. We use the lexicographical monomial ordering. In this case, the signatures of g_1 and g_2 are respectively (1,1) and (1,2).

Let's define the following operations with signatures and labeled polynomials, which will be used further in the paper:

1) The body of the labeled polynomial: $poly(r) = f$
2) The signature of the labeled polynomial: $S(r) = tF_i$
3) The index of the labeled polynomial: $index(r) = i$
4) The leading monomial of the labeled polynomial: $LM(r) = LM(f)$
5) The leading coefficient of the labeled polynomial: $LC(r) = LC(f)$
6) The reduction of the labeled polynomial by the set of polynomials G: $\acute{r}^G = \left(S(r), \acute{f}^G\right)$
7) The multiplication of the labeled polynomial with the nonzero element of the field$\lambda \in K$: $\lambda r = (tF_i, \lambda f)$
8) The multiplication of the labeled polynomial with the monomial u: $ur = (utF_i, uf)$
9) The multiplication of the signature $w = tF_i$ with the monomial v: $vw = (vt)F_i$

The labeled polynomial is called the admissible, if there is $g = (g_1, \dots, g_m)$ such that $LM(g) = S(r)$.

We denote the set of all labeled polynomials as R.

The F5 algorithm.

The F5 algorithm was developed by Jean-Charles Faugere in 2002. This algorithm is incremental, it sequentially calculates the Gröbner bases of the collections $(f_m), (f_{m-1}, f_m), (f_{m-2}, f_{m-1}, f_m), \dots, (f_1, \dots, f_m)$. The algorithm doesn't use the Buchberger's criteria to recognize the critical pairs of polynomials. Instead of them, it uses its own criteria.

Let's introduce the following definitions, which are very important for the algorithm.

Definition 2.3. Let P be the finite subset of R and let s, t be the labeled polynomials, for which $poly(s) = f$ and $poly(t) = g$, where $f, g \neq 0$. If f can be represented in the form

$$f = \sum_{p \in P} \mu_p poly(p)$$

where for any $p \in P$ with the nonzero body the following condition holds:

$$LM(\mu_p)LM(p) \leq LM(t) \text{ and } LM(\mu_p)S(p) \leq S(s)$$

then this representation is called the t-representation of the labeled polynomial s with respect to P. This property can be shown by $f = O_P(t)$. If there is another labeled polynomial t' such that $LM(t') < LM(t)$, $S(t') \leq S(t)$ and $f = O_P(t')$, then this property can be shown by $f = o_P(t)$.

Definition 2.4a. The labeled polynomial r with the signature tF_k is normalized with respect to the collection of polynomials $F = (f_1, \dots, f_m)$, if $t \notin LM\langle F \rangle$.

Definition 2.4b. The pair of a labeled polynomial and a monomial is normalized with respect to the collection of polynomials, if the product of pair elements is normalized.

Definition 2.4c. The pair of labeled polynomials (r_i, r_j) is normalized with respect to the collection of polynomials, if the following holds:

1) $S(r_j) < S(r_i)$
2) The pairs (u_i, r_i) and (u_j, r_j) are normalized, where $w_{i,j} = LCM\left(LM(r_i), LM(r_j)\right)$, $u_i = \frac{w_{i,j}}{LM(r_i)}$ и $u_j = \frac{w_{i,j}}{LM(r_j)}$.

The S-polynomial of the normalized pair of labeled polynomials (r_i, r_j) is the following labeled polynomial:

Example 2.2. In our above example

$$g_3 = S(g_2, g_1) = xg_2 - yg_1 = x(y^2 - z) - y(xy - z^4) = -xz + yz^4$$
$$g_4 = S(g_3, g_1) = yg_3 + xg_1 = y(-xz + yz^4) + z(xy - z^4) = y^2z^4 - z^5$$
$$S(g_3) = xS(g_2) = x(1,2) = (x, 2)$$
$$S(g_4) = yS(g_3) = y(x, 2) = (xy, 2)$$

The F5 algorithm considers only the normalized pairs of labeled polynomials (r_i, r_j), for which $S(u_j r_j) < S(u_i r_i)$, so that it uses the defined ordering. Also, we see that the algorithm uses only syzygies, which are generated by S-polynomials.

Using the introduced terminology, we formulate the new criterion. The F5 algorithm uses it to discard unnecessary pairs of labeled polynomials.

Theorem 2.1. Let $F = (f_1, \dots, f_m)$ be the ordered collection of polynomials and let $R = (r_1, \dots, r_s)$ be the collection of admissible labeled polynomials such that $F \subset poly(R)$. Suppose, that for any normalized pair $(r_i, r_j) \in R^2$

$$S\left(poly(r_i), poly(r_j)\right) = o_{poly(R)}(u_i r_i)(0)$$

where $u_i = \frac{LCM\left(LM(poly(r_i)), LM\left(poly(r_j)\right)\right)}{LM(poly(r_i))}$. In this case the set $G = poly(R)$ is the the Gröbner basis of the ideal, generated by F.

This criterion is basic, but not unique in the F5. In addition, the algorithm create rules at the process of calculation, according to which new labeled polynomials can be expressed from the old ones, multiplied by some monomials. If the resulting S-polynomial can be expressed from either of two arguments, then the pair is discarded.

Thus, in a simpler language, the F5 algorithm uses two following criteria:

1) F5 Criterion: the S-polynomial $S(p, q) = LC(q)u_p p - LC(p)u_q q$ can be skipped at the Gröbner basis computation during the iteration l if:
   a) With representation of $S(p) = tF_l$ there is $g \in G_{l-1}$ such that $LM(g) \vee u_p t$.
   b) With representation of $S(q) = tF_l$ there is $g \in G_{l-1}$ such that $LM(g) \vee u_q t$.
2) Rewritten Criterion: the S-polynomial $S(p, q) = LC(q)u_p p - LC(p)u_q q$ can be skipped at the Gröbner basis computation during the iteration l if:
   a) With representation of $S(p) = tF_l$ there is $g \in G_{l-1}$, which was computed after p such that $S(g)=vF_l$ and $v \vee u_p t$.
   b) With representation of $S(q) = tF_l$ there is $g \in G_{l-1}$, which was computed after p such that $S(g)=vF_l$ and $v \vee u_q t$.

Example 2.3. In our above example $S(g_4) = S\big(S(g_3, g_1)\big) = (xy, 2)$ and $LM(g_1) = xy$. The monomial xy divides xy, so we don't need to compute g_4, because it will reduce to 0.

We have to note that the termination of the F5 algorithm is proven only for the regular sequences of polynomials.

Definition 2.5. The sequence of polynomials$F = (f_1, \dots, f_m) \in K[x_1, \dots, x_n]^m$ is regular if for any i such that $1 \le i < m$

$$\forall g \in K[x_1, \dots, x_n] g f_i \in \langle f_{i+1}, \dots, f_m \rangle \Rightarrow g \in \langle f_{i+1}, \dots, f_m \rangle$$

It's hard enough to recognize whether the sequence of polynomials is regular. For example, it's known that any overdetermined system of equations is not regular. It causes certain difficulties when we work with systems of equations in finite fields, because we use there the additional equations in the form $x^q + x = 0$. The F5 algorithm has an indicator to determine whether the sequence of polynomials is not regular. It is the reduction operation, resulting with the zero polynomial. Faugere proved that in the case of regular sequence all reduction operations during the Gröbner basis will result with nonzero polynomials, which can be used for generation of the new elements of the basis. Also, Faugere notes that the algorithm can be slightly changed in order to guarantee its termination for the nonregular sequences of polynomials.

We don't write the pseudocode of the algorithm here, because it is too big. It's fully available in the Faugere's and Stegers' works. Instead of this, we will make a brief overview of its main procedures and functions:

1) IncrementalF5 – the main body of the algorithm. The input is the sequence of polynomials $F = (f_1, \dots, f_m) \in K[x_1, \dots, x_n]^m$. The output is the Gröbner basis $G = \langle F \rangle$. The procedure at the start of the execution initiates the lists of rules for expressing new polynomials from the old ones. After that it makes m iterations, on each of which it performs calculation of the intermediate Gröbner bases $G_i = \langle f_{m+1-i}, \dots, f_m \rangle$.
2) AlgorithmF5 – performs calculation of the intermediate Gröbner bases, using the following sequence of actions:
    - Generation of the list of critical pairs of labeled polynomials
    - Generation of the list of S-polynomials, using the critical pairs with the smallest degree
    - Reduction of S-polynomials, using the previously calculated Gröbner basis.
    - Generation of the new list of critical pairs, using the results of reduction of S-polynomials

    The procedure repeats the last three steps until all possible critical pairs of labeled polynomials are processed.
3) CritPair – generates the critical pair of polynomials, discarding those that are not accepted by the F5 criterion.
4) Spol – generates S-polynomials, using the list of critical pairs and discarding those that are not accepted by the Rewritten criterion. Also, it makes expression rules for new polynomials.
5) Reduction – performs reduction of S-polynomials using the previously calculated Gröbner basis and some checks, based on the F5 and Rewritten criteria.

6) AddRule – is called every time, when the new S-polynomal is generated, and adds the new expression rule in the existing list of rules.
7) Rewritten? – checks the Rewritten criterion.

The F5R algorithm.

Despite the fact that the F5 algorithm is quite efficient, it has a significant drawback. It doesn't make reduction of intermediate Gröbner bases. As a result, the intermediate bases have too many redundant elements by the end of the algorithm. Consequently:

1) The algorithm spends more time on polynomial reduction operations, because it have to check more divisors.
2) A much larger number of available S-polynomials is generated.

Therefore, the question of using the reduced intermediate Gröbner bases in the F5 algorithm appeared. The main difficulty was that it was necessary to come up with mechanism for tracking the signatures of polynomials in reduced bases. One of the versions of the improved algorithm was proposed by Till Stegers in 2005 ([15]) and was called F5R. The difference from the F5 is that after each iteration the new intermediate Gröbner basis G_i is reduced to the basis B_i. This reduced basis does not affect the polynomial signatures and is used only for the reduction operations, while G_i continues to be used to generate new S-polynomials. The reduction operations are used in the two subalgorithms of the F5: CritPair and Reduction. This approach significantly reduces the number of reduction operations, but the number of considered S-polynomials remains the same.

The F5C algorithm.

The F5C algorithm was presented by John Perry and Christian Eder in 2009 ([11]). Like F5R, it uses reduced versions of intermediate Gröbner bases for computations. But now the reduced bases are used not only for S-polynomial reduction operations, but for the generation of new critical pairs of labeled polynomials. The F5C solved the problem of signature tracking of elements of the reduced basis. In the F5C algorithm expression rules for labeled polynomials are cleared after each calculation of the intermediate Gröbner basis and then recalculated for the elements of the reduced basis, using the new SetupReducedBasis procedure:

1) Let G_i be the intermediate Gröbner basis and result of the i-th iteration of the algorithm. Firstly, the reduced Gröbner basis B_i is calculated.
2) The F_{i+j-1} signature is defined for each element b_j of the reduced basis B_i.

3) All expression rules are cleared.
4) The expression rules are added for each possible S-polynomial in the basis B_i. Let (b_j, b_k) be the considered pair of polynomials. Then, the expression rule will have the form $(uF_{i+j-1}, 0)$, where $u = \frac{LCM(LM(b_j),LM(b_k))}{LM(b_j)}$ and 0 is the zero polynomial signature.

   These expression rules, which use the zero polynomial signature, are preferable to any other rules, so they will be checked first at the Rewritten criterion execution.

Nowadays, the F5C algorithm is one of the most efficient algorithms for Gröbner bases calculation. While the usual F5 algorithm is able to find bases for systems of the types Cyclic-8 and Cyclic-9 efficiently, it is possible to get the basis of the previously intractable Cyclic-10 system

$$C_{10}(x) = \left\{ \sum_{j=0}^{9} \prod_{k=j}^{j+i-1} x_{kmod10} = 0, i = 1,10 \right\}$$

quickly, using the F5C algorithm ([11]).

Chapter 3. Solving systems of equations in F_q.

The Buchberger's method.

Suppose, we have a system of equations

$$\begin{cases} f_1 = 0 \\ f_2 = 0 \\ \dots \\ f_s = 0 \end{cases}$$

where $f_1, \dots, f_s \in F_q[x_1, \dots, x_n]$. Let $F = (f_1, \dots, f_s)$ be the ordered collection of polynomials from $F_q[x_1, \dots, x_n]$ and let the lexicographical monomial ordering be defined. For example, in the case of two variables it will be like this:

$$1 < x_2 < x_2^2 < x_1 < x_1x_2 < x_1x_2^2 < x_1^2 < x_1^2x_2 < x_1^2x_2^2 < \cdots$$

We also define this ordering for leading terms of polynomials and products of them with elements of F_q. Suppose that each polynomial in the system is defined in the form:

$$f = \mathrm{c}_{\alpha_1}x^{\alpha_1} + \cdots + \mathrm{c}_{\alpha_m}x^{\alpha_m},\ \mathrm{c}_{\alpha_i} \in F_q,$$

where all multidegrees are arranged in the decreasing order $_1 > \ _2 > \cdots > \ _m$. Now we can determine all leading coefficients, leading monomials and leading terms.

$$LC(f) = c_{\alpha_1},$$

$$LM(f) = x^{\alpha_1},$$

$$(f) = \mathrm{c}_{\alpha_1}x^{\alpha_1}.$$

Let $I = \langle f_1, \dots, f_n \rangle$ be an ideal of the system. Let's use algorithm for the Gröbner basis calculation on this system. As a result, we will obtain a new system G

$$\begin{cases} g_1 = 0 \\ g_2 = 0 \\ \dots \\ g_s = 0 \end{cases}$$

which is in triangular form and possibly contains an equation of one variable x_iX. If we solve this equation, for example, using the Berlekamp algorithm [4], we can substitute the obtained values of x_i to G and repeat the process again until the system is successfully solved.

The question appears, why we always obtain the Gröbner basis in the triangular form. This is a corollary from the elimination theory. Let's introduce the following definition.

Definition 3.1. Let $I = \langle f_1, \dots, f_s \rangle \subset K[x_1, \dots, x_n]$. In that case, the ideal $I_l = I \cap K[x_{l+1}, \dots, x_n] \subset K[x_{l+1}, \dots, x_n]$ is called an l-eliminating ideal.

In fact, this ideal consists of all corollaries of the system $f_1 = \cdots = f_s = 0$ which depend on only $x_{l+1}, \dots, x_n$ variables. The ideal I is equal to I_0. If we change the order of variables, we obtain the different l-eliminating ideals. Therefore, in order to eliminate the $x_1, \dots, x_l$ variables,

we need to find nonzero polynomials in the l-eliminating ideal I_l. So, we need to get elements of I_l somehow. We can do this, using the following theorem.

Theorem 3.1 (about elimination, [8]). Let $I \subset K[x_1, \dots, x_n]$ be an ideal and G be a Gröbner basis of I with respect to the lexicographical monomial ordering $x_1 > x_2 > \cdots > x_n$. In that case for each l from 0 to n the set $G_l = G \cap K[x_{l+1}, \dots, x_n]$ is a Gröbner basis of l-eliminating ideal I_l.

Therefore, the Gröbner basis building process in the case of lexicographical monomial ordering sequentially eliminates all variables and the result always be in a triangular form.

It's not necessary to use the lexicographical monomial ordering to obtain such result. In order to have it, we can use any "l-elimination" ordering. This ordering is the ordering such that any monomial, which contains any of the $x_1, \dots, x_l$ variables is greater than any monomial from $K[x_{l+1}, \dots, x_n]$. There is the more general version of Theorem 7, based on the any "l-elimination" ordering, including the lexicographical monomial ordering.

The above method of solving nonlinear systems of equations terminates either by finding all the values of the variables of the system, or by constructing the set of equations G in a triangular form that does not contain a univariate equation. The following statement holds: the system doesn't have a solution if and only if the corresponding Gröbner basis contains a nonzero constant.

Hence, this method is a nonlinear generalization of the Gaussian method, which finds solutions of systems of linear equations. We have to note that the use of the lexicographical monomial ordering results with the Gröbner basis with the highest degrees of polynomials and time-memory consumption, if we compare it with the other monomial orderings.

The example of solving a system of equations in F_2.

In the coding theory or cryptography, we often have to deal with equations with coefficients in finite fields. Suppose that we need to solve the following equation to break a cryptosystem:

$$x^9 + \alpha x + \alpha^{13}.$$

The roots of this equation are in the F_{16}. The field F_{16} is considered as a quotient ring $F_2[x] \vee I$, where I is an ideal generated by the irreducible polynomial $f(x) = x^4 + x + 1 \in F_2[x]$. The elements of the field can be represented either in the power basis $A = (1, \alpha, \alpha^2, \alpha^3)$, or in the normal basis $B = (\beta, \beta^2, \beta^4, \beta^8)$ where $\beta = \alpha^3$, α is a primitive element of F_{16}.Hence, in the normal basis

$$\forall x \in F_{16} x = x_0\beta + x_1\beta^2 + x_2\beta^4 + x_3\beta^8,$$

$$\alpha = \beta + \beta^8,$$

$$\alpha^{13} = \beta + \beta^4 + \beta^8$$

Substituting these values in the original equation, we get the following system of nonlinear quadratic equations of the variables $x_0, x_1, x_2, x_3 \in F_2$:

$$x_0x_1 + x_1x_2 + x_1x_3 + x_1 + x_2x_3 + x_3 = 1$$
$$x_0x_2 + x_0x_3 + x_0 + x_1x_2 + x_1 + x_2x_3 + x_3 = 0$$
$$x_0x_1 + x_0x_3 + x_1x_3 + x_1 + x_2x_3 + x_2 + x_3 = 1$$
$$x_0x_1 + x_0x_2 + x_0x_3 + x_1x_2 + x_1 + x_2 = 1$$

Also, we add to these 4 equations another ones, which overdetermine the system with the properties of elements of F_2.

$$x_i^2 + x_i = 0, i = 0..3$$

Using the lexicographical monomial ordering $x_0 > x_1 > x_2 > x_3$, we obtain the Gröbner basis of the system:

$$x_2^2 + x_2 = 0$$
$$x_3^2 + x_3 = 0$$
$$x_0 + x_3 = 1$$
$$x_1 = 1$$
$$x_2x_3 + x_2 + x_3 = 1$$

Now we can easily obtain three solutions of the original equation:

- $x_0 = 1, x_1 = 1, x_2 = 1, x_3 = 0 \Longrightarrow x = \alpha^3 + \alpha^6 + \alpha^{12} = \alpha^7$
- $x_0 = 0, x_1 = 1, x_2 = 0, x_3 = 1 \Longrightarrow x = \alpha^6 + \alpha^9 = \alpha^5$
- $x_0 = 0, x_1 = 1, x_2 = 1, x_3 = 1 \Longrightarrow x = \alpha^6 + \alpha^{12} + \alpha^9 = \alpha^{14}$

Practical part.

As a practical part of this thesis, all above mentioned algorithms for the Gröbner basis calculation in different algebraic fields (C, Q, F_p, F_{2^n}) were implemented in the form of class library:

1) The original Buchberger's algorithm
2) The improved Buchberger's algorithm.
3) The Faugere's F5 algorithm.
4) The Stegers' F5R algorithm.
5) The Eders' and Perry's F5C algorithm.

The C# programming language was used for the implementation. The whole code is available in the Appendix. Here we will describe only the structure and the main parts of the library.

Library's description.

All main classes of the implemented library can be divided into the following groups:

1) Field's element classes – their objects are the elements of various algebraic fields. We have implemented four different classes for complex numbers, rational numbers, elements of the simple finite field and elements of the arbitrary finite field with characteristic 2.
2) The classes of polynomials – their objects are the polynomials with coefficients in the determined algebraic field. All these classes use the general monomial class.
3) The Gröbner basis calculation's modules – these static classes implements all above mentioned algorithms, which computes the Gröbner bases. These algorithms use reduction operation for S-polynomials and various criteria for their discarding.

Field's element classes provides methods for all operations in algebraic fields. Also, we can compare elements using methods of the corresponding class:

1) FieldQ – represents rational numbers as a structure of two BigIntegers: numerator and denominator. The class operates with only irreducible numbers and uses the Euclidean algorithm to provide this property.
2) FieldC – represents complex numbers as a structure of two doubles: real and imaginary parts. In addition to operations in algebraic fields, the class supports methods to compute a conjugate number, an absolute value and argument. Also, we can find the roots of any degree using this class.

3) FieldF_P – represents elements of simple finite fields, where every operation is modulo a prime numbers. An object of the class is a structure of two integers: characteristic and value. The characteristic number is prime and defines the concrete finite field. Operations with elements from different fields are forbidden.
4) FieldF_2N – represents elements of arbitrary finite fields with characteristic two. Every element is a Boolean polynomial expressed in the standard basis. All operations are done modulo some primitive polynomial, which defines the fields. Objects of this class is structure of two bit arrays: the primitive polynomial and the value. Also, we can express the element of the fields, using different basis, for example, the normal one. Again, operations with elements from different fields are forbidden.

We defined the coefficients of polynomials, but we need to implement the monomials. This is provided by the Monom class. Monomials are representes as an array of integer degrees of variables. The class supports the following operations:

1) Finding the full degree of monomial
2) Multiplication of monomials
3) Division of one monomial by another
4) Checking of divisibility.

All classes of polynomials represents them as a sorted list of terms (pairs of a monomial and a coefficient) using the SortedList collection. For each monomial, we can obtain the corresponding coefficient using the indexer. Each class provides methods for:

1) All ring operations (addition, subtraction, multiplication of polynomials)
2) Adding new terms
3) Finding the multidegree
4) Finding the leading coefficient, the leading monomial and the leading term.
5) Simplification, i.e. the multiplication to the inverse of leading coefficient.

The way, with which an object of Polynomial class sorts its own list of monomials, is defined by the monomial orderings. They represented as comparer classes, which are used in operations with the list of monomials to maintain the required ordering. We made comparers for the following monomial orderings:

1) Lexicographical
2) Graded lexicographical
3) Graded reverse lexigographical
4) Inverse lexicographical

The “Grebner” module provides the implementations of original and improved Buchberger’s algorithms for every mentioned field. Also, it transforms the univariate equation

in the complex finite field with characteristic 2, like in the above example, to the system of equations in F_2. The main methods of the module are:

1) Mod – gets the remainder of division operation of polynomial by a collection of polynomial. This method uses an algorithm from the Theorem 1.1 and finds the normal form of the polynomial;
2) S_Pol – constructs the S-polynomial of the pair of polynomials.
3) MakeGrebnerBasisOrig – finds the Gröbner basis of an ideal, which is generated by collection of polynomials, using the original Buchberger's algorithm from the Theorem 1.4.
4) MakeGrebnerBasis – finds the Gröbner basis of an ideal, which is generated by collection of polynomials, using the improved Buchberger's algorithm from the Theorem 1.8.
5) Criteria – used for the calculations in MakeGrebnerBasis method, checks the Criteria from the Theorem 1.8.
6) ReduceBasis – do the reduction of the Gröbner basis, using the Lemma 1.4.
7) MakeSystem – transforms the univariate equation in the finite field F_{2^n} to the multivariate system of $2n$ equations and n variables in F_2, which can be solved later with the Gröbner basis computation. The variables x_i in the new systems are the coefficients in the representation of x in the normal basis. The example of this operation was presented in chapter 3.

The F5, F5R and F5C modules provides the implementations of corresponding algorithms for every mentioned field. They need the functionality from the "Grebner" module and use some auxiliary classes:

1) Signature class represents obviously the polynomial signatures. If the value of signature is equal to tF_i, where t is monomial and F_i is the i-th unit vector in $K[x_1, \dots, x_n]^m$, then the corresponding object of this class are the pairs of a monomial object, which represents t, and an integer value i. The class provides method for the multiplication with a monomial operation. Also, the class is able to order signatures, using defined comparers.
2) The classes of labeled polynomials represents them as a pair of polynomial and a signature. These pairs can be ordered by their signatures, using different compares for monomial orderings.
3) The critical pair class is the static class, which is able to determine the useless pair of labeled polynomials using the F5 criterion described in Chapter 2.

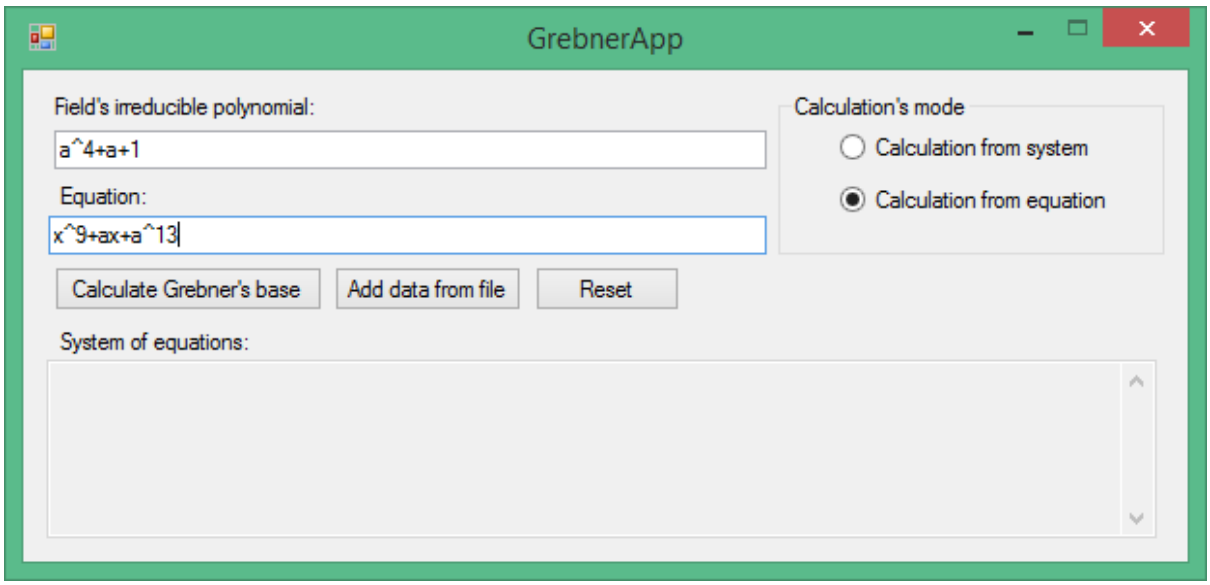
4) RuleF5 class objects represents the expression rules, which are used for determining the useless pairs of labeled polynomials using Rewritten criterion described in chapter 2. In our library, a rule (t, F_i) is just a pair of a monomial t and an index i.
5) F5, F5R and F5C static classes do the computations of corresponding algorithms. Each of these classes have implementations of all submethods, which were mentioned in Chapter 2, and is able to maintain its own list of expression rules to check the F5 and the Rewritten criteria.

Also, the desktop application with GUI was made for the Gröbner basis calculation in the field F_{2^n}. It works in two different modes. The first one calculates the Gröbner basis using the collection of polynomials. The second one calculates the Gröbner basis of the system of equations in F_2, which is obtained by transforming the input equation in F_{2^n}. Of course, in both modes we need to define the field with a primitive polynomial. All polynomials, equations and systems of them are defined by strings and parsed with regular expressions. Systems of equations can be passed to the program from the simple file with TXT extension. The application is configurated by the configuration file. This file contains the settings for:

1) Monomial ordering
2) Algorithm for the Gröbner basis calculation

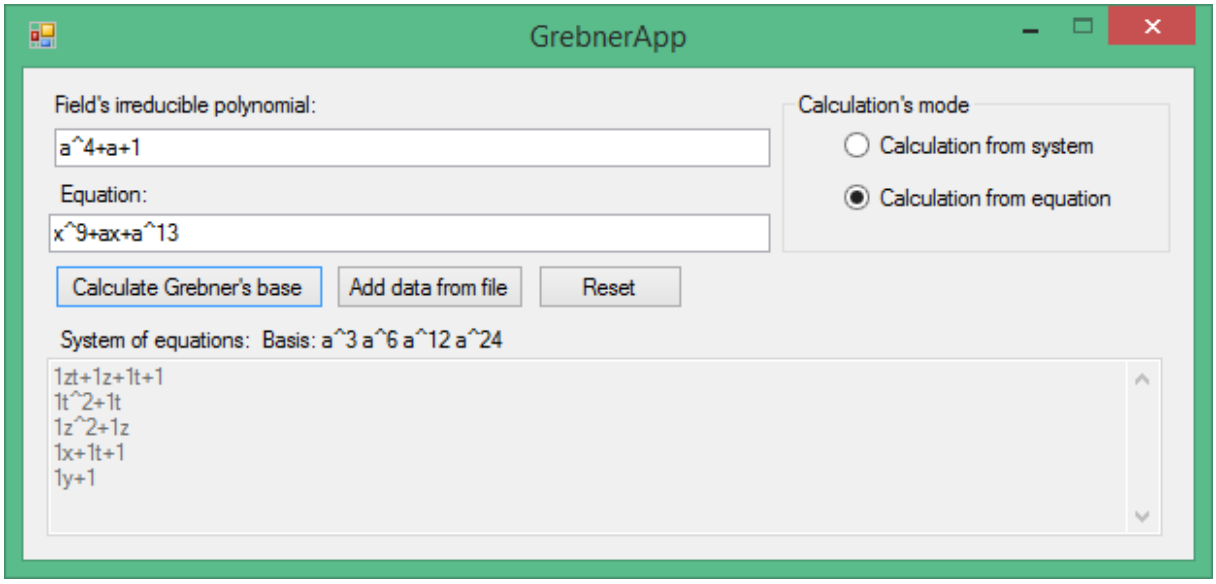
The views of the application:

1) The Gröbner basis calculation from the equation in F_{2^n}



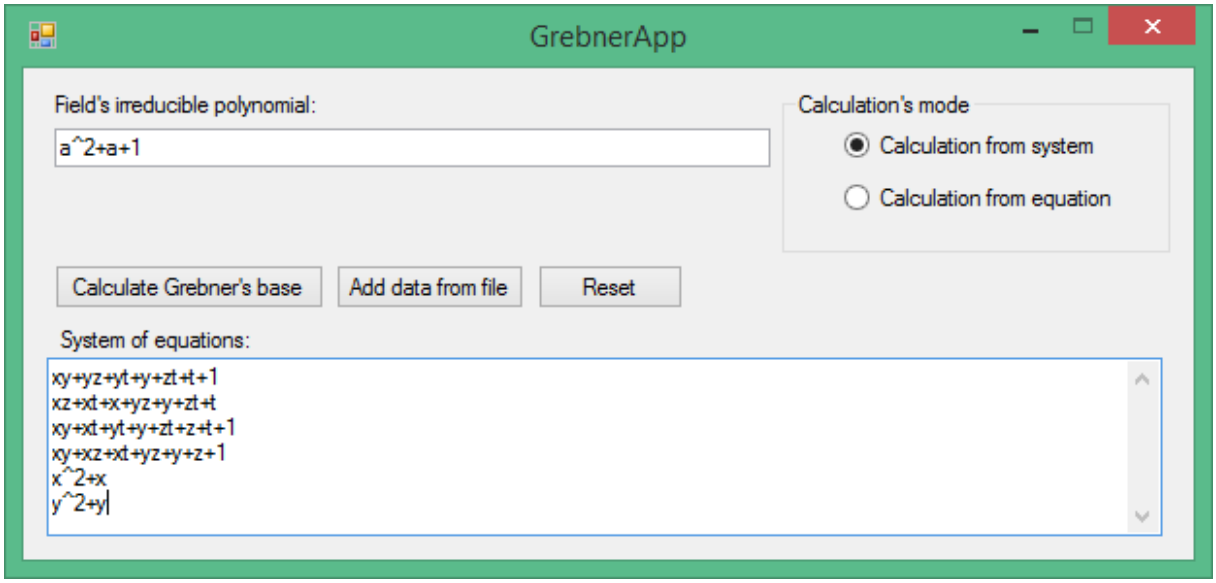
1. The GUI. The Gröbner basis calculation from the equation in F_(2^n)

2) Result of the calculation



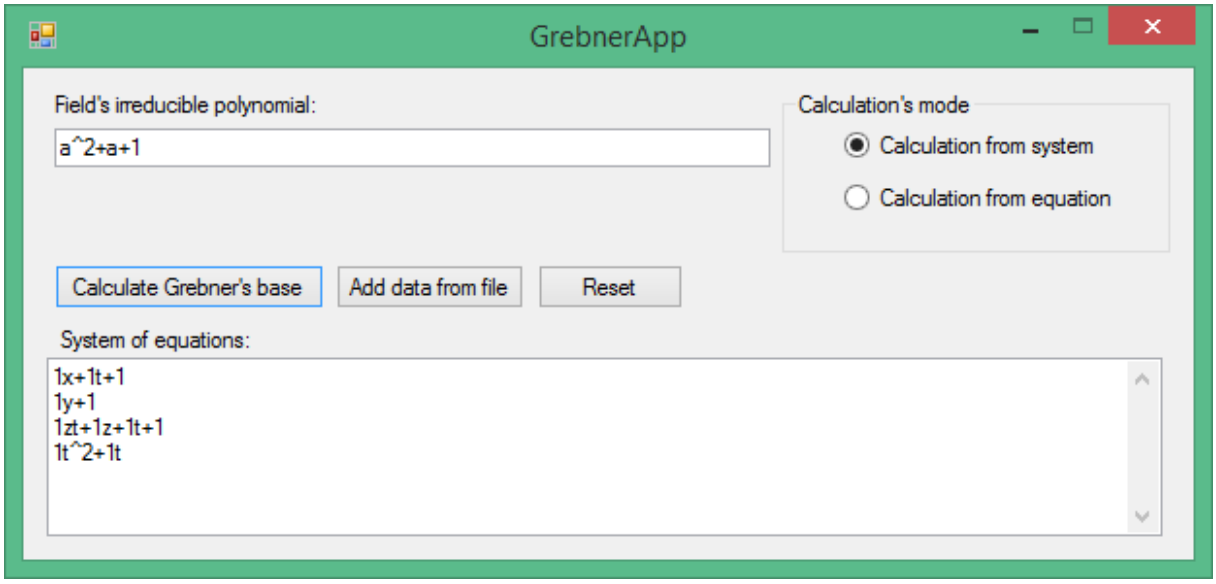
2. *The GUI. Result of the calculation in the second mode*

3) The Gröbner basis calculation from the system of equations in F_{2^n}



3. *The GUI. The Gröbner basis calculation from the system of equations in F_(2^n)*

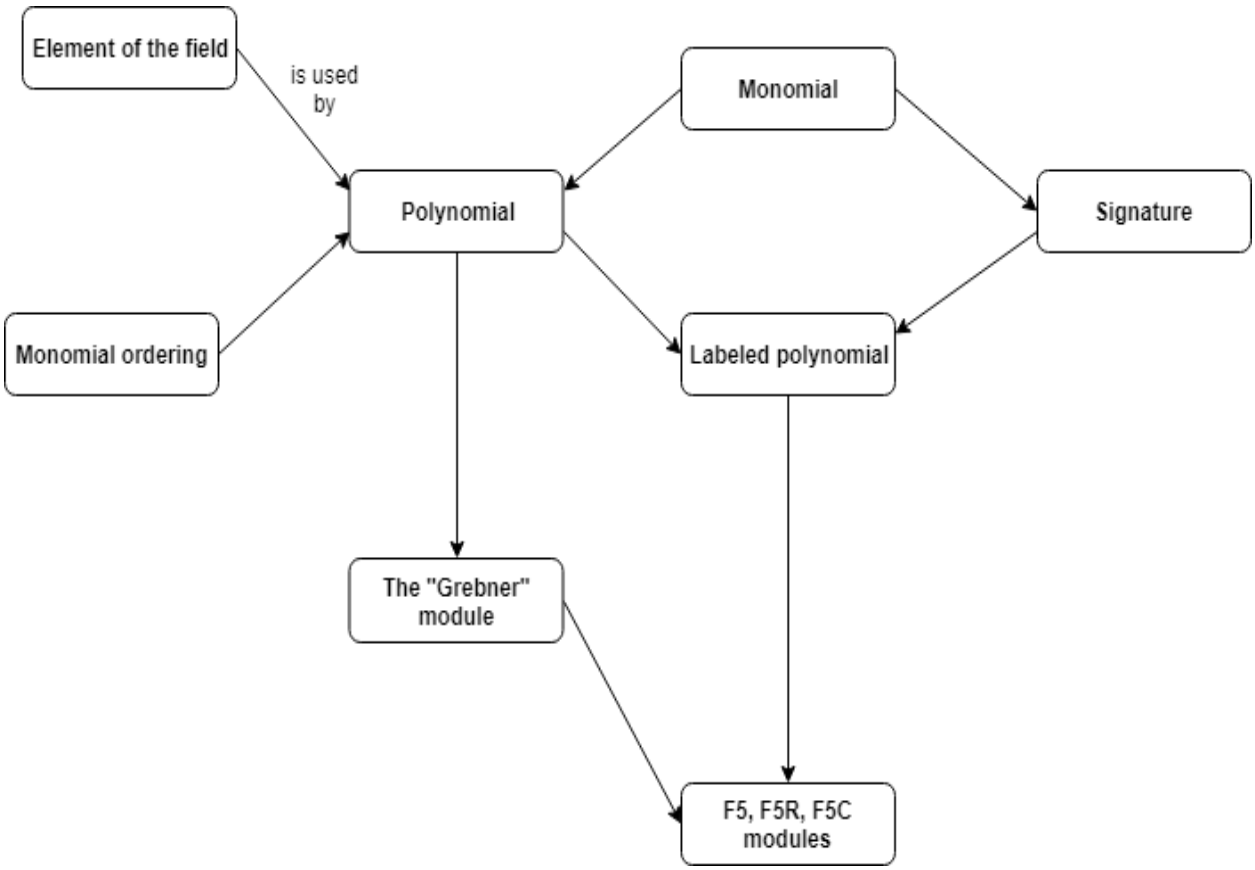
4) The result of calculation



4. *The GUI. Result of calculation in the first mode.*

As the final step of implementation, the unit tests were made for every important class, using various types of testing coverages, including Condition\Decision Coverage and Multiple Condition\Decision Coverage.

The whole class hierarchy used in the library is presented in the following picture:



5. *The class hierarchy used in the library*

Experimental results.

All experiments were held on the following configuration: Intel Core i5-2430M 2.4GHz, 4GB RAM, Windows 8.1.

The following measures of algorithm's efficiency were chosen:

1) The Gröbner basis computation time
2) The number of reduction operations, which were performed during computation. These operations are the most resource-consuming part of the algorithm.

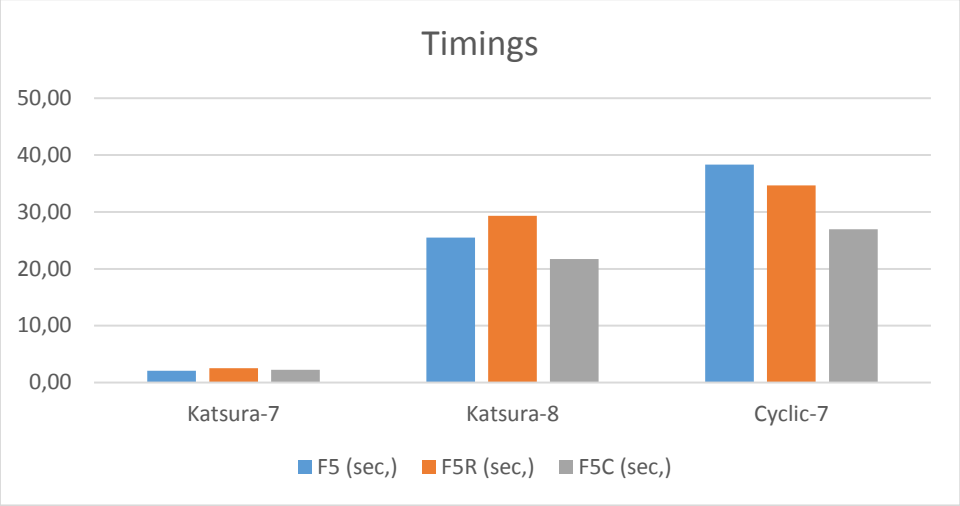
The second measure is more representative than the first one, because the issues of the implementation of structures and operations, which are involved during execution of an algorithm, affect the resulting computation time.

All experiments were held using systems of polynomials, coefficients of which were in the finite field $F_{2^{16}}$. The systems are the analogs of known named systems. Only F5-family algorithms were tested, because the other two algorithms are much slower.

All timings were taken with the help of Stopwatch object from System.Diagnostics namespace. The average of 4 measurements was considered as a result value. The results are shown in the following two tables and diagrams.

Table 1.Obtained timings of the signature-based algorithms for the Gröbner basis calculation.

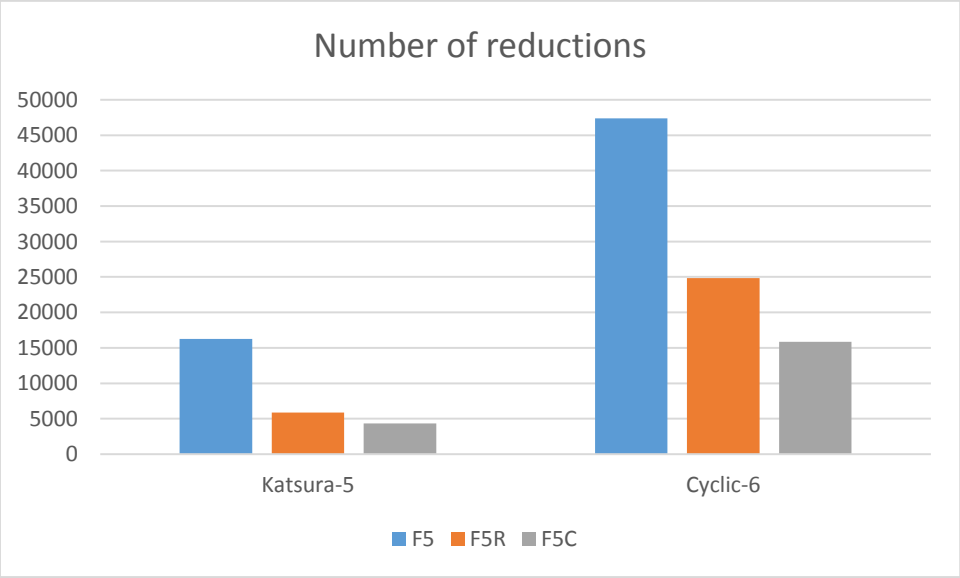
System	*F5 (sec.)*	*F5R (sec.)*	*F5C (sec.)*	*F5R/F5*	*F5C/F5*
Katsura-7	2.09	2.51	2.23	1.2	1.07
Katsura-8	25.47	29.32	21.73	1.15	0.85
Cyclic-6	0.28	0.24	0.22	0.86	0.78
Cyclic-7	38.32	34.64	26.96	0.9	0.7



6. Results. The diagram with timings.

Table 2.The number of reduction operations appeared during the Gröbner basis computation for the different signature-based algorithms.

System	*F5*	*F5R*	*F5C*
Katsura-4	824	335	272
Katsura-5	16273	5859	4326
Cyclic-5	530	507	474
Cyclic-6	47378	24820	15872



7. Results. The diagram with numbers of reductions.

We see that our results mostly coincide with results, obtained by Faugere and Perry. The most efficient algorithm is F5C, the second one is F5R , and the last place is taken by the classic F5. It's obvious that F5C is faster than both of F5R and F5, but F5R is faster than F5 only in the situation when the algorithm deals with the large amount of considered polynomials. This happens, because, as mentioned before, F5 and F5R both generate and use the same sets of S-polynomials, which is constructed using the same original redundant Gröbner bases. At the same time, we obtain that both F5R and F5C perform much less reduction operations than the classic F5. This is a consequence of the use of reduced intermediate bases, the size of which is much smaller than the original ones.

In general, we see that the use of reduced bases allows to significantly reduce the Gröbner basis computation time, in spite of the fact that the reduction of the basis is also a resource-consuming operation. Our implementation of algorithms for the Gröbner bases calculation provides the same performance characteristics as the other known implementations (Magma, Singular, Maple, Mathematica), and also is able to operate with polynomials, coefficients of which are the elements of arbitrary finite field with characteristic 2.

Conclusion.

In this paper, we have studied the problem of finding the solution of systems of nonlinear polynomial equations. Firstly, we reviewed widely known methods for solving such systems. Their main features and fields of application were indicated. In addition to, we also studied the complexity of these methods.

Secondly, we gave a definition the Gröbner basis of a polynomial ideal. To do this, we needed to introduce the notion of a monomial ordering and show that all polynomial ideals are finitely generated with the help of the Dickson lemma and the Hilbert basis theorem. Also, in order to construct these bases, we introduced an algorithm for dividing multivariate polynomials in the polynomial ring. Introducing the definition of the S-polynomial, we were able to give a clear definition of the bases whose name gave the name to the whole thesis. Also, we described some useful properties of these bases.

Thirdly, we indicated an algorithm for constructing the required Gröbner bases introduced by Buchberger. This algorithm allowed us to derive the Gröbner basis of an ideal generated by any set by a polynomial in a finite number of steps. This algorithm allowed us to transform systems of linear equations to a form in which there are univariate equations, but these equations probably can have higher degrees. As follows from the paper, the complexity of this algorithm is determined by the huge number of calculations of the remainders from dividing by the collection of polynomials. Therefore, the only way to improve the algorithm is to reduce the number of S-polynomials under consideration.

Fourthly, we described and compared the F5-family signature algorithms for the Gröbner bases calculation. We gave a description of the criteria for excluding critical pairs based on the signatures of polynomials in the original F5 algorithm, introduced by Jean-Charles Faugere, and also showed how this algorithm can be improved by describing its modifications F5R and F5C, which were introduced by respectively Till Stegers and John Perry.

Fifthly, we described a method for solving systems of equations in finite fields by constructing the Gröbner basis. This method uses lexicographic monomial ordering, which allows us to build a basis in the form of a triangular system of equations, sequentially eliminating variables. The only inconvenience is that this ordering constructs a Gröbner basis with the highest powers of the elements of the basis.

As a practical part, we implemented in the C# programming language and tested 5 algorithms for the Gröbner bases calculation in all basic algebraic fields, including finite fields of characteristic 2, operations with which is not implemented in popular mathematical

applications. Also, the test results of signature algorithms for these fields was presented and analyzed.

We see that the given task has been fully accomplished. All necessary mathematical and algorithmic apparatus using Gröbner bases were given to solve the systems of equations in finite fields. The implementation of five algorithms for the Gröbner bases calculation was prepared, which can be used for tasks in finite fields of characteristic 2.

We know that the time and complexity of the Gröbner basis calculation strongly depend on the degree and number of variables in the system, and also on the used monomial ordering. It is widely known that this method has extremely high complexity in the worst case. However, we also know that the probability of such a case is extremely low, and most algorithms for the Gröbner bases calculation are much faster. With the new efficient algorithms such as F5, F5C and the further modifications of Buchberger's algorithm, it is hoped that the widespread opinion of Gröbner bases as a complex, inconvenient and labor-intensive means of solving systems of polynomial equations will change, and this pessimism will disappear.

References

[1] Agibalov G. P., 2006. Metody resheniya sistem polinomial'nykh uravneniy nad konechnym polem [Methods for solving systems of polynomial equations over a finite field]. Vestnik TSU. Prilozhenie, no. 17, pp. 4-9. (in Russian)

[2] Arri, A. and Perry, J. The F5 Criterion revised, 2011. Journal of Symbolic Computation, 46(2):1017–1029, June 2011. Preprint online at arxiv.org/abs/1012.3664.

[3] Arghantsev I.V., 2002. The Gröbner bases and the systems of algebraic equations. ISBN 5-94057-095-X. (in Russian)

[4] Berlekamp, Elwyn R., 1968. Algebraic Coding Theory. McGraw Hill. ISBN 0-89412-063-8.

[5] Buchberger, Bruno, 1965. Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. Ph.D. Thesis, University of Innsbruck. (in German) English translation in Buchberger (2006).

[6] Buchberger, B., 1985. Gröbner bases: An algorithmic method in polynomial ideal theory. In: (Bose, N. K., ed.) Recent Trends in Multidimensional Systems Theft T, D. Reidel.

[7] Buchberger, Bruno, 2006. Bruno Buchberger's PhD thesis 1965: an algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. J. Symbolic Comput. 41 (3–4), 475–511. Translated from the 1965 German original by Michael P. Abramson.

[8] Cox, David; Little, John; O'Shea, Donal, 1997. Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra. Springer. ISBN 0-387-94680-2.

[9] Davenport, J., Siret,Y., Tournier, E., 1991. Computer algebra : systems and algorithms for algebraic computation. ISBN 978-0-12-204230-0 (in Russian)

[10] Thomas Dubé, 1990. The Structure of Polynomial Ideals and Gröbner Bases. SIAM J. Comput. 19(4): 750-773

[11] Eder, C., Perry, J. E., 2010. F5C: A Variant of Faugere's F5 Algorithm With Reduced Gröbner Bases. Journal of Symbolic Computation, 45(12), 1442-1458.

[12] Faugère, J.-C., 1999. A new efficient algorithm for computing Gröbner bases (F4). Journal of Pure and Applied Algebra 139 (1–3), 61–88

[13] Faugère, J.C., 2002. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In: ISSAC ’02: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation. ACM Press, New York, NY, USA, pp. 75–83.

[14] Lidl, Rudolf; Niederreiter, Harald, 1997. Finite Fields (2nd ed.), Cambridge University Press, ISBN 0-521-39231-4

[15] Stegers, T., 2006. Faugere’s F5 algorithm revisited. Cryptology ePrint Archive, Report 2006/404.

Appendix

The module of rational numbers RationalNumbersField.cs

```
using System;
using System.Numerics;

public class FieldQ: Field //field of rational numbers
{
    public BigInteger Numerator { get; set; }
    public BigInteger Denominator { get; set; }
    //constructors
    public FieldQ(BigInteger num, BigInteger denom)
    {
        if (denom == 0) { throw new DivideByZeroException(); }
        if (denom < 0) //the denominator must be the natural number
        {
            Numerator = -num;
            Denominator = -denom;
        }
        else
        {
            Numerator = num;
            Denominator = denom;
        }
        Shrink();
    }

    public FieldQ(BigInteger num) : this(num, 1) { }

    public FieldQ(FieldQ num) : this(num.Numerator, num.Denominator) { }

    public void Shrink() //Euclidean algorithm
    {
        BigInteger q = BigInteger.GreatestCommonDivisor(Numerator, Denominator);
        Numerator = Numerator / q;
        Denominator = Denominator / q;
    }
    //the conversion of an integer number to the rational one
    public static implicit operator FieldQ(BigInteger a) { return new FieldQ(a); }
    // the conversion of a rational number to the real one
    public static implicit operator double(FieldQ a) { return (double)a.Numerator /
(double)a.Denominator; }
    //addition
    public static FieldQ operator + (FieldQ a, FieldQ b)
    {
        return new FieldQ(a.Numerator * b.Denominator + a.Denominator * b.Numerator,
a.Denominator * b.Denominator);
    }
    //multiplication
    public static FieldQ operator * (FieldQ a, FieldQ b)
    {
        return new FieldQ(a.Numerator * b.Numerator, a.Denominator * b.Denominator);
    }
    //inverse for +
    public static FieldQ operator - (FieldQ a) { return new FieldQ(-a.Numerator,
a.Denominator); }
    //subtraction
    public static FieldQ operator - (FieldQ a, FieldQ b) { return a + (-b); }
    //inverse for *
    public FieldQ Inv() { return new FieldQ(Denominator, Numerator); }
    //division
```

```
    public static FieldQ operator / (FieldQ a, FieldQ b) { return a * b.Inv(); }
    //power to an integer degree
    public FieldQ Pow(int n)
    {
        BigInteger k1 = 1, k2 = 1;
        for (int i = 0; i < Math.Abs(n); i++)
        {
            k1 = k1 * Numerator;
            k2 = k2 * Denominator;
        }
        if (n >= 0) { return new FieldQ(k1, k2); } else { return new FieldQ(k1,
k2).Inv(); }
    }
    //absolute value
    public FieldQ Abs() { return new FieldQ(BigInteger.Abs(Numerator), Denominator); }
    //signature
    public int Sgn()
    {
        if (Numerator == 0) { return 0; }
        else if (Numerator > 0) { return 1; } else { return -1; }
    }
    //output
    public override string ToString()
    {
        string res = "";
        if (Denominator != 1)
        {
            res = res + Numerator + '/' + Denominator;
        }
        else res = res + Numerator;
        return res;
    }
    //comparisons
    public static bool operator== (FieldQ a, FieldQ b) { if ((a - b).Sgn() == 0) {
return true; } else { return false; } }

    public static bool operator> (FieldQ a, FieldQ b) { if ((a - b).Sgn() == 1) {
return true; } else { return false; } }

    public static bool operator< (FieldQ a, FieldQ b) { if ((a - b).Sgn() == -1) {
return true; } else { return false; } }

    public static bool operator!= (FieldQ a, FieldQ b) { return !(a == b); }

    public static bool operator<= (FieldQ a, FieldQ b) { return !(a > b); }

    public static bool operator>= (FieldQ a, FieldQ b) { return !(a < b); }

    public override bool Equals(object obj)
    {
        FieldQ Qobj = obj as FieldQ;
        if ((object)Qobj == null) return false;
        else
            return (Qobj == this);
    }

    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}
```

The module of complex numbers ComplexNumbersField.cs

```
using System;

public class FieldC : Field //the field of complex numbers
{
    public double Re { get; set; } //real part
    public double Im { get; set; } //imaginary part
    //constructors
    public FieldC(double Re, double Im)
    {
        this.Re = Re;
        this.Im = Im;
    }

    public FieldC(double Re) : this(Re, 0) { }

    public FieldC(FieldC num) : this(num.Re, num.Im) { }
    //conversion of a real number to the complex one
    public static implicit operator FieldC(double a) { return new FieldC(a); }
    //equality to 0
    public bool IsZero() { if ((Re == 0) && (Im == 0)) { return true; } else { return
false; } }
    //conjunction
    public static FieldC operator ~ (FieldC a) { return new FieldC(a.Re, -a.Im); }
    //addition
    public static FieldC operator + (FieldC a, FieldC b) { return new FieldC(a.Re +
b.Re, a.Im + b.Im); }
    //multiplication
    public static FieldC operator * (FieldC a, FieldC b)
    {
        return new FieldC(a.Re * b.Re - a.Im * b.Im, a.Im * b.Re + a.Re * b.Im);
    }
    //inverse for +
    public static FieldC operator - (FieldC a) { return new FieldC(-a.Re, -a.Im); }
    //subtraction
    public static FieldC operator - (FieldC a, FieldC b) { return a + (-b); }
    //absolute value
    public double Abs() { return Math.Sqrt(Re * Re + Im * Im); }
    //argument
    public double Arg() { return Math.Atan2(Im, Re); }
    //division
    public static FieldC operator / (FieldC a, FieldC b)
    {
        double abs = b.Abs();
        if (abs == 0) { throw new DivideByZeroException(); }
        return a * (~b) * (1 / (abs * abs));
    }
    //inverse for *
    public FieldC Inv() { return 1 / this; }
    //power to the integer degree
    public FieldC Pow (int n)
    {
        FieldC res = 1;
        for (int i = 0; i < Math.Abs(n); i++)
        {
            res = res * this;
        }
        if (n >= 0) { return res; } else { return res.Inv(); }
    }
    //the roots of the natural degree
    public FieldC[] Root (int n)
```

```
    {
        if (n <= 0) { throw new ArgumentOutOfRangeException(); }
        FieldC[] res = new FieldC[n];
        double abs = Math.Pow(Abs(), (double)1 / n);
        double arg0 = Arg(), arg;
        for (int i = 0; i < n; i++)
        {
            arg = (arg0 + 2 * Math.PI * i) / n;
            res[i] = new FieldC(abs * Math.Cos(arg), abs * Math.Sin(arg));
        }
        return res;
    }
    //output
    public override string ToString()
    {
        string res = "";
        if (Re != 0)
        {
            res = res + Re;
            if (Im > 0)
            {
                res = res + "+" + Im + "i";
            }
            if (Im < 0)
            {
                res = res + Im + "i";
            }
        }
        else
        {
            if (Im != 0) res = res + Im + "i";
        }
        return res;
    }
    //comparisons
    public static bool operator== (FieldC a, FieldC b) { return (a - b).IsZero(); }

    public static bool operator!= (FieldC a, FieldC b) { return !(a - b).IsZero(); }

    public override bool Equals(object obj)
    {
        FieldC Cobj = obj as FieldC;
        if ((object)Cobj == null) return false;
        else
            return (Cobj == this);
    }

    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}
```

The module of the elements of simple finite field SimpleGaloisField.cs

```
using System;

public class FieldF_P : Field //the simple Galois field
{
    public int Char { get; set; } //characterstic (prime number)
    public int Val { get; set; } //value
    //primarility test
    private bool IsSimple(int Char)
    {
```

```
        if (Char == 2) { return true; }
        bool res = true; int i = 2;
        do
        {
            if (Char % i == 0) { res = false; }
            i++;
        }
        while (res && (i <= Math.Sqrt(Char)));
        return res;
    }
    //constructors
    public FieldF_P(int Char, int Val)
    {
        if (!IsSimple(Char) || (Math.Abs(Val) >= Char)) { throw new
ArgumentException(); }
        this.Char = Char;
        if (Val >= 0) { this.Val = Val; } else { this.Val = Char + Val; }
    }

    public FieldF_P(FieldF_P num): this(num.Char, num.Val) { }
    //equality to 0
    public bool IsNull()
    {
        if (Val == 0) { return true; } else { return false; }
    }
    //addition
    public static FieldF_P operator+ (FieldF_P a, FieldF_P b)
    {
        if (a.Char != b.Char) { throw new ArgumentException(); }
        return new FieldF_P(a.Char,(a.Val + b.Val) % a.Char);
    }
    //multiplication
    public static FieldF_P operator * (FieldF_P a, FieldF_P b)
    {
        if (a.Char != b.Char) { throw new ArgumentException(); }
        return new FieldF_P(a.Char, (a.Val * b.Val) % a.Char);
    }
    //inverse for +
    public static FieldF_P operator - (FieldF_P a) { return new FieldF_P(a.Char,
(a.Char - a.Val) % a.Char); }
    //subtraction
    public static FieldF_P operator - (FieldF_P a, FieldF_P b) { return a + (-b); }
    //power to the integer degree
    public FieldF_P Pow(int n)
    {
        FieldF_P res = new FieldF_P(Char, 1);
        for (int i = 0; i < Math.Abs(n); i++)
        {
            res = res * this;
        }
        if (n >= 0) { return res; } else { return res.Inv(); }
    }
    //inverse for *
    public FieldF_P Inv()
    {
        if (Val == 0) throw new ArithmeticException();
        return Pow(Char - 2);
    }
    //division
    public static FieldF_P operator / (FieldF_P a, FieldF_P b)
    {
        if (a.Char != b.Char) { throw new ArgumentException(); }
        return a * b.Inv();
    }
```

```
    //output
    public override string ToString()
    {
        return ("(" + Char + "," + Val + ")");
    }

    public static bool operator ==(FieldF_P a, FieldF_P b)
    {
        try
        {
            return (a - b).IsNull();
        }
        catch (ArgumentException e)
        {
            return false;
        }
    }

    public static bool operator !=(FieldF_P a, FieldF_P b)
    {
        return !(a == b);
    }

    public override bool Equals(object obj)
    {
        FieldF_P Pobj = obj as FieldF_P;
        if ((object)Pobj == null) return false;
        else return (Pobj == this);
    }

    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}
```

The module of the elements of the finite field with characteristic 2 Vector2GaloisField.cs

```
using System;
using System.Collections;

struct FieldF_2NLeadTerm //the leading term of polynomial with coefficients in GF(2)
{
    public BitArray Sign; //signature
    public int Degree;
    public bool IsZero; //equality to zero
}

public class FieldF_2N : Field //Galois field with characteristic 2
{
    public BitArray Prim { get; set; } //primitive polynomial, which defines field
    public BitArray Val { get; set; } //value
    public bool GetVal(int i)
    {
        return Val[i];
    }
    public void SetVal(int i, bool bit)
    {
        Val[i] = bit;
    }
    //conversion of a number to the bit array
```

```
public static BitArray Convert(int a)
{
    if (a == 0) { return new BitArray(1); }
    int val, deg, ver;
    val = 1; deg = 0;
    while (a >= val)
    {
        val = val * 2;
        deg++;
    }
    bool[] res = new bool[deg]; val = val / 2; ver = val / 2;
    for (int i = deg - 1; i >= 0; i--)
    {
        if (a >= val)
        {
            res[i] = true;
            val = val + ver;
        }
        else
        {
            res[i] = false;
            val = val - ver;
        }
        ver = ver / 2;
    }
    return new BitArray(res);
}
//equality to zero polynomial
public static bool IsNull(BitArray p)
{
    bool found = false; int i = 0;
    while ((!found) && (i < p.Count))
    {
        if (p[i] == true) { found = true; }
        else { i++; }
    }
    return (!found);
}
//leading term of polynomial
private static FieldF_2NLeadTerm LT(BitArray p)
{
    bool found = false; int i = p.Count - 1;
    while ((!found) && (i >= 0))
    {
        if (p[i] == true) { found = true; }
        else { i--; }
    }
    FieldF_2NLeadTerm res = new FieldF_2NLeadTerm();
    res.Sign = new BitArray(p.Count);
    if (found)
    {
        res.IsZero = false;
        res.Degree = i;
        res.Sign[i] = true; }
    else
    {
        res.IsZero = true;
    }
    return res;
}
//multiplication of polynomials
private static BitArray Mult(BitArray p1, BitArray p2)
{
    BitArray res = new BitArray(p1.Count + p2.Count);
```

```
            for (int i = 0; i < p1.Count; i++)
            {
                for (int j = 0; j < p2.Count; j++)
                {
                    res[i + j] = res[i + j] ^ (p1[i] && p2[j]);
                }
            }
            return res;
        }
        //constructors
        public FieldF_2N(BitArray Prim, BitArray Val)
        {
            if (Prim.Count <= Val.Count) { throw new ArgumentException(); }
            this.Prim = Prim;
            this.Val = new BitArray(Prim.Count - 1);
            for (int i = 0; i < Val.Count; i++)
            {
                this.Val[i] = Val[i];
            }
        }

        public FieldF_2N(int prim, int val) : this(Convert(prim),Convert(val)) { }

        public FieldF_2N(FieldF_2N num) : this(num.Prim, num.Val) { }

        public int FieldDegree() { return Prim.Count - 1; }
        //is value equal to zero
        public bool IsNull()
        {
            return IsNull(Val);
        }
        //determines whether two elements belong to one field
        public bool FieldEQ(FieldF_2N num)
        {
            if (Prim.Count != num.Prim.Count) { return false; }
            bool found = false; int i = 0;
            while ((!found) && (i < Prim.Count))
            {
                if (Prim[i] != num.Prim[i]) { found = true; }
                else { i++; }
            }
            return (!found);
        }
        //addition
        public FieldF_2N Add(FieldF_2N b)
        {
            if (!FieldEQ(b)) { throw new ArgumentException(); }
            return new FieldF_2N(Prim, Val.Xor(b.Val));
        }
    //inverse to +
        public static FieldF_2N operator - (FieldF_2N a) { return a; }
        //subtraction
        public static FieldF_2N operator - (FieldF_2N a, FieldF_2N b) { return a.Add(b); }
        //multiplication modulo the primitive polynomial
        public static FieldF_2N operator * (FieldF_2N a, FieldF_2N b)
        {
            if (!a.FieldEQ(b)) { throw new ArgumentException(); }
            BitArray prod = Mult(a.Val, b.Val);
            BitArray res = prod, cur;
            FieldF_2NLeadTerm ltg = LT(a.Prim), ltr = LT(res);
            while ((!IsNull(res)) && (ltg.Degree <= ltr.Degree))
            {
                int k = ltr.Degree - ltg.Degree;
                cur = new BitArray(k + 1);
```

```
            cur[k] = true;
            cur = Mult(cur, a.Prim);
            for (int i = 0; i < cur.Count; i++)
            {
                res[i] = res[i] ^ cur[i];
            }
            ltr = LT(res);
        }
        res.Length = a.Prim.Count - 1;
        return new FieldF_2N(a.Prim, res);
    }
    //power to the integer degree
    public FieldF_2N Pow(int n)
    {
        FieldF_2N res = new FieldF_2N(Prim, new BitArray(Prim.Count - 1));
        res.Val[0] = true;
        for (int i = 0; i < Math.Abs(n); i++)
        {
            res = res * this;
        }
        if (n >= 0) { return res; } else { return res.Inv(); }
    }
    //inverse to *
    public FieldF_2N Inv()
    {
        if (IsNull()) throw new ArithmeticException();
        int n = FieldDegree();
        return Pow((int)Math.Pow(2, n) - 2);
    }
    //division
    public static FieldF_2N operator /(FieldF_2N a, FieldF_2N b)
    {
        if (!a.FieldEQ(b)) { throw new ArgumentException(); }
        return a * b.Inv();
    }
public override bool Equals(object obj)
    {
        FieldF_2N Pobj = obj as FieldF_2N;
        if (Pobj == null) return false;
        else
        {
            bool found = false; int i = 0;
            if (Val.Count != Pobj.Val.Count) return false;
            if (Prim.Count != Pobj.Prim.Count) return false;
            while ((!found) && (i < Val.Count))
            {
                if (Val[i] != Pobj.Val[i]) found = true;
                else i++;
            }
            i = 0;
            while ((!found) && (i < Prim.Count))
            {
                if (Prim[i] != Pobj.Prim[i]) found = true;
                else i++;
            }
            return !found;
        }
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
    //making the matrix for the representation in other basis
    public static bool[,] MakeBasisMatrix(int[] Degs, BitArray prim)
```

```
    {
        if (Degs.Length != prim.Length - 1) { throw new ArgumentException(); }
        bool[,] res = new bool[Degs.Length,Degs.Length];
        FieldF_2N a = new FieldF_2N(prim, Convert(2));
        for (int i = 0; i < Degs.Length; i++)
        {
            FieldF_2N b = a.Pow(Degs[i]);
            for (int j = Degs.Length - 1; j >= 0 ; j--)
            {
                res[i, j] = b.Val[j];
            }
        }
        return res;
    }
    //checks the transformation matrix for the equality of determinant to zero
    public static bool IsBasis(bool[,] matr)
    {
        bool[,] res = new bool[matr.GetLength(0), matr.GetLength(0)];
        for (int i = 0; i < matr.GetLength(0); i++)
        {
            for (int j = 0; j < matr.GetLength(0); j++)
            {
                res[i, j] = matr[i, j];
            }
        }
        for (int i = 0; i < res.GetLength(0); i++)
        {
            //if diagonal element equals to 1, then we remove the other elements of
the column by adding rows
            if (res[i, i])
            {
                for (int j = i + 1; j < res.GetLength(0); j++)
                {
                    if (res[j, i])
                    {
                        for (int k = 0; k < res.GetLength(0); k++)
                        {
                            res[j, k] = res[j, k] ^ res[i, k];
                        }
                    }
                }
            }
            else
            {
                //else we find such row and swap it with the initial one
                bool found = false; int j = i + 1;
                while ((!found) && (j < res.GetLength(0)))
                {
                    if (res[j, i]) { found = true; } else { j++; }
                }
                if (found)
                {
                    //if the row is found, then we make the first step
                    for (int k = 0; k < res.GetLength(0); k++)
                    {
                        bool tmp = res[i, k];
                        res[i, k] = res[j, k];
                        res[j, k] = tmp;
                    }
                    for (j = i + 1; j < res.GetLength(0); j++)
                    {
                        if (res[j, i])
                        {
                            for (int k = 0; k < res.GetLength(0); k++)
```

```
                        {
                            res[j, k] = res[j, k] ^ res[i, k];
                        }
                    }
                }
                //else the determinant is zero
                else { return false; }
            }
        }
        //if all diagonal elements are equal to 1, then the determinant is equal to
zero
        return true;
    }
    //finds inverse matrix
    public static bool[,] Inv(bool[,] matr)
    {
        //we concate the identity matrix to the initial one from the right side
        bool[,] M = new bool[matr.GetLength(0), 2 * matr.GetLength(0)];
        for (int i = 0; i < matr.GetLength(0); i++)
        {
            for (int j = 0; j < matr.GetLength(0); j++)
            {
                M[i, j] = matr[i, j];
            }
            M[i, i + matr.GetLength(0)] = true;
        }
        //if we transform the left matrix to triangular form, then the right one will
be the inverse matrix
        //we use the Gaussian lgorithm 2 times
        for (int i = 0; i < M.GetLength(0); i++)
        {
            if (M[i, i])
            {
                for (int j = i + 1; j < M.GetLength(0); j++)
                {
                    if (M[j, i])
                    {
                        for (int k = 0; k < M.GetLength(1); k++)
                        {
                            M[j, k] = M[j, k] ^ M[i, k];
                        }
                    }
                }
            }
            else
            {
                bool found = false; int j = i + 1;
                while ((!found) && (j < M.GetLength(0)))
                {
                    if (M[j, i]) { found = true; } else { j++; }
                }
                if (found)
                {
                    for (int k = 0; k < M.GetLength(1); k++)
                    {
                        bool tmp = M[i, k];
                        M[i, k] = M[j, k];
                        M[j, k] = tmp;
                    }
                    for (j = i + 1; j < M.GetLength(0); j++)
                    {
                        if (M[j, i])
                        {
```

```
                        for (int k = 0; k < M.GetLength(1); k++)
                        {
                            M[j, k] = M[j, k] ^ M[i, k];
                        }
                    }
                }
            }
        }
        for (int i = M.GetLength(0) - 1; i >= 0; i--)
        {
            if (M[i, i])
            {
                for (int j = i - 1; j >= 0; j--)
                {
                    if (M[j, i])
                    {
                        for (int k = 0; k < M.GetLength(1); k++)
                        {
                            M[j, k] = M[j, k] ^ M[i, k];
                        }
                    }
                }
            }
            else
            {
                bool found = false; int j = i - 1;
                while ((!found) && (j >= 0))
                {
                    if (M[j, i]) { found = true; } else { j--; }
                }
                if (found)
                {
                    for (int k = 0; k < M.GetLength(1); k++)
                    {
                        bool tmp = M[i, k];
                        M[i, k] = M[j, k];
                        M[j, k] = tmp;
                    }
                    for (j = i - 1; j >= 0; j--)
                    {
                        if (M[j, i])
                        {
                            for (int k = 0; k < M.GetLength(1); k++)
                            {
                                M[j, k] = M[j, k] ^ M[i, k];
                            }
                        }
                    }
                }
            }
        }
        bool[,] res = new bool[matr.GetLength(0), matr.GetLength(0)];
        for (int i = 0; i < matr.GetLength(0); i++)
        {
            for (int j = 0; j < matr.GetLength(0); j++)
            {
                res[i, j] = M[i, j + matr.GetLength(0)];
            }
        }
        return res;
    }
    //the representation of an element of the field in the new basis.
public FieldF_2N InNewBasis(bool[,] BasisMatr)
```

```
    {
        BitArray Newval = new BitArray(Val.Length);
        for (int i = 0; i < Val.Length; i++)
        {
            bool s = false;
            for (int j = 0; j < Val.Length; j++)
            {
                s = s ^ (Val[j] & BasisMatr[j, i]);
            }
            Newval[i] = s;
        }
        return new FieldF_2N(Prim, Newval);
    }
    private static string BitArrayString(BitArray arr)
    {
        int k = 0;
        for (int i = 0; i < arr.Length; i++)
        {
            if (arr[i]) k++;
        }
        if (k == 0) return "";
        string res = "";
        for (int i = arr.Length - 1; i >= 2; i--)
        {
            if (arr[i]) { res = res + "+a^" + i; }
        }
        if (arr.Length >= 2 && arr[1]) { res = res + "+a"; }
        if (arr[0]) { res = res + "+1"; }
        if (k > 1) { res = "(" + res + ")"; }
        return res;
    }
    //output
    public override string ToString()
    {
        string PrimStr = BitArrayString(Prim);
        PrimStr = PrimStr.Remove(0, 1);
        string ValStr = BitArrayString(Val);
        if (ValStr.Length == 0)
        {
            ValStr = "0";
        }
        else
        {
            ValStr = ValStr.Remove(0, 1);
        }
        return ValStr;
    }
}
```

The module of monomials Monom.cs

```
using System;

public class Monom //monomial
{
    public int N;
    public static char[] Chars = new char[] {'x','y','z','t','u','v','w'};
    public int[] Sign; //the signature of monomial (multidegree)
    //constructors
    public Monom(int[] sgn)
    {
        N = sgn.Length;
        Sign = new int[N];
        for (int i = 0; i < N; i++)
```

```
    {
        Sign[i] = sgn[i];
    }
}

public Monom(int N)
{
    this.Sign = new int[N];
    this.N = N;
}

public Monom(Monom mon) : this(mon.Sign) { }
//we use this if we change the order of variables
public Monom NewVarOrder(int[] ords)
{
    if (ords.Length != N) { throw new ArgumentException(); }
    int[] res = new int[N];
    for (int i = 0; i < N; i++) { res[i] = Sign[ords[i] - 1]; }
    return new Monom(res);
}
//full degree
public int FullDeg()
{
    int s = 0;
    for (int i = 0; i < N; i++) { s = s + Sign[i]; }
    return s;
}
//the least common multiple of monomials
public static Monom LCM(Monom a, Monom b)
{
    if (a.N != b.N) throw new ArithmeticException();
    int[] sgnres = new int[a.N];
    for (int i = 0; i < a.N; i++) { sgnres[i] = Math.Max(a.Sign[i], b.Sign[i]); }
    return new Monom(sgnres);
}
//the product of monomials
public static Monom Prod(Monom a, Monom b)
{
    if (a.N != b.N) throw new ArithmeticException();
    int[] sgnres = new int[a.N];
    for (int i = 0; i < a.N; i++) { sgnres[i] = a.Sign[i] + b.Sign[i]; }
    return new Monom(sgnres);
}

public static Monom Div(Monom a, Monom b)
{
    if (a.N != b.N) throw new ArithmeticException();
    int[] sgnres = new int[a.N];
    for (int i = 0; i < a.N; i++) { sgnres[i] = a.Sign[i] - b.Sign[i]; }
    return new Monom(sgnres);
}

public static bool IsDivisible(Monom a, Monom b)
{
    if (a.N != b.N) throw new ArithmeticException();
    int i = 0; bool div = true;
    while ((i < a.N) && div)
    {
        if ((a.Sign[i] - b.Sign[i]) < 0) div = false;
        else i++;
    }
    return div;
}
```

```
    public override bool Equals(object obj)
    {
        Monom Mobj = obj as Monom;
        if (Mobj == null) return false;
        if (N != Mobj.N) return false;
        else
        {
            bool found = false; int i = 0;
            while ((!found) && (i < N))
            {
                if (Sign[i] != Mobj.Sign[i]) found = true;
                else i++;
            }
            return !found;
        }
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
    //output
    public override string ToString()
    {
        string res = "";
        for (int i = 0; i < N; i++)
        {
            if (Sign[i] > 0)
            {
                res = res + Chars[i];
                if (Sign[i] > 1) { res = res + "^" + Sign[i]; }
            }
        }
        return res;
    }
}
```

The module MonomComparers.cs

```
using System.Collections.Generic;
//lexicographical ordering
public class CompareLex : IComparer<Monom>
{
    public int Compare(Monom mon1, Monom mon2)
    {
        int N = mon1.sgn.Length;
        int[] sub = new int[N];
        for (int j = 0; j < N; j++) { sub[j] = mon1.sgn[j] - mon2.sgn[j]; }
        bool found = false; int i = 0;
        while ((!found) && (i < N)) { if (sub[i] != 0) { found = true; } else { i++; }
}
        if (!found) { return 0; }
        if (sub[i] > 0) { return -1; }
        return 1;
    }
}
//graded lexicographical ordering
public class CompareGrLex : IComparer<Monom>
{
    public int Compare(Monom mon1, Monom mon2)
    {
        int s1, s2;
        s1 = mon1.FullDeg();
        s2 = mon2.FullDeg();
        if (s1 > s2) { return -1; }
```

```
        if (s1 < s2) { return 1; }
        CompareLex Comp = new CompareLex();
        return Comp.Compare(mon1, mon2);
    }
}
//graded reverse lexicographical ordering
public class CompareGrevLex : IComparer<Monom>
{
    public int Compare(Monom mon1, Monom mon2)
    {
        int s1, s2;
        s1 = mon1.FullDeg();
        s2 = mon2.FullDeg();
        if (s1 > s2) { return -1; }
        if (s1 < s2) { return 1; }
        int N = mon1.sgn.Length;
        int[] sub = new int[N];
        for (int j = 0; j < N; j++) { sub[j] = mon1.sgn[j] - mon2.sgn[j]; }
        bool found = false; int i = N - 1;
        while ((!found) && (i >= 0)) { if (sub[i] != 0) { found = true; } else { i--;
} }
        if (!found) { return 0; }
        if (sub[i] < 0) { return -1; }
        return 1;
    }
}
//inverse lexicographical ordering
public class CompareInvLex : IComparer<Monom>
{
    public int Compare(Monom mon1, Monom mon2)
    {
        int N = mon1.sgn.Length;
        int[] sub = new int[N];
        for (int j = 0; j < N; j++) { sub[j] = mon1.sgn[j] - mon2.sgn[j]; }
        bool found = false; int i = N - 1;
        while ((!found) && (i >= 0)) { if (sub[i] != 0) { found = true; } else { i--;
} }
        if (!found) { return 0; }
        if (sub[i] > 0) { return -1; }
        return 1;
    }
}
//reverse Invlex-ordering
public class CompareRinvLex : IComparer<Monom>
{
    public int Compare(Monom mon1, Monom mon2)
    {
        CompareInvLex Comp = new CompareInvLex();
        return Comp.Compare(mon2, mon1);
    }
}
```

The module of polynomials Polynom

```
using System;
using System.Collections;
using System.Collections.Generic;

public class PolynomC //the polynomial with complex coefficients
{
    public int VarCnt = -1;
    public SortedList<Monom, FieldC> list;
    public int sortWay;
```

```
public Monom GetMonom(int index)
{
    return list.Keys[index];
}
public FieldC GetCoefficient(int index)
{
    return list.Values[index];
}
public void Add(Monom mon, FieldC coef)
{
    if (VarCnt == -1) VarCnt = mon.N;
    if (VarCnt != mon.N) throw new ArgumentException();
    if (list.ContainsKey(mon))
    {
        FieldC sum = list[mon] + coef;
        if (sum != 0) { list[mon] = sum; } else { list.Remove(mon); }
    }
    else { list.Add(mon, coef); }
}
public PolynomC(int sortWay)
{
    switch (sortWay)
    {
        case 1:
            list = new SortedList<Monom, FieldC>(new CompareLex()); break;
        case 2:
            list = new SortedList<Monom, FieldC>(new CompareGrLex()); break;
        case 3:
            list = new SortedList<Monom, FieldC>(new CompareGrevLex()); break;
        case 4:
            list = new SortedList<Monom, FieldC>(new CompareInvLex()); break;
        case 5:
            list = new SortedList<Monom, FieldC>(new CompareRinvLex()); break;
    }
    this.sortWay = sortWay;
}
public PolynomC(Monom[] monoms, FieldC[] coefs, int sortWay) : this(sortWay)
{
    if (monoms.Length != coefs.Length) throw new ArgumentException();
    for (int i = 0; i < monoms.Length; i++)
    {
        Add(monoms[i], coefs[i]);
    }
}
public int[] Multideg() { return GetMonom(0).Sign; }
public FieldC LC() { return GetCoefficient(0); }
public Monom LM() { return GetMonom(0); }
public PolynomC LT()
{
    Monom[] monoms = new Monom[1];
    monoms[0] = LM();
    FieldC[] coefs = new FieldC[1];
    coefs[0] = LC();
    return new PolynomC(monoms, coefs, sortWay);
}
public PolynomC NewVarOrder(int[] ords)
{
    Monom[] monoms = new Monom[list.Count];
    FieldC[] coefs = new FieldC[list.Count];
    for (int i = 0; i < list.Count; i++)
    {
        monoms[i] = new Monom(GetMonom(i).NewVarOrder(ords));
        coefs[i] = new FieldC(GetCoefficient(i));
    }
```

```
        return new PolynomC(monoms, coefs, sortWay);
    }
    public static PolynomC operator+ (PolynomC a, PolynomC b)
    {
        PolynomC res = new PolynomC(a.sortWay);
        for (int i = 0; i < a.list.Count; i++)
        {
            res.Add(new Monom(a.GetMonom(i)), new FieldC(a.GetCoefficient(i)));
        }
        for (int i = 0; i < b.list.Count; i++)
        {
            res.Add(new Monom(b.GetMonom(i)), new FieldC(b.GetCoefficient(i)));
        }
        return res;
    }
    public static PolynomC operator- (PolynomC a)
    {
        PolynomC res = new PolynomC(a.sortWay);
        for (int i = 0; i < a.list.Count; i++)
        {
            res.Add(new Monom(a.GetMonom(i)), new FieldC(-a.GetCoefficient(i)));
        }
        return res;
    }
    public static PolynomC operator* (PolynomC a, PolynomC b)
    {
        PolynomC res = new PolynomC(a.sortWay);
        Monom ma, mb; int[] Signres;
        for (int i = 0; i < a.list.Count; i++)
        {
            ma = new Monom(a.GetMonom(i));
            for (int j = 0; j < b.list.Count; j++)
            {
                mb = new Monom(b.GetMonom(j));
                if (ma.N != mb.N) throw new ArgumentException();
                Signres = new int[ma.N];
                for (int k = 0; k < ma.N; k++)
                {
                    Signres[k] = ma.Sign[k] + mb.Sign[k];
                }
                res.Add(new Monom(Signres), a.GetCoefficient(i) *
b.GetCoefficient(j));
            }
        }
        return res;
    }
    public PolynomC Simplify()
    {
        FieldC coefinv = new FieldC(GetCoefficient(0)).Inv();
        PolynomC res = new PolynomC(sortWay);
        for (int i = 0; i < list.Count; i++)
        {
            FieldC coef = new FieldC(GetCoefficient(i));
            Monom mon = new Monom(GetMonom(i));
            res.Add(mon, coef * coefinv);
        }
        return res;
    }
    public override bool Equals(object obj)
    {
        PolynomC Pobj = obj as PolynomC;
        if (Pobj == null) return false;
        bool Eq = (VarCnt == Pobj.VarCnt) && (sortWay == Pobj.sortWay) && (list.Count
== Pobj.list.Count);
```

```
        if (Eq)
        {
            bool found = false; int i = 0;
            while ((Eq) && (i < list.Count))
            {
                if ((!GetMonom(i).Equals(Pobj.GetMonom(i))) ||
(!GetCoefficient(i).Equals(Pobj.GetCoefficient(i)))) Eq = false;
                i++;
            }
            return !found;
        }
        else return false;
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}

public class PolynomQ //polynomial with rational coefficients
{
    public int VarCnt = -1;
    public SortedList<Monom, FieldQ> list;
    public int sortWay;
    public Monom GetMonom(int index)
    {
        return list.Keys[index];
    }
    public FieldQ GetCoefficient(int index)
    {
        return list.Values[index];
    }
    public void Add(Monom mon, FieldQ coef)
    {
        if (VarCnt == -1) VarCnt = mon.N;
        if (VarCnt != mon.N) throw new ArgumentException();
        if (list.ContainsKey(mon))
        {
            FieldQ sum = list[mon] + coef;
            if (sum != 0) { list[mon] = sum; } else { list.Remove(mon); }
        }
        else { list.Add(mon, coef); }
    }
    public PolynomQ(int sortWay)
    {
        switch (sortWay)
        {
            case 1:
                list = new SortedList<Monom, FieldQ>(new CompareLex()); break;
            case 2:
                list = new SortedList<Monom, FieldQ>(new CompareGrLex()); break;
            case 3:
                list = new SortedList<Monom, FieldQ>(new CompareGrevLex()); break;
            case 4:
                list = new SortedList<Monom, FieldQ>(new CompareInvLex()); break;
            case 5:
                list = new SortedList<Monom, FieldQ>(new CompareRinvLex()); break;
        }
        this.sortWay = sortWay;
    }
    public PolynomQ(Monom[] monoms, FieldQ[] coefs, int sortWay) : this(sortWay)
    {
        if (monoms.Length != coefs.Length) throw new ArgumentException();
        for (int i = 0; i < monoms.Length; i++)
```

```
    {
        Add(monoms[i], coefs[i]);
    }
}
public PolynomQ(PolynomQ pol) : this(pol.sortWay)
{
    for (int i = 0; i < pol.list.Count; i++)
    {
        Add(new Monom(pol.GetMonom(i)), new FieldQ(pol.GetCoefficient(i)));
    }
}
public int[] Multideg() { return GetMonom(0).Sign; }
public FieldQ LC() { return GetCoefficient(0); }
public Monom LM() { return GetMonom(0); }
public PolynomQ LT()
{
    Monom[] monoms = new Monom[1];
    monoms[0] = LM();
    FieldQ[] coefs = new FieldQ[1];
    coefs[0] = LC();
    return new PolynomQ(monoms, coefs, sortWay);
}
public PolynomQ NewVarOrder(int[] ords)
{
    Monom[] monoms = new Monom[list.Count];
    FieldQ[] coefs = new FieldQ[list.Count];
    for (int i = 0; i < list.Count; i++)
    {
        monoms[i] = new Monom(GetMonom(i).NewVarOrder(ords));
        coefs[i] = new FieldQ(GetCoefficient(i));
    }
    return new PolynomQ(monoms, coefs, sortWay);
}
public static PolynomQ operator +(PolynomQ a, PolynomQ b)
{
    PolynomQ res = new PolynomQ(a.sortWay);
    for (int i = 0; i < a.list.Count; i++)
    {
        res.Add(new Monom(a.GetMonom(i)), new FieldQ(a.GetCoefficient(i)));
    }
    for (int i = 0; i < b.list.Count; i++)
    {
        res.Add(new Monom(b.GetMonom(i)), new FieldQ(b.GetCoefficient(i)));
    }
    return res;
}
public static PolynomQ operator -(PolynomQ a)
{
    PolynomQ res = new PolynomQ(a.sortWay);
    for (int i = 0; i < a.list.Count; i++)
    {
        res.Add(new Monom(a.GetMonom(i)), new FieldQ(-a.GetCoefficient(i)));
    }
    return res;
}
public static PolynomQ operator *(PolynomQ a, PolynomQ b)
{
    PolynomQ res = new PolynomQ(a.sortWay);
    Monom ma, mb; int[] Signres;
    for (int i = 0; i < a.list.Count; i++)
    {
        ma = new Monom(a.GetMonom(i));
        for (int j = 0; j < b.list.Count; j++)
        {
```

```
                mb = new Monom(b.GetMonom(j));
                if (ma.N != mb.N) throw new ArgumentException();
                Signres = new int[ma.N];
                for (int k = 0; k < ma.N; k++)
                {
                    Signres[k] = ma.Sign[k] + mb.Sign[k];
                }
                res.Add(new Monom(Signres), a.GetCoefficient(i) *
b.GetCoefficient(j));
            }
        }
        return res;
    }
    public PolynomQ Simplify()
    {
        FieldQ coefinv = new FieldQ(GetCoefficient(0)).Inv();
        PolynomQ res = new PolynomQ(sortWay);
        for (int i = 0; i < list.Count; i++)
        {
            FieldQ coef = new FieldQ(GetCoefficient(i));
            Monom mon = new Monom(GetMonom(i));
            res.Add(mon, coef * coefinv);
        }
        return res;
    }
    public override bool Equals(object obj)
    {
        PolynomQ Pobj = obj as PolynomQ;
        if (Pobj == null) return false;
        bool Eq = (VarCnt == Pobj.VarCnt) && (sortWay == Pobj.sortWay) && (list.Count
== Pobj.list.Count);
        if (Eq)
        {
            int i = 0;
            while ((Eq) && (i < list.Count))
            {
                if ((!GetMonom(i).Equals(Pobj.GetMonom(i))) ||
(!GetCoefficient(i).Equals(Pobj.GetCoefficient(i)))) Eq = false;
                i++;
            }
            return Eq;
        }
        else return false;
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}

public class PolynomF_P //polynomial with coefficients from the simple Galois field
{
    public int VarCnt = -1;
    public SortedList<Monom, FieldF_P> list;
    public int sortWay;
    public Monom GetMonom(int index)
    {
        return list.Keys[index];
    }
    public FieldF_P GetCoefficient(int index)
    {
        return list.Values[index];
    }
    public void Add(Monom mon, FieldF_P coef)
```

```
{
    if (VarCnt == -1) VarCnt = mon.N;
    if (VarCnt != mon.N) throw new ArgumentException();
    if (list.ContainsKey(mon))
    {
        FieldF_P sum = list[mon] + coef;
        if (!sum.IsNull()) { list[mon] = sum; } else { list.Remove(mon); }
    }
    else { list.Add(mon, coef); }
}
public PolynomF_P(int sortWay)
{
    switch (sortWay)
    {
        case 1:
            list = new SortedList<Monom, FieldF_P>(new CompareLex()); break;
        case 2:
            list = new SortedList<Monom, FieldF_P>(new CompareGrLex()); break;
        case 3:
            list = new SortedList<Monom, FieldF_P>(new CompareGrevLex()); break;
        case 4:
            list = new SortedList<Monom, FieldF_P>(new CompareInvLex()); break;
        case 5:
            list = new SortedList<Monom, FieldF_P>(new CompareRinvLex()); break;
    }
    this.sortWay = sortWay;
}
public PolynomF_P(Monom[] monoms, FieldF_P[] coefs, int sortWay) : this(sortWay)
{
    if (monoms.Length != coefs.Length) throw new ArgumentException();
    for (int i = 0; i < monoms.Length; i++)
    {
        Add(monoms[i], coefs[i]);
    }
}
public int[] Multideg() { return GetMonom(0).Sign; }
public FieldF_P LC() { return GetCoefficient(0); }
public Monom LM() { return GetMonom(0); }
public PolynomF_P LT()
{
    Monom[] monoms = new Monom[1];
    monoms[0] = LM();
    FieldF_P[] coefs = new FieldF_P[1];
    coefs[0] = LC();
    return new PolynomF_P(monoms, coefs, sortWay);
}
public PolynomF_P NewVarOrder(int[] ords)
{
    Monom[] monoms = new Monom[list.Count];
    FieldF_P[] coefs = new FieldF_P[list.Count];
    for (int i = 0; i < list.Count; i++)
    {
        monoms[i] = new Monom(GetMonom(i).NewVarOrder(ords));
        coefs[i] = new FieldF_P(GetCoefficient(i));
    }
    return new PolynomF_P(monoms, coefs, sortWay);
}
public static PolynomF_P operator +(PolynomF_P a, PolynomF_P b)
{
    PolynomF_P res = new PolynomF_P(a.sortWay);
    for (int i = 0; i < a.list.Count; i++)
    {
        res.Add(new Monom(a.GetMonom(i)), new FieldF_P(a.GetCoefficient(i)));
    }
```

```
        for (int i = 0; i < b.list.Count; i++)
        {
            res.Add(new Monom(b.GetMonom(i)), new FieldF_P(b.GetCoefficient(i)));
        }
        return res;
    }
    public static PolynomF_P operator -(PolynomF_P a)
    {
        PolynomF_P res = new PolynomF_P(a.sortWay);
        for (int i = 0; i < a.list.Count; i++)
        {
            res.Add(new Monom(a.GetMonom(i)), new FieldF_P(-a.GetCoefficient(i)));
        }
        return res;
    }
    public static PolynomF_P operator *(PolynomF_P a, PolynomF_P b)
    {
        PolynomF_P res = new PolynomF_P(a.sortWay);
        Monom ma, mb; int[] Signres;
        for (int i = 0; i < a.list.Count; i++)
        {
            ma = new Monom(a.GetMonom(i));
            for (int j = 0; j < b.list.Count; j++)
            {
                mb = new Monom(b.GetMonom(j));
                if (ma.N != mb.N) throw new ArgumentException();
                Signres = new int[ma.N];
                for (int k = 0; k < ma.N; k++)
                {
                    Signres[k] = ma.Sign[k] + mb.Sign[k];
                }
                res.Add(new Monom(Signres), a.GetCoefficient(i) *
b.GetCoefficient(j));
            }
        }
        return res;
    }
    public PolynomF_P Simplify()
    {
        FieldF_P coefinv = new FieldF_P(GetCoefficient(0)).Inv();
        PolynomF_P res = new PolynomF_P(sortWay);
        for (int i = 0; i < list.Count; i++)
        {
            FieldF_P coef = new FieldF_P(GetCoefficient(i));
            Monom mon = new Monom(GetMonom(i));
            res.Add(mon, coef * coefinv);
        }
        return res;
    }
    public override bool Equals(object obj)
    {
        PolynomF_P Pobj = obj as PolynomF_P;
        if (Pobj == null) return false;
        bool Eq = (VarCnt == Pobj.VarCnt) && (sortWay == Pobj.sortWay) && (list.Count
== Pobj.list.Count);
        if (Eq)
        {
            bool found = false; int i = 0;
            while ((Eq) && (i < list.Count))
            {
                if ((!GetMonom(i).Equals(Pobj.GetMonom(i))) ||
(!GetCoefficient(i).Equals(Pobj.GetCoefficient(i)))) Eq = false;
                i++;
            }
```

```
            return !found;
        }
        else return false;
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}

public class PolynomF_2N //polynomial with coefficients from the finite field with
characteristic 2
{
    public int VarCnt = -1;
    public SortedList<Monom, FieldF_2N> list;
    public int sortWay;
    public Monom GetMonom(int index)
    {
        return list.Keys[index];
    }
    public FieldF_2N GetCoefficient(int index)
    {
        return list.Values[index];
    }
    public void Add(Monom mon, FieldF_2N coef)
    {
        if (VarCnt == -1) VarCnt = mon.N;
        if (VarCnt != mon.N) throw new ArgumentException();
        if (list.ContainsKey(mon))
        {
            FieldF_2N sum = list[mon].Add(coef);
            if (!sum.IsNull()) { list[mon] = sum; } else { list.Remove(mon); }
        }
        else { list.Add(mon, coef); }
    }
    public PolynomF_2N(int sortWay)
    {
        switch (sortWay)
        {
            case 1:
                list = new SortedList<Monom, FieldF_2N>(new CompareLex()); break;
            case 2:
                list = new SortedList<Monom, FieldF_2N>(new CompareGrLex()); break;
            case 3:
                list = new SortedList<Monom, FieldF_2N>(new CompareGrevLex()); break;
            case 4:
                list = new SortedList<Monom, FieldF_2N>(new CompareInvLex()); break;
            case 5:
                list = new SortedList<Monom, FieldF_2N>(new CompareRinvLex()); break;
        }
        this.sortWay = sortWay;
    }
    public PolynomF_2N(Monom[] monoms, FieldF_2N[] coefs, int sortWay) : this(sortWay)
    {
        if (monoms.Length != coefs.Length) throw new ArgumentException();
        for (int i = 0; i < monoms.Length; i++)
        {
            Add(monoms[i], coefs[i]);
        }
    }
    public PolynomF_2N(PolynomF_2N pol) : this(pol.sortWay)
    {
        for (int i = 0; i < pol.list.Count; i++)
        {
```

```
            Add(new Monom(pol.GetMonom(i)), new FieldF_2N(pol.GetCoefficient(i)));
        }
    }
    public int[] Multideg() { return GetMonom(0).Sign; }
    public FieldF_2N LC() { return GetCoefficient(0); }
    public Monom LM() { return GetMonom(0); }
    public PolynomF_2N LT()
    {
        Monom[] monoms = new Monom[1];
        monoms[0] = LM();
        FieldF_2N[] coefs = new FieldF_2N[1];
        coefs[0] = LC();
        return new PolynomF_2N(monoms, coefs, sortWay);
    }
    public PolynomF_2N NewVarOrder(int[] ords)
    {
        Monom[] monoms = new Monom[list.Count];
        FieldF_2N[] coefs = new FieldF_2N[list.Count];
        for (int i = 0; i < list.Count; i++)
        {
            monoms[i] = new Monom(GetMonom(i).NewVarOrder(ords));
            coefs[i] = new FieldF_2N(GetCoefficient(i));
        }
        return new PolynomF_2N(monoms, coefs, sortWay);
    }
    public static PolynomF_2N operator +(PolynomF_2N a, PolynomF_2N b)
    {
        PolynomF_2N res = new PolynomF_2N(a.sortWay);
        for (int i = 0; i < a.list.Count; i++)
        {
            res.Add(new Monom(a.GetMonom(i)), new FieldF_2N(a.GetCoefficient(i)));
        }
        for (int i = 0; i < b.list.Count; i++)
        {
            res.Add(new Monom(b.GetMonom(i)), new FieldF_2N(b.GetCoefficient(i)));
        }
        return res;
    }
    public static PolynomF_2N operator -(PolynomF_2N a)
    {
        PolynomF_2N res = new PolynomF_2N(a.sortWay);
        for (int i = 0; i < a.list.Count; i++)
        {
            res.Add(new Monom(a.GetMonom(i)), new FieldF_2N(a.GetCoefficient(i)));
        }
        return res;
    }
    public static PolynomF_2N operator *(PolynomF_2N a, PolynomF_2N b)
    {
        PolynomF_2N res = new PolynomF_2N(a.sortWay);
        Monom ma, mb; int[] Signres;
        for (int i = 0; i < a.list.Count; i++)
        {
            ma = new Monom(a.GetMonom(i));
            for (int j = 0; j < b.list.Count; j++)
            {
                mb = new Monom(b.GetMonom(j));
                if (ma.N != mb.N) throw new ArgumentException();
                Signres = new int[ma.N];
                for (int k = 0; k < ma.N; k++)
                {
                    Signres[k] = ma.Sign[k] + mb.Sign[k];
                }
```

```
                res.Add(new Monom(Signres), a.GetCoefficient(i) *
b.GetCoefficient(j));
            }
        }
        return res;
    }
    public PolynomF_2N Simplify()
    {
        FieldF_2N coefinv = new FieldF_2N(GetCoefficient(0)).Inv();
        PolynomF_2N res = new PolynomF_2N(sortWay);
        for (int i = 0; i < list.Count; i++)
        {
            FieldF_2N coef = new FieldF_2N(GetCoefficient(i));
            Monom mon = new Monom(GetMonom(i));
            res.Add(mon, coef * coefinv);
        }
        return res;
    }
    public PolynomF_2N Pow(int n)
    {
        PolynomF_2N res = new PolynomF_2N(sortWay);
        BitArray prim = GetCoefficient(0).Prim;
        FieldF_2N one = new FieldF_2N(prim, new BitArray(prim.Length - 1));
        one.SetVal(0, true);
        res.Add(new Monom(new int[prim.Length - 1]), one);
        for (int i = 0; i < n; i++)
        {
            res = res * this;
        }
        return res;
    }
    public PolynomF_2N Mult(FieldF_2N coef)
    {
        PolynomF_2N res = new PolynomF_2N(sortWay);
        for (int i = 0; i < list.Count; i++)
        {
            res.Add(GetMonom(i), GetCoefficient(i) * coef);
        }
        return res;
    }
    public PolynomF_2N UnifyDegrees() //if the variable degree is greater than 1, it
sets to 1.
    {
        PolynomF_2N res = new PolynomF_2N(sortWay);
        for (int i = 0; i < list.Count; i++)
        {
            int[] sgn = GetMonom(i).Sign;
            for (int j = 0; j < sgn.Length; j++)
            {
                if (sgn[j] > 1) { sgn[j] = 1; }
            }
            res.Add(new Monom(sgn), GetCoefficient(i));
        }
        return res;
    }
    public override bool Equals(object obj)
    {
        PolynomF_2N Pobj = obj as PolynomF_2N;
        if (Pobj == null) return false;
        bool Eq = (VarCnt == Pobj.VarCnt) && (sortWay == Pobj.sortWay) && (list.Count
== Pobj.list.Count);
        if (Eq)
        {
            bool found = false; int i = 0;
```

```
            while ((Eq) && (i < list.Count))
            {
                if ((!GetMonom(i).Equals(Pobj.GetMonom(i))) ||
(!GetCoefficient(i).Equals(Pobj.GetCoefficient(i)))) Eq = false;
                i++;
            }
            return Eq;
        }
        else return false;
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
    public override string ToString()
    {
        string res = "";
        for (int i = 0; i < list.Count; i++)
        {
            res = res + "+" + GetCoefficient(i).ToString() + GetMonom(i).ToString();
        }
        if (res == "") res = "0";
        else res = res.Remove(0, 1);
        if (res == "1") return "";
        return res;
    }
}
```

The module Grebner.cs

```
using System;
using System.Collections;
using System.Collections.Generic;

public class Grebner
{
    //obtaing the remainder, using the division algorithm in the polynomial field
    public static PolynomQ Mod(PolynomQ f, PolynomQ[] fs)
    {
        PolynomQ r = new PolynomQ(f.sortWay);
        PolynomQ p = new PolynomQ(f.sortWay);
        for (int i = 0; i < f.list.Count; i++){ p.Add(f.GetMonom(i),
f.GetCoefficient(i)); }
        while (p.list.Count != 0)
        {
            int i = 0; bool havediv = false;
            PolynomQ ltp, ltfi, ltdiv;
            ltp = p.LT();
            while ((i < fs.Length) &&(!havediv))
            {
                ltfi = fs[i].LT(); bool div = true;
                int[] sgndiv = new int[ltp.GetMonom(0).N];
                for (int j = 0; j < sgndiv.Length; j++)
                {
                    sgndiv[j] = ltp.GetMonom(0).Sign[j] - ltfi.GetMonom(0).Sign[j];
                    if (sgndiv[j] < 0) { div = false; }
                }
                if (div)
                {
                    ltdiv = new PolynomQ(ltp.sortWay);
                    ltdiv.Add(new Monom(sgndiv), ltp.LC() / ltfi.LC());
                    ltdiv = ltdiv * fs[i]; p = p + (-ltdiv); havediv = true;
                }
                else { i++; }
            }
```

```
            if (!havediv) { ltp = p.LT(); r = ltp + r; p = p + (-ltp); }
        }
        return r;
    }
    public static PolynomC Mod(PolynomC f, PolynomC[] fs)
    {
        PolynomC r = new PolynomC(f.sortWay);
        PolynomC p = new PolynomC(f.sortWay);
        for (int i = 0; i < f.list.Count; i++) { p.Add(f.GetMonom(i),
f.GetCoefficient(i)); }
        while (p.list.Count != 0)
        {
            int i = 0; bool havediv = false;
            PolynomC ltp, ltfi, ltdiv;
            ltp = p.LT();
            while ((i < fs.Length) && (!havediv))
            {
                ltfi = fs[i].LT(); bool div = true;
                int[] sgndiv = new int[ltp.LM().N];
                for (int j = 0; j < sgndiv.Length; j++)
                {
                    sgndiv[j] = ltp.LM().Sign[j] - ltfi.LM().Sign[j];
                    if (sgndiv[j] < 0) { div = false; }
                }
                if (div)
                {
                    ltdiv = new PolynomC(ltp.sortWay);
                    ltdiv.Add(new Monom(sgndiv), ltp.LC() / ltfi.LC());
                    ltdiv = ltdiv * fs[i]; p = p + (-ltdiv); havediv = true;
                }
                else { i++; }
            }
            if (!havediv) { ltp = p.LT(); r = r + ltp; p = p + (-ltp); }
        }
        return r;
    }
    public static PolynomF_P Mod(PolynomF_P f, PolynomF_P[] fs)
    {
        PolynomF_P r = new PolynomF_P(f.sortWay);
        PolynomF_P p = new PolynomF_P(f.sortWay);
        for (int i = 0; i < f.list.Count; i++) { p.Add(f.GetMonom(i),
f.GetCoefficient(i)); }
        while (p.list.Count != 0)
        {
            int i = 0; bool havediv = false;
            PolynomF_P ltp, ltfi, ltdiv;
            ltp = p.LT();
            while ((i < fs.Length) && (!havediv))
            {
                ltfi = fs[i].LT(); bool div = true;
                int[] sgndiv = new int[ltp.LM().N];
                for (int j = 0; j < sgndiv.Length; j++)
                {
                    sgndiv[j] = ltp.LM().Sign[j] - ltfi.LM().Sign[j];
                    if (sgndiv[j] < 0) { div = false; }
                }
                if (div)
                {
                    ltdiv = new PolynomF_P(ltp.sortWay);
                    ltdiv.Add(new Monom(sgndiv), ltp.LC() / ltfi.LC());
                    ltdiv = ltdiv * fs[i]; p = p + (-ltdiv); havediv = true;
                }
                else { i++; }
            }
```

```
            if (!havediv) { ltp = p.LT(); r = r + ltp; p = p + (-ltp); }
        }
        return r;
    }
    public static PolynomF_2N Mod(PolynomF_2N f, PolynomF_2N[] fs)
    {
        PolynomF_2N r = new PolynomF_2N(f.sortWay);
        PolynomF_2N p = new PolynomF_2N(f.sortWay);
        for (int i = 0; i < f.list.Count; i++) { p.Add(new Monom(f.GetMonom(i)), new
FieldF_2N(f.GetCoefficient(i))); }
        while (p.list.Count != 0)
        {
            int i = 0; bool havediv = false;
            PolynomF_2N ltp, ltfi, ltdiv;
            ltp = p.LT();
            while ((i < fs.Length) && (!havediv))
            {
                ltfi = fs[i].LT(); bool div = true;
                int[] sgndiv = new int[ltp.LM().N];
                for (int j = 0; j < sgndiv.Length; j++)
                {
                    sgndiv[j] = ltp.LM().Sign[j] - ltfi.LM().Sign[j];
                    if (sgndiv[j] < 0) { div = false; }
                }
                if (div)
                {
                    ltdiv = new PolynomF_2N(ltp.sortWay);
                    ltdiv.Add(new Monom(sgndiv), ltp.LC() / ltfi.LC());
                    ltdiv = ltdiv * fs[i]; p = p + ltdiv; havediv = true;
                }
                else { i++; }
            }
            if (!havediv)
            {
                FieldF_2N coef = new FieldF_2N(p.LC()); Monom mon = new Monom(p.LM());
                p.Add(mon, coef);
                r.Add(mon, coef);
            }
        }
        return r;
    }
//S-polynomial construction
    public static PolynomQ S_Pol(PolynomQ f, PolynomQ g)
    {
        Monom lcm = Monom.LCM(f.LM(), g.LM());
        PolynomQ ltf = f.LT(), ltg = g.LT();
        PolynomQ res, resf, resg;
        int[] sgndivf = new int[lcm.N];
        int[] sgndivg = new int[lcm.N];
        for (int i = 0; i < lcm.N; i++)
        {
            sgndivf[i] = lcm.Sign[i] - ltf.LM().Sign[i];
            sgndivg[i] = lcm.Sign[i] - ltg.LM().Sign[i];
        }
        resf = new PolynomQ(f.sortWay); resg = new PolynomQ(f.sortWay);
        resf.Add(new Monom(sgndivf), ltf.LC().Inv()); resg.Add(new Monom(sgndivg),
ltg.LC().Inv());
        resf = resf * f; resg = resg * g;
        res = resf + (-resg);
        return res;
    }

    public static PolynomC S_Pol(PolynomC f, PolynomC g)
    {
```

```
        Monom lcm = Monom.LCM(f.LM(), g.LM());
        PolynomC ltf = f.LT(), ltg = g.LT();
        PolynomC res, resf, resg;
        int[] sgndivf = new int[lcm.N];
        int[] sgndivg = new int[lcm.N];
        for (int i = 0; i < lcm.N; i++)
        {
            sgndivf[i] = lcm.Sign[i] - ltf.LM().Sign[i];
            sgndivg[i] = lcm.Sign[i] - ltg.LM().Sign[i];
        }
        resf = new PolynomC(f.sortWay); resg = new PolynomC(f.sortWay);
        resf.Add(new Monom(sgndivf), ltf.LC().Inv()); resg.Add(new Monom(sgndivg),
ltg.LC().Inv());
        resf = resf * f; resg = resg * g;
        res = resf + (-resg);
        return res;
    }

    public static PolynomF_P S_Pol(PolynomF_P f, PolynomF_P g)
    {
        Monom lcm = Monom.LCM(f.LM(), g.LM());
        PolynomF_P ltf = f.LT(), ltg = g.LT();
        PolynomF_P res, resf, resg;
        int[] sgndivf = new int[lcm.N];
        int[] sgndivg = new int[lcm.N];
        for (int i = 0; i < lcm.N; i++)
        {
            sgndivf[i] = lcm.Sign[i] - ltf.LM().Sign[i];
            sgndivg[i] = lcm.Sign[i] - ltg.LM().Sign[i];
        }
        resf = new PolynomF_P(f.sortWay); resg = new PolynomF_P(f.sortWay);
        resf.Add(new Monom(sgndivf), ltf.LC().Inv()); resg.Add(new Monom(sgndivg),
ltg.LC().Inv());
        resf = resf * f; resg = resg * g;
        res = resf + (-resg);
        return res;
    }

    public static PolynomF_2N S_Pol(PolynomF_2N f, PolynomF_2N g)
    {
        Monom lcm = Monom.LCM(f.LM(), g.LM());
        PolynomF_2N ltf = f.LT(), ltg = g.LT();
        PolynomF_2N res, resf, resg;
        int[] sgndivf = new int[lcm.N];
        int[] sgndivg = new int[lcm.N];
        for (int i = 0; i < lcm.N; i++)
        {
            sgndivf[i] = lcm.Sign[i] - ltf.LM().Sign[i];
            sgndivg[i] = lcm.Sign[i] - ltg.LM().Sign[i];
        }
        resf = new PolynomF_2N(f.sortWay); resg = new PolynomF_2N(f.sortWay);
        resf.Add(new Monom(sgndivf), ltf.LC().Inv()); resg.Add(new Monom(sgndivg),
ltg.LC().Inv());
        resf = resf * f; resg = resg * g;
        res = resf + resg;
        return res;
    }
//the original Buchberger's algorithm for the Grobner basis calculation
    public static HashSet<PolynomF_P> MakeGrebnerBasis(PolynomF_P[] F)
    {
        HashSet<PolynomF_P> res = new HashSet<PolynomF_P>(); PolynomF_P[] G;
        for (int i = 0; i < F.Length; i++) { res.Add(F[i]); }
        do
        {
```

```
            PolynomF_P S;
            G = new PolynomF_P[res.Count];
            int k = 0;
            foreach (PolynomF_P pol in res) { G[k] = pol; k++; }
            for (int i = 0; i < G.Length; i++)
            {
                for (int j = i + 1; j < G.Length; j++)
                {
                    S = Mod(S_Pol(G[i], G[j]), G);
                    if ((S.list.Count != 0) && (!res.Contains(S))) { res.Add(S); }
                }
            }
        }
        while (G.Length != res.Count);
        return res;
    }
    public static bool Criteria(int fi, int fj, List<int[]> B, List<PolynomQ> res,
Monom lcm)
    {
        bool found = false; int l = 0;
        while (!found && (l < res.Count))
        {
            if ((l != fi) && (l != fj))
            {
                bool a, b;
                if (l > fi) { a = B.Exists(x => (x[0] == fi) && (x[1] == l)); }
                else { a = B.Exists(x => (x[0] == l) && (x[1] == fi)); }
                if (l > fj) { b = B.Exists(x => (x[0] == fj) && (x[1] == l)); }
                else { b = B.Exists(x => (x[0] == l) && (x[1] == fj)); }
                if (!a && !b)
                {
                    Monom fl = res[l].LM();
                    bool fact = true; int i = 0;
                    while (fact && (i < fl.N))
                    {
                        if (fl.Sign[i] > lcm.Sign[i]) { fact = false; } else { i++; }
                    }
                    if (fact) { found = true; }
                }
            }
            if (!found) { l++; }
        }
        return found;
    }
    //the second criterion in the improved Buchberger's algorithm
    public static bool Criteria(int fi, int fj, List<int[]> B, List<PolynomF_2N> res,
Monom lcm)
    {
        bool found = false; int l = 0;
        while (!found && (l < res.Count))
        {
            if ((l != fi) && (l != fj))
            {
                bool a, b;
                if (l > fi) { a = B.Exists(x => (x[0] == fi) && (x[1] == l)); }
                else { a = B.Exists(x => (x[0] == l) && (x[1] == fi)); }
                if (l > fj) { b = B.Exists(x => (x[0] == fj) && (x[1] == l)); }
                else { b = B.Exists(x => (x[0] == l) && (x[1] == fj)); }
                if (!a && !b)
                {
                    Monom fl = res[l].LM();
                    bool fact = true; int i = 0;
                    while (fact && (i < fl.N))
                    {
```

```
                        if (fl.Sign[i] > lcm.Sign[i]) { fact = false; } else { i++; }
                    }
                    if (fact) { found = true; }
                }
            }
            if (!found) { l++; }
        }
        return found;
    }
//the calculation of reduced Grobner basis using the improved Buchberger's algorithm
    public static List<PolynomF_2N> MakeGrebnerBasis(PolynomF_2N[] F)
    {
        List<int[]> B = new List<int[]>();
        for (int i = 0; i < F.Length; i++)
        {
            for (int j = i + 1; j < F.Length; j++)
            {
                B.Add(new int[] { i, j });
            }
        }
        List<PolynomF_2N> res = new List<PolynomF_2N>(); PolynomF_2N[] G;
        for (int i = 0; i < F.Length; i++) { res.Add(F[i]); }
        int t = F.Length;
        while (B.Count != 0)
        {
            int[] var = B[0];
            Monom lcm = Monom.LCM(res[var[0]].LM(), res[var[1]].LM());
            Monom prod = Monom.Prod(res[var[0]].LM(), res[var[1]].LM());
            bool eq = true; bool crit = false;
            int i = 0;
            while (eq && (i < lcm.N))
            {
                if (lcm.Sign[i] != prod.Sign[i]) { eq = false; } else { i++; }
            }
            crit = Criteria(var[0], var[1], B, res, lcm);
            if (!eq && !crit)
            {
                PolynomF_2N S;
                G = new PolynomF_2N[res.Count];
                res.CopyTo(G);
                S = Mod(S_Pol(G[var[0]], G[var[1]]), G);
                if (S.list.Count != 0)
                {
                    t++;
                    res.Add(S);
                    for (int j = 0; j < t; j++)
                    {
                        B.Add(new int[] { j, t - 1 });
                    }
                }
            }
            B.RemoveAt(0);
        }
        //the basis is constructed, now we reduce it
        int p = 0;
        do
        {
            G = new PolynomF_2N[res.Count - 1];
            for (int i = 0; i < p; i++) { G[i] = res[i]; }
            for (int i = p + 1; i < res.Count; i++) { G[i - 1] = res[i]; }
            PolynomF_2N P = new PolynomF_2N(res[p].sortWay);
            for (int i = 0; i < res[p].list.Count; i++)
            {
                FieldF_2N coef = new FieldF_2N(res[p].GetCoefficient(i));
```

```
                Monom mon = new Monom(res[p].GetMonom(i));
                P.Add(mon,coef);
            }
            PolynomF_2N S = Mod(P, G);
            if (S.list.Count == 0) { res.RemoveAt(p); } else { res[p] = S.Simplify();
p++; }
        } while (p != res.Count);
        return res;
    }
public static void ReduceBasis(List<PolynomQ> GB)
    {
        int p = 0;
        do
        {
            PolynomQ[] G = new PolynomQ[GB.Count - 1];
            for (int i = 0; i < p; i++) { G[i] = GB[i]; }
            for (int i = p + 1; i < GB.Count; i++) { G[i - 1] = GB[i]; }
            PolynomQ P = new PolynomQ(GB[p].sortWay);
            for (int i = 0; i < GB[p].list.Count; i++)
            {
                FieldQ coef = new FieldQ(GB[p].GetCoefficient(i));
                Monom mon = new Monom(GB[p].GetMonom(i));
                P.Add(mon, coef);
            }
            PolynomQ S = Mod(P, G);
            if (S.list.Count == 0) { GB.RemoveAt(p); } else { GB[p] = S.Simplify();
p++; }
        } while (p != GB.Count);
    }

    public static void ReduceBasis(List<PolynomF_2N> GB)
    {
        int p = 0;
        do
        {
            PolynomF_2N[] G = new PolynomF_2N[GB.Count - 1];
            for (int i = 0; i < p; i++) { G[i] = GB[i]; }
            for (int i = p + 1; i < GB.Count; i++) { G[i - 1] = GB[i]; }
            PolynomF_2N P = new PolynomF_2N(GB[p].sortWay);
            for (int i = 0; i < GB[p].list.Count; i++)
            {
                FieldF_2N coef = new FieldF_2N(GB[p].GetCoefficient(i));
                Monom mon = new Monom(GB[p].GetMonom(i));
                P.Add(mon, coef);
            }
            PolynomF_2N S = Mod(P, G);
            if (S.list.Count == 0) { GB.RemoveAt(p); } else { GB[p] = S.Simplify();
p++; }
        } while (p != GB.Count);
    }

    public static List<PolynomQ> MakeGrebnerBasis(PolynomQ[] F)
    {
        List<int[]> B = new List<int[]>();
        for (int i = 0; i < F.Length; i++)
        {
            for (int j = i + 1; j < F.Length; j++)
            {
                B.Add(new int[] { i, j });
            }
        }
        List<PolynomQ> res = new List<PolynomQ>(); PolynomQ[] G;
        for (int i = 0; i < F.Length; i++) { res.Add(F[i]); }
        int t = F.Length;
```

```
        while (B.Count != 0)
        {
            int[] var = B[0];
            Monom lcm = Monom.LCM(res[var[0]].LM(), res[var[1]].LM());
            Monom prod = Monom.Prod(res[var[0]].LM(), res[var[1]].LM());
            bool eq = true; bool crit = false;
            int i = 0;
            while (eq && (i < lcm.N))
            {
                if (lcm.Sign[i] != prod.Sign[i]) { eq = false; } else { i++; }
            }
            crit = Criteria(var[0], var[1], B, res, lcm);
            if (!eq && !crit)
            {
                PolynomQ S;
                G = new PolynomQ[res.Count];
                res.CopyTo(G);
                S = Mod(S_Pol(G[var[0]], G[var[1]]), G);
                if (S.list.Count != 0)
                {
                    t++;
                    res.Add(S);
                    for (int j = 0; j < t; j++)
                    {
                        B.Add(new int[] { j, t - 1 });
                    }
                }
            }
            B.RemoveAt(0);
        }
        int p = 0;
        do
        {
            G = new PolynomQ[res.Count - 1];
            for (int i = 0; i < p; i++) { G[i] = res[i]; }
            for (int i = p + 1; i < res.Count; i++) { G[i - 1] = res[i]; }
            PolynomQ P = new PolynomQ(res[p].sortWay);
            for (int i = 0; i < res[p].list.Count; i++)
            {
                FieldQ coef = new FieldQ(res[p].GetCoefficient(i));
                Monom mon = new Monom(res[p].GetMonom(i));
                P.Add(mon,coef);
            }
            PolynomQ S = Mod(P, G);
            if (S.list.Count == 0) { res.RemoveAt(p); } else { res[p] = S.Simplify();
p++; }
        } while (p != res.Count);
        return res;
    }
//Construction of the system of n nonlimear equations of n variables in the field F_2
based on the equation in the field F_2N
    public static PolynomF_2N[] MakeSystem(PolynomF_2N eq, BitArray prim, int sortWay,
out int[] Degs)
    {
        if (eq.list.Count == 0) { throw new ArgumentException(); }
        //we obtain the normal basis, and the inverse matrix of corresponding
transformation matrix
        int i = 1; Degs = new int[prim.Length - 1];
        bool found = false; bool[,] matr = null, invmatr;
        while (!found)
        {
            int k = 1;
            for (int j = 0; j < Degs.Length; j++)
            {
```

```
                Degs[j] = i * k; k = k * 2;
            }
            matr = FieldF_2N.MakeBasisMatrix(Degs, prim);
            if (FieldF_2N.IsBasis(matr)) { found = true; } else { i++; }
        }
        invmatr = FieldF_2N.Inv(matr);
        //ans is the representation of x in the basis, exteq is the result of
substitutioning x into the equation
        PolynomF_2N ans = new PolynomF_2N(sortWay);
        PolynomF_2N exteq = new PolynomF_2N(sortWay);
        for (i = 0; i < Degs.Length; i++)
        {
            int[] sgn = new int[Degs.Length];
            sgn[i] = 1;
            ans.Add(new Monom(sgn), new FieldF_2N(prim,
FieldF_2N.Convert(2)).Pow(Degs[i]));
        }
        for (i = 0; i < eq.list.Count; i++)
        {
            int deg = eq.GetMonom(i).Sign[0];
            FieldF_2N coef = eq.GetCoefficient(i);
            exteq = ans.Pow(deg).Mult(coef) + exteq;
            exteq = exteq.UnifyDegrees();
        }
        //the construction of the system
        PolynomF_2N[] Sys = new PolynomF_2N[2*Degs.Length];
        for (i = 0; i < Sys.Length; i++)
        {
            Sys[i] = new PolynomF_2N(sortWay);
        }
        for (i = 0; i < exteq.list.Count; i++)
        {
            FieldF_2N coef = exteq.list.Values[i].InNewBasis(invmatr);
            for (int j = 0; j < Degs.Length; j++)
            {
                if (coef.GetVal(j)) { Sys[j].Add(exteq.GetMonom(i), new FieldF_2N(2,
1)); }
            }
        }
        for (i = 0; i < Degs.Length; i++)
        {
            int[] sgn1 = new int[Degs.Length], sgn2 = new int[Degs.Length];
            sgn1[i] = 2; sgn2[i] = 1;
            Sys[i + Degs.Length].Add(new Monom(sgn1), new FieldF_2N(2, 1));
            Sys[i + Degs.Length].Add(new Monom(sgn2), new FieldF_2N(2, 1));
        }
        return Sys;
    }
}
```

The module Faugere/Signature

This module contains the implementation of polynomial signatures for the F5-family algorithms for the Grobner's bases construction. The module provides the comparison and multiplication operations. Also, the module contains different compares, based on which the F5 algorithm compares the signatures.

```
public class Signature
{
    private int N;
    public Monom Coefficient;
```

```
    public int PolynomIndex;

    public Signature(Monom coef, int index)
    {
        this.N = coef.N;
        this.Coefficient = new Monom(coef);
        this.PolynomIndex = index;
    }

    public Signature(int N, int index)
    {
        this.N = N;
        this.Coefficient = new Monom(new int[N]);
        this.PolynomIndex = index;
    }

    public Signature Prod(Monom mon)
    {
        return new Signature(Monom.Prod(mon, Coefficient), PolynomIndex);
    }

    public bool IsBetterThan(int sortWay, Signature s2)
    {
        CompareSignGrevLex Comp = new CompareSignGrevLex();
        switch (sortWay)
        {
            case 1:
                return (Comp.Compare(this, s2) > 0);
            case 2:
                return (Comp.Compare(this, s2) > 0);
            case 3:
                return (Comp.Compare(this, s2) > 0);
            case 4:
                return (Comp.Compare(this, s2) > 0);
            case 5:
                return (Comp.Compare(this, s2) > 0);
            default:
                return false;
        }
    }
    public override bool Equals(object obj)
    {
        Signature Sobj = obj as Signature;
        if (Sobj == null) return false;
        return (PolynomIndex == Sobj.PolynomIndex) &&
Coefficient.Equals(Sobj.Coefficient);
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}
```

The module Faugere/CriticalPair

This module contains an implementation of critical pair's structure, which is used by the F5 algorithm. Also, the module contains different comparers, based on which the algorithm arranges its own list of critical pairs.

```
public class CritPairQ
```

```
{
    public Monom LCM;
    public Monom U_1;
    public PolyQWithSign R_1;
    public Monom U_2;
    public PolyQWithSign R_2;

    public static CritPairQ Get(PolyQWithSign r_1, PolyQWithSign r_2, int k,
List<PolynomQ> G_last)
    {
        CritPairQ cp = new CritPairQ();
        PolynomQ p_1 = r_1.Poly; PolynomQ p_2 = r_2.Poly;
        int sortWay = p_1.sortWay;
        cp.LCM = Monom.LCM(p_1.LM(), p_2.LM());
        cp.U_1 = Monom.Div(cp.LCM, p_1.LM());
        cp.U_2 = Monom.Div(cp.LCM, p_2.LM());
        cp.R_1 = r_1; cp.R_2 = r_2;

        if (r_1.Sign.Prod(cp.U_1).IsBetterThan(sortWay, r_2.Sign.Prod(cp.U_2))) return
Get(r_2, r_1, k, G_last);

        Signature s_1 = r_1.Sign; Signature s_2 = r_2.Sign;
        if (s_1.PolynomIndex > k) return null;

        PolynomQ pol1 = new PolynomQ(p_1.sortWay);
        PolynomQ pol2 = new PolynomQ(p_2.sortWay);
        pol1.Add(Monom.Prod(cp.U_1, s_1.Coefficient), new FieldQ(1));
        pol2.Add(Monom.Prod(cp.U_2, s_2.Coefficient), new FieldQ(1));

        PolynomQ NFpol1 = Grebner.Mod(pol1, G_last.ToArray());
        if (!NFpol1.Equals(pol1)) return null;

        if (s_2.PolynomIndex == k)
        {
            PolynomQ NFpol2 = Grebner.Mod(pol2, G_last.ToArray());
            if (!NFpol2.Equals(pol2)) return null;
        }

        return cp;
    }
}
public class CritPairF_2N
{
    public Monom LCM;
    public Monom U_1;
    public PolyF_2NWithSign R_1;
    public Monom U_2;
    public PolyF_2NWithSign R_2;

    public static CritPairF_2N Get(PolyF_2NWithSign r_1, PolyF_2NWithSign r_2, int k,
List<PolynomF_2N> G_last)
    {
        CritPairF_2N cp = new CritPairF_2N();
        PolynomF_2N p_1 = r_1.Poly; PolynomF_2N p_2 = r_2.Poly;
        int sortWay = p_1.sortWay;
        cp.LCM = Monom.LCM(p_1.LM(), p_2.LM());
        cp.U_1 = Monom.Div(cp.LCM, p_1.LM());
        cp.U_2 = Monom.Div(cp.LCM, p_2.LM());
        cp.R_1 = r_1; cp.R_2 = r_2;

        if (r_1.Sign.Prod(cp.U_1).IsBetterThan(sortWay, r_2.Sign.Prod(cp.U_2))) return
Get(r_2, r_1, k, G_last);

        Signature s_1 = r_1.Sign; Signature s_2 = r_2.Sign;
```

```
        if (s_1.PolynomIndex > k) return null;

        PolynomF_2N pol1 = new PolynomF_2N(p_1.sortWay);
        PolynomF_2N pol2 = new PolynomF_2N(p_2.sortWay);
        var field = p_1.LC().Prim;
        pol1.Add(Monom.Prod(cp.U_1, s_1.Coefficient), new FieldF_2N(field,
FieldF_2N.Convert(1)));
        pol2.Add(Monom.Prod(cp.U_2, s_2.Coefficient), new FieldF_2N(field,
FieldF_2N.Convert(1)));

        PolynomF_2N NFpol1 = Grebner.Mod(pol1, G_last.ToArray());
        if (!NFpol1.Equals(pol1)) return null;

        if (s_2.PolynomIndex == k)
        {
            PolynomF_2N NFpol2 = Grebner.Mod(pol2, G_last.ToArray());
            if (!NFpol2.Equals(pol2)) return null;
        }

        return cp;
    }
}

public class CompareCritPairQ : IComparer<CritPairQ>
{
    public int Compare(CritPairQ s1, CritPairQ s2)
    {
        int a = s1.LCM.FullDeg();
        int b = s2.LCM.FullDeg();
        if (a > b) return 1;
        if (a < b) return -1;
        switch (s1.R_1.Poly.sortWay)
        {
            case 1:
                return new CompareLex().Compare(s1.LCM, s2.LCM);
            case 3:
                return new CompareGrevLex().Compare(s1.LCM, s2.LCM);
            default:
                return 0;
        }
    }
}

public class CompareCritPairF_2N : IComparer<CritPairF_2N>
{
    public int Compare(CritPairF_2N s1, CritPairF_2N s2)
    {
        int a = s1.LCM.FullDeg();
        int b = s2.LCM.FullDeg();
        if (a > b) return 1;
        if (a < b) return -1;
        switch (s1.R_1.Poly.sortWay)
        {
            case 1:
                return new CompareLex().Compare(s1.LCM, s2.LCM);
            case 3:
                return new CompareGrevLex().Compare(s1.LCM, s2.LCM);
            default:
                return 0;
        }
    }
}
```

The module Faugere/RuleF5.cs

This module contains an implementation of expression rules, which are used by the F5 algorithm.

```
public class RuleF5
{
    public Monom Monom;
    public int Index;

    public RuleF5(Monom mon, int index)
    {
        Monom = mon;
        Index = index;
    }
}
```

The module Faugere/F5

There is versions of F5's implementations for the fields Q and F_{2^n}

```
public static class FaugereF5F_2N
{
    private static List<RuleF5>[] Rules;
    private static Dictionary<int, PolyF_2NWithSign> Nums;
    private static Dictionary<PolyF_2NWithSign, int> Pols;
    static int K;

    private static void ResetRules(int M)
    {
        Rules = new List<RuleF5>[M];
        Nums = new Dictionary<int, PolyF_2NWithSign>();
        Pols = new Dictionary<PolyF_2NWithSign, int>();
        for (int i = 0; i < M; i++)
        {
            Rules[i] = new List<RuleF5>();
        }
    }

    private static void AddRule(int K, PolyF_2NWithSign r_N)
    {
        Rules[r_N.Sign.PolynomIndex].Insert(0, new RuleF5(r_N.Sign.Coefficient, K));
    }

    private static Monom Rewritten(Monom u, PolyF_2NWithSign r_k, out int r_o)
    {
        List<RuleF5> L = Rules[r_k.Sign.PolynomIndex];
        for (int i = 0; i < L.Count; i++)
        {
            Monom ut = Monom.Prod(u, r_k.Sign.Coefficient);
            if (Monom.IsDivisible(ut, L[i].Monom))
            {
                Monom v = Monom.Div(ut, L[i].Monom);
                r_o = L[i].Index;
                return Monom.Div(ut, L[i].Monom);
            }
        }
        r_o = Pols[r_k];
        return u;
    }
    private static bool IsRewritten(Monom u, int r_k)
    {
        int r_l;
```

```
        Monom v = Rewritten(u, Nums[r_k], out r_l);
        return r_l != r_k;
    }
    private static List<PolyF_2NWithSign> Spol(List<CritPairF_2N> P)
    {
        List<PolyF_2NWithSign> F = new List<PolyF_2NWithSign>();
        for (int i = 0; i < P.Count; i++)
        {
            if (!IsRewritten(P[i].U_1, Pols[P[i].R_1]) && !IsRewritten(P[i].U_2,
Pols[P[i].R_2]))
            {
                var field = P[i].R_1.Poly.LC().Prim;

                PolynomF_2N pol1 = new PolynomF_2N(P[i].R_1.Poly.sortWay);
                pol1.Add(P[i].U_1, new FieldF_2N(field, FieldF_2N.Convert(1)));

                PolynomF_2N pol2 = new PolynomF_2N(P[i].R_2.Poly.sortWay);
                pol2.Add(P[i].U_2, new FieldF_2N(field, FieldF_2N.Convert(1)));

                K++;
                PolyF_2NWithSign r_N = new
PolyF_2NWithSign(P[i].R_1.Sign.Prod(P[i].U_1), P[i].R_1.Poly * pol1 + (P[i].R_2.Poly *
pol2));
                AddRule(K, r_N);
                F.Add(r_N);
            }
        }
        F.Sort(new ComparePolyF_2NSign());
        return F;
    }

    private static PolyF_2NWithSign IsReducible(PolyF_2NWithSign r_i0,
List<PolyF_2NWithSign> G, int k, List<PolyF_2NWithSign> G_last)
    {
        for (int i = 0; i < G.Count; i++)
        {
            bool a = Monom.IsDivisible(r_i0.Poly.LM(), G[i].Poly.LM());
            if (a)
            {
                Monom u = Monom.Div(r_i0.Poly.LM(), G[i].Poly.LM());
                PolynomF_2N pol = new PolynomF_2N(r_i0.Poly.sortWay);
                var field = r_i0.Poly.LC().Prim;
                pol.Add(Monom.Prod(u, G[i].Sign.Coefficient), new FieldF_2N(field,
FieldF_2N.Convert(1)));
                bool b = Grebner.Mod(pol, G_last.Select(x =>
x.Poly).ToArray()).Equals(pol);
                if (b)
                {
                    Signature s = new Signature(Monom.Prod(u, G[i].Sign.Coefficient),
G[i].Sign.PolynomIndex);
                    bool c = s.Equals(r_i0.Sign);
                    if (c)
                    {
                        bool d = (!IsRewritten(u, Pols[G[i]]));
                        if (d) return G[i];
                    }
                }
            }
        }
        return null;
    }
    private static PolyF_2NWithSign TopReduction(PolyF_2NWithSign r_k0,
List<PolyF_2NWithSign> G, int k, List<PolyF_2NWithSign> G_last, out
List<PolyF_2NWithSign> ToDo_1)
```

```
    {
        PolyF_2NWithSign r = IsReducible(r_k0, G, k, G_last);
        if (r == null)
        {
            ToDo_1 = new List<PolyF_2NWithSign>();
            return new PolyF_2NWithSign(r_k0.Sign, r_k0.Poly.Simplify());
        }
        else
        {
            Monom u = Monom.Div(r_k0.Poly.LM(), r.Poly.LM());
            if (r.Sign.Prod(u).IsBetterThan(r.Poly.sortWay, r_k0.Sign))
            {
                PolynomF_2N pol = new PolynomF_2N(r_k0.Poly.sortWay);
                var field = r_k0.Poly.LC().Prim;
                pol.Add(u, new FieldF_2N(field, FieldF_2N.Convert(1)));
                r_k0.Poly = r_k0.Poly + (r.Poly * pol);
                ToDo_1 = new List<PolyF_2NWithSign>();
                ToDo_1.Add(r_k0);
                return null;
            }
            else
            {
                K++;
                PolynomF_2N pol = new PolynomF_2N(r_k0.Poly.sortWay);
                var field = r_k0.Poly.LC().Prim;
                pol.Add(u, new FieldF_2N(field, FieldF_2N.Convert(1)));
                PolyF_2NWithSign r_N = new PolyF_2NWithSign(r.Sign.Prod(u), r.Poly *
pol + (-r_k0.Poly));
                AddRule(K, r_N);
                ToDo_1 = new List<PolyF_2NWithSign>();
                ToDo_1.Add(r_N);
                ToDo_1.Add(r_k0);
                return null;
            }
        }

    }
    private static List<PolyF_2NWithSign> Reduction(List<PolyF_2NWithSign> ToDo,
List<PolyF_2NWithSign> G_i, int k, List<PolyF_2NWithSign> G_last)
    {
        List<PolyF_2NWithSign> Done = new List<PolyF_2NWithSign>();
        while (ToDo.Count != 0)
        {
            PolyF_2NWithSign h = ToDo[0];
            ToDo.RemoveAt(0);
            PolyF_2NWithSign phi_h = new PolyF_2NWithSign(h.Sign, Grebner.Mod(h.Poly,
G_last.Select(x => x.Poly).ToArray()));
            List<PolyF_2NWithSign> ToDo_1 = new List<PolyF_2NWithSign>();
            if (phi_h.Poly.list.Count == 0) continue;
            PolyF_2NWithSign h_1 = TopReduction(phi_h, G_i.Union(Done).ToList(), k,
G_last, out ToDo_1);
            if (h_1 != null) Done.Add(h_1);
            ToDo.AddRange(ToDo_1);
            ToDo.Sort(new ComparePolyF_2NSign());
        }
        return Done;
    }
    private static List<PolyF_2NWithSign> F5(int i, PolynomF_2N f_i,
List<PolyF_2NWithSign> G_last)
    {
        List<PolyF_2NWithSign> G_i = new List<PolyF_2NWithSign>();
        int N = G_last[0].Poly.VarCnt;
        int Cnt = K + 1;
        PolyF_2NWithSign r_i = new PolyF_2NWithSign(new Signature(N, i), f_i);
```

```
            G_i.AddRange(G_last);
            G_i.Add(r_i);
            Nums.Add(r_i.Sign.PolynomIndex, r_i);
            Pols.Add(r_i, r_i.Sign.PolynomIndex);
            List<CritPairF_2N> P = new List<CritPairF_2N>();
            foreach (PolyF_2NWithSign r in G_last)
            {
                CritPairF_2N cp = CritPairF_2N.Get(r_i, r, i, G_last.Select(x =>
x.Poly).ToList());
                if (cp != null) P.Add(cp);
            }
            CompareCritPairF_2N Comp = new CompareCritPairF_2N();
            P.Sort(Comp);
            while (P.Count != 0)
            {
                int d = P[0].LCM.FullDeg();
                List<CritPairF_2N> P_d = P.FindAll(p => p.LCM.FullDeg() == d);
                P_d.Sort(Comp);
                P.RemoveAll(p => p.LCM.FullDeg() == d);
                List<PolyF_2NWithSign> F = Spol(P_d);
                List<PolyF_2NWithSign> R_d = Reduction(F, G_i, i, G_last);
                foreach (var r in R_d)
                {
                    foreach (var p in G_i)
                    {
                        CritPairF_2N cp = CritPairF_2N.Get(r, p, i, G_last.Select(x =>
x.Poly).ToList());
                        if (cp != null) P.Add(cp);
                    }
                    G_i.Add(r);
                    Nums.Add(Cnt, r);
                    Pols.Add(r, Cnt);
                    Cnt++;
                }
                P.Sort(Comp);
            }
            return G_i;
        }
        public static List<PolynomF_2N> MakeGrebnerBase(PolynomF_2N[] F)
        {
            int M = F.Length;
            ResetRules(M);
            int N = F[0].VarCnt;
            K = M - 1;
            List<PolyF_2NWithSign>[] G = new List<PolyF_2NWithSign>[M];
            G[M - 1] = new List<PolyF_2NWithSign>();
            G[M - 1].Add(new PolyF_2NWithSign(new Signature(N, M - 1), F[M - 1]));
            Nums.Add(K, G[M - 1][0]);
            Pols.Add(G[M - 1][0], K);
            for (int i = M - 2; i >= 0; i--)
            {
                G[i] = F5(i, F[i], G[i + 1]);
            }
            List<PolynomF_2N> res = new List<PolynomF_2N>();
            for (int i = 0; i < G[0].Count; i++)
            {
                res.Add(G[0][i].Poly);
            }
            Grebner.ReduceBasis(res);
            return res;
        }
    }

    public static class FaugereF5Q
    {
```

```
    private static List<RuleF5>[] Rules;
    private static Dictionary<int, PolyQWithSign> Nums;
    private static Dictionary<PolyQWithSign, int> Pols;
    static int K;

    private static void ResetRules(int M)
    {
        Rules = new List<RuleF5>[M];
        Nums = new Dictionary<int, PolyQWithSign>();
        Pols = new Dictionary<PolyQWithSign, int>();
        for (int i = 0; i < M; i++)
        {
            Rules[i] = new List<RuleF5>();
        }
    }

    private static void AddRule(int K, PolyQWithSign r_N)
    {
        Rules[r_N.Sign.PolynomIndex].Insert(0, new RuleF5(r_N.Sign.Coefficient, K));
    }

    private static Monom Rewritten(Monom u, PolyQWithSign r_k, out int r_o)
    {
        List<RuleF5> L = Rules[r_k.Sign.PolynomIndex];
        for (int i = 0; i < L.Count; i++)
        {
            Monom ut = Monom.Prod(u, r_k.Sign.Coefficient);
            if (Monom.IsDivisible(ut, L[i].Monom))
            {
                Monom v = Monom.Div(ut, L[i].Monom);
                r_o = L[i].Index;
                return Monom.Div(ut, L[i].Monom);
            }
        }
        r_o = Pols[r_k];
        return u;
    }
    private static bool IsRewritten(Monom u, int r_k)
    {
        int r_l;
        Monom v = Rewritten(u, Nums[r_k], out r_l);
        return r_l != r_k;
    }
    private static List<PolyQWithSign> Spol(List<CritPairQ> P)
    {
        List<PolyQWithSign> F = new List<PolyQWithSign>();
        for (int i = 0; i < P.Count; i++)
        {
            if (!IsRewritten(P[i].U_1, Pols[P[i].R_1]) && !IsRewritten(P[i].U_2,
Pols[P[i].R_2]))
            {
                PolynomQ pol1 = new PolynomQ(P[i].R_1.Poly.sortWay);
                pol1.Add(P[i].U_1, new FieldQ(1));

                PolynomQ pol2 = new PolynomQ(P[i].R_2.Poly.sortWay);
                pol2.Add(P[i].U_2, new FieldQ(-1));

                K++;
                PolyQWithSign r_N = new PolyQWithSign(P[i].R_1.Sign.Prod(P[i].U_1),
P[i].R_1.Poly * pol1 + (P[i].R_2.Poly * pol2));
                AddRule(K, r_N);
                F.Add(r_N);
            }
        }
```

```
        F.Sort(new ComparePolyQSign());
        return F;
    }

    private static PolyQWithSign IsReducible(PolyQWithSign r_i0, List<PolyQWithSign>
G, int k, List<PolyQWithSign> G_last)
    {
        for (int i = 0; i < G.Count; i++)
        {
            bool a = Monom.IsDivisible(r_i0.Poly.LM(), G[i].Poly.LM());
            if (a)
            {
                Monom u = Monom.Div(r_i0.Poly.LM(), G[i].Poly.LM());
                PolynomQ pol = new PolynomQ(r_i0.Poly.sortWay);
                pol.Add(Monom.Prod(u, G[i].Sign.Coefficient), new FieldQ(1));
                bool b = Grebner.Mod(pol, G_last.Select(x =>
x.Poly).ToArray()).Equals(pol);
                if (b)
                {
                    bool c = (!IsRewritten(u, Pols[G[i]]));
                    if (c)
                    {
                        Signature s = new Signature(Monom.Prod(u,
G[i].Sign.Coefficient), G[i].Sign.PolynomIndex);
                        bool d = s.Equals(r_i0.Sign);
                        if (d) return G[i];
                    }
                }
            }
        }
        return null;
    }
    private static PolyQWithSign TopReduction(PolyQWithSign r_k0, List<PolyQWithSign>
G, int k, List<PolyQWithSign> G_last, out List<PolyQWithSign> ToDo_1)
    {
        PolyQWithSign r = IsReducible(r_k0, G, k, G_last);
        if (r == null)
        {
            ToDo_1 = new List<PolyQWithSign>();
            return new PolyQWithSign(r_k0.Sign, r_k0.Poly.Simplify());
        }
        else
        {
            Monom u = Monom.Div(r_k0.Poly.LM(), r.Poly.LM());
            if (r.Sign.Prod(u).IsBetterThan(r.Poly.sortWay, r_k0.Sign))
            {
                PolynomQ pol = new PolynomQ(r_k0.Poly.sortWay);
                pol.Add(u, new FieldQ(-1));
                r_k0.Poly = r_k0.Poly + (r.Poly * pol);
                ToDo_1 = new List<PolyQWithSign>();
                ToDo_1.Add(r_k0);
                return null;
            }
            else
            {
                K++;
                PolynomQ pol = new PolynomQ(r_k0.Poly.sortWay);
                pol.Add(u, new FieldQ(1));
                PolyQWithSign r_N = new PolyQWithSign(r.Sign.Prod(u), r.Poly * pol +
(-r_k0.Poly));
                AddRule(K, r_N);
                ToDo_1 = new List<PolyQWithSign>();
                ToDo_1.Add(r_N);
                ToDo_1.Add(r_k0);
```

```
                return null;
            }
        }

    }
    private static List<PolyQWithSign> Reduction(List<PolyQWithSign> ToDo,
List<PolyQWithSign> G_i, int k, List<PolyQWithSign> G_last)
    {
        List<PolyQWithSign> Done = new List<PolyQWithSign>();
        while (ToDo.Count != 0)
        {
            PolyQWithSign h = ToDo[0];
            ToDo.RemoveAt(0);
            PolyQWithSign phi_h = new PolyQWithSign(h.Sign, Grebner.Mod(h.Poly,
G_last.Select(x => x.Poly).ToArray()));
            List<PolyQWithSign> ToDo_1 = new List<PolyQWithSign>();
            PolyQWithSign h_1 = TopReduction(phi_h, G_i.Union(Done).ToList(), k,
G_last, out ToDo_1);
            if (h_1 != null) Done.Add(h_1);
            ToDo.AddRange(ToDo_1);
            ToDo.Sort(new ComparePolyQSign());
        }
        return Done;
    }
    private static List<PolyQWithSign> F5(int i, PolynomQ f_i, List<PolyQWithSign>
G_last)
    {
        List<PolyQWithSign> G_i = new List<PolyQWithSign>();
        int N = G_last[0].Poly.VarCnt;
        int Cnt = K + 1;
        PolyQWithSign r_i = new PolyQWithSign(new Signature(N, i), f_i);
        G_i.AddRange(G_last);
        G_i.Add(r_i);
        Nums.Add(r_i.Sign.PolynomIndex, r_i);
        Pols.Add(r_i, r_i.Sign.PolynomIndex);
        List<CritPairQ> P = new List<CritPairQ>();
        foreach (PolyQWithSign r in G_last)
        {
            CritPairQ cp = CritPairQ.Get(r_i, r, i, G_last.Select(x =>
x.Poly).ToList());
            if (cp != null) P.Add(cp);
        }
        CompareCritPairQ Comp = new CompareCritPairQ();
        P.Sort(Comp);
        while (P.Count != 0)
        {
            int d = P[0].LCM.FullDeg();
            List<CritPairQ> P_d = P.FindAll(p => p.LCM.FullDeg() == d);
            P_d.Sort(Comp);
            P.RemoveAll(p => p.LCM.FullDeg() == d);
            List<PolyQWithSign> F = Spol(P_d);
            List<PolyQWithSign> R_d = Reduction(F, G_i, i, G_last);
            foreach (var r in R_d)
            {
                foreach (var p in G_i)
                {
                    CritPairQ cp = CritPairQ.Get(r, p, i, G_last.Select(x =>
x.Poly).ToList());
                    if (cp != null) P.Add(cp);
                }
                G_i.Add(r);
                Nums.Add(Cnt, r);
                Pols.Add(r, Cnt);
                Cnt++;
```

```
            }
            P.Sort(Comp);
        }
        return G_i;
    }
    public static List<PolynomQ> MakeGrebnerBase(PolynomQ[] F)
    {
        int M = F.Length;
        ResetRules(M);
        int N = F[0].VarCnt;
        K = M - 1;
        List<PolyQWithSign>[] G = new List<PolyQWithSign>[M];
        G[M - 1] = new List<PolyQWithSign>();
        G[M - 1].Add(new PolyQWithSign(new Signature(N, M - 1), F[M - 1]));
        Nums.Add(K, G[M - 1][0]);
        Pols.Add(G[M - 1][0], K);
        for (int i = M - 2; i >= 0; i--)
        {
            G[i] = F5(i, F[i], G[i + 1]);
        }
        List<PolynomQ> res = new List<PolynomQ>();
        for (int i = 0; i < G[0].Count; i++)
        {
            res.Add(G[0][i].Poly);
        }
        Grebner.ReduceBasis(res);
        return res;
    }
}
```

The module Faugere/F5R

There is versions of F5R's implementations for the fields Q and F_{2^n}

```
public static class FaugereF5RF_2N
{
    private static List<RuleF5>[] Rules;
    private static Dictionary<int, PolyF_2NWithSign> Nums;
    private static Dictionary<PolyF_2NWithSign, int> Pols;
    static int K;

    private static void ResetRules(int M)
    {
        Rules = new List<RuleF5>[M];
        Nums = new Dictionary<int, PolyF_2NWithSign>();
        Pols = new Dictionary<PolyF_2NWithSign, int>();
        for (int i = 0; i < M; i++)
        {
            Rules[i] = new List<RuleF5>();
        }
    }

    private static void AddRule(int K, PolyF_2NWithSign r_N)
    {
        Rules[r_N.Sign.PolynomIndex].Insert(0, new RuleF5(r_N.Sign.Coefficient, K));
    }

    private static Monom Rewritten(Monom u, PolyF_2NWithSign r_k, out int r_o)
    {
        List<RuleF5> L = Rules[r_k.Sign.PolynomIndex];
        for (int i = 0; i < L.Count; i++)
        {
            Monom ut = Monom.Prod(u, r_k.Sign.Coefficient);
            if (Monom.IsDivisible(ut, L[i].Monom))
```

```
            {
                Monom v = Monom.Div(ut, L[i].Monom);
                r_o = L[i].Index;
                return Monom.Div(ut, L[i].Monom);
            }
        }
        r_o = Pols[r_k];
        return u;
    }
    private static bool IsRewritten(Monom u, int r_k)
    {
        int r_l;
        Monom v = Rewritten(u, Nums[r_k], out r_l);
        return r_l != r_k;
    }
    private static List<PolyF_2NWithSign> Spol(List<CritPairF_2N> P)
    {
        List<PolyF_2NWithSign> F = new List<PolyF_2NWithSign>();
        for (int i = 0; i < P.Count; i++)
        {
            if (!IsRewritten(P[i].U_1, Pols[P[i].R_1]) && !IsRewritten(P[i].U_2,
Pols[P[i].R_2]))
            {
                var field = P[i].R_1.Poly.LC().Prim;

                PolynomF_2N pol1 = new PolynomF_2N(P[i].R_1.Poly.sortWay);
                pol1.Add(P[i].U_1, new FieldF_2N(field, FieldF_2N.Convert(1)));

                PolynomF_2N pol2 = new PolynomF_2N(P[i].R_2.Poly.sortWay);
                pol2.Add(P[i].U_2, new FieldF_2N(field, FieldF_2N.Convert(1)));

                K++;
                PolyF_2NWithSign r_N = new
PolyF_2NWithSign(P[i].R_1.Sign.Prod(P[i].U_1), P[i].R_1.Poly * pol1 + (P[i].R_2.Poly *
pol2));
                AddRule(K, r_N);
                F.Add(r_N);
            }
        }
        F.Sort(new ComparePolyF_2NSign());
        return F;
    }

    private static PolyF_2NWithSign IsReducible(PolyF_2NWithSign r_i0,
List<PolyF_2NWithSign> G, int k, List<PolynomF_2N> B_last)
    {
        for (int i = 0; i < G.Count; i++)
        {
            bool a = Monom.IsDivisible(r_i0.Poly.LM(), G[i].Poly.LM());
            if (a)
            {
                Monom u = Monom.Div(r_i0.Poly.LM(), G[i].Poly.LM());
                PolynomF_2N pol = new PolynomF_2N(r_i0.Poly.sortWay);
                var field = r_i0.Poly.LC().Prim;
                pol.Add(Monom.Prod(u, G[i].Sign.Coefficient), new FieldF_2N(field,
FieldF_2N.Convert(1)));
                bool b = Grebner.Mod(pol, B_last.ToArray()).Equals(pol);
                if (b)
                {
                    Signature s = new Signature(Monom.Prod(u, G[i].Sign.Coefficient),
G[i].Sign.PolynomIndex);
                    bool c = s.Equals(r_i0.Sign);
                    if (c)
                    {
```

```
                        bool d = (!IsRewritten(u, Pols[G[i]]));
                        if (d) return G[i];
                    }
                }
            }
        }
        return null;
    }
    private static PolyF_2NWithSign TopReduction(PolyF_2NWithSign r_k0,
List<PolyF_2NWithSign> G, int k, List<PolynomF_2N> B_last, out List<PolyF_2NWithSign>
ToDo_1)
    {
        PolyF_2NWithSign r = IsReducible(r_k0, G, k, B_last);
        if (r == null)
        {
            ToDo_1 = new List<PolyF_2NWithSign>();
            return new PolyF_2NWithSign(r_k0.Sign, r_k0.Poly.Simplify());
        }
        else
        {
            Monom u = Monom.Div(r_k0.Poly.LM(), r.Poly.LM());
            if (r.Sign.Prod(u).IsBetterThan(r.Poly.sortWay, r_k0.Sign))
            {
                PolynomF_2N pol = new PolynomF_2N(r_k0.Poly.sortWay);
                var field = r_k0.Poly.LC().Prim;
                pol.Add(u, new FieldF_2N(field, FieldF_2N.Convert(1)));
                r_k0.Poly = r_k0.Poly + (r.Poly * pol);
                ToDo_1 = new List<PolyF_2NWithSign>();
                ToDo_1.Add(r_k0);
                return null;
            }
            else
            {
                K++;
                PolynomF_2N pol = new PolynomF_2N(r_k0.Poly.sortWay);
                var field = r_k0.Poly.LC().Prim;
                pol.Add(u, new FieldF_2N(field, FieldF_2N.Convert(1)));
                PolyF_2NWithSign r_N = new PolyF_2NWithSign(r.Sign.Prod(u), r.Poly *
pol + (-r_k0.Poly));
                AddRule(K, r_N);
                ToDo_1 = new List<PolyF_2NWithSign>();
                ToDo_1.Add(r_N);
                ToDo_1.Add(r_k0);
                return null;
            }
        }

    }
    private static List<PolyF_2NWithSign> Reduction(List<PolyF_2NWithSign> ToDo,
List<PolyF_2NWithSign> G_i, int k, List<PolynomF_2N> B_last)
    {
        List<PolyF_2NWithSign> Done = new List<PolyF_2NWithSign>();
        while (ToDo.Count != 0)
        {
            PolyF_2NWithSign h = ToDo[0];
            ToDo.RemoveAt(0);
            PolyF_2NWithSign phi_h = new PolyF_2NWithSign(h.Sign, Grebner.Mod(h.Poly,
B_last.ToArray()));
            List<PolyF_2NWithSign> ToDo_1 = new List<PolyF_2NWithSign>();
            if (phi_h.Poly.list.Count == 0) continue;
            PolyF_2NWithSign h_1 = TopReduction(phi_h, G_i.Union(Done).ToList(), k,
B_last, out ToDo_1);
            if (h_1 != null) Done.Add(h_1);
            ToDo.AddRange(ToDo_1);
```

```
            ToDo.Sort(new ComparePolyF_2NSign());
        }
        return Done;
    }
    private static List<PolyF_2NWithSign> F5(int i, PolynomF_2N f_i,
List<PolyF_2NWithSign> G_last, List<PolynomF_2N> B_last)
    {
        List<PolyF_2NWithSign> G_i = new List<PolyF_2NWithSign>();
        int N = G_last[0].Poly.VarCnt;
        int Cnt = K + 1;
        PolyF_2NWithSign r_i = new PolyF_2NWithSign(new Signature(N, i), f_i);
        G_i.AddRange(G_last);
        G_i.Add(r_i);
        Nums.Add(r_i.Sign.PolynomIndex, r_i);
        Pols.Add(r_i, r_i.Sign.PolynomIndex);
        List<CritPairF_2N> P = new List<CritPairF_2N>();
        foreach (PolyF_2NWithSign r in G_last)
        {
            CritPairF_2N cp = CritPairF_2N.Get(r_i, r, i, B_last);
            if (cp != null) P.Add(cp);
        }
        CompareCritPairF_2N Comp = new CompareCritPairF_2N();
        P.Sort(Comp);
        while (P.Count != 0)
        {
            int d = P[0].LCM.FullDeg();
            List<CritPairF_2N> P_d = P.FindAll(p => p.LCM.FullDeg() == d);
            P_d.Sort(Comp);
            P.RemoveAll(p => p.LCM.FullDeg() == d);
            List<PolyF_2NWithSign> F = Spol(P_d);
            List<PolyF_2NWithSign> R_d = Reduction(F, G_i, i, B_last);
            foreach (var r in R_d)
            {
                foreach (var p in G_i)
                {
                    CritPairF_2N cp = CritPairF_2N.Get(r, p, i, B_last);
                    if (cp != null) P.Add(cp);
                }
                G_i.Add(r);
                Nums.Add(Cnt, r);
                Pols.Add(r, Cnt);
                Cnt++;
            }
            P.Sort(Comp);
        }
        return G_i;
    }
    public static List<PolynomF_2N> MakeGrebnerBase(PolynomF_2N[] F)
    {
        int M = F.Length; ResetRules(M);
        int N = F[0].VarCnt; K = M - 1;
        List<PolyF_2NWithSign>[] G = new List<PolyF_2NWithSign>[M];
        G[M - 1] = new List<PolyF_2NWithSign>();
        G[M - 1].Add(new PolyF_2NWithSign(new Signature(N, M - 1), F[M - 1]));
        Nums.Add(K, G[M - 1][0]); Pols.Add(G[M - 1][0], K);
        List<PolynomF_2N> B_last = G[M - 1].Select(x => x.Poly).ToList();
        for (int i = M - 2; i >= 0; i--)
        {
            G[i] = F5(i, F[i], G[i + 1], B_last);
            B_last = new List<PolynomF_2N>();
            for (int j = 0; j < G[i].Count; j++)
            {
                B_last.Add(new PolynomF_2N(G[i][j].Poly));
            }
```

```
            Grebner.ReduceBasis(B_last);
        }
        return B_last;
    }
}
public static class FaugereF5RQ
{
    private static List<RuleF5>[] Rules;
    private static Dictionary<int, PolyQWithSign> Nums;
    private static Dictionary<PolyQWithSign, int> Pols;
    static int K;

    private static void ResetRules(int M)
    {
        Rules = new List<RuleF5>[M];
        Nums = new Dictionary<int, PolyQWithSign>();
        Pols = new Dictionary<PolyQWithSign, int>();
        for (int i = 0; i < M; i++)
        {
            Rules[i] = new List<RuleF5>();
        }
    }

    private static void AddRule(int K, PolyQWithSign r_N)
    {
        Rules[r_N.Sign.PolynomIndex].Insert(0, new RuleF5(r_N.Sign.Coefficient, K));
    }

    private static Monom Rewritten(Monom u, PolyQWithSign r_k, out int r_o)
    {
        List<RuleF5> L = Rules[r_k.Sign.PolynomIndex];
        for (int i = 0; i < L.Count; i++)
        {
            Monom ut = Monom.Prod(u, r_k.Sign.Coefficient);
            if (Monom.IsDivisible(ut, L[i].Monom))
            {
                Monom v = Monom.Div(ut, L[i].Monom);
                r_o = L[i].Index;
                return Monom.Div(ut, L[i].Monom);
            }
        }
        r_o = Pols[r_k];
        return u;
    }
    private static bool IsRewritten(Monom u, int r_k)
    {
        int r_l;
        Monom v = Rewritten(u, Nums[r_k], out r_l);
        return r_l != r_k;
    }
    private static List<PolyQWithSign> Spol(List<CritPairQ> P)
    {
        List<PolyQWithSign> F = new List<PolyQWithSign>();
        for (int i = 0; i < P.Count; i++)
        {
            if (!IsRewritten(P[i].U_1, Pols[P[i].R_1]) && !IsRewritten(P[i].U_2,
Pols[P[i].R_2]))
            {
                PolynomQ pol1 = new PolynomQ(P[i].R_1.Poly.sortWay);
                pol1.Add(P[i].U_1, new FieldQ(1));

                PolynomQ pol2 = new PolynomQ(P[i].R_2.Poly.sortWay);
                pol2.Add(P[i].U_2, new FieldQ(-1));
```

```
                K++;
                PolyQWithSign r_N = new PolyQWithSign(P[i].R_1.Sign.Prod(P[i].U_1),
P[i].R_1.Poly * pol1 + (P[i].R_2.Poly * pol2));
                AddRule(K, r_N);
                F.Add(r_N);
            }
        }
        F.Sort(new ComparePolyQSign());
        return F;
    }

    private static PolyQWithSign IsReducible(PolyQWithSign r_i0, List<PolyQWithSign>
G, int k, List<PolynomQ> B_last)
    {
        for (int i = 0; i < G.Count; i++)
        {
            bool a = Monom.IsDivisible(r_i0.Poly.LM(), G[i].Poly.LM());
            if (a)
            {
                Monom u = Monom.Div(r_i0.Poly.LM(), G[i].Poly.LM());
                PolynomQ pol = new PolynomQ(r_i0.Poly.sortWay);
                pol.Add(Monom.Prod(u, G[i].Sign.Coefficient), new FieldQ(1));
                bool b = Grebner.Mod(pol, B_last.ToArray()).Equals(pol);
                if (b)
                {
                    bool c = (!IsRewritten(u, Pols[G[i]]));
                    if (c)
                    {
                        Signature s = new Signature(Monom.Prod(u,
G[i].Sign.Coefficient), G[i].Sign.PolynomIndex);
                        bool d = s.Equals(r_i0.Sign);
                        if (d) return G[i];
                    }
                }
            }
        }
        return null;
    }
    private static PolyQWithSign TopReduction(PolyQWithSign r_k0, List<PolyQWithSign>
G, int k, List<PolynomQ> B_last, out List<PolyQWithSign> ToDo_1)
    {
        PolyQWithSign r = IsReducible(r_k0, G, k, B_last);
        if (r == null)
        {
            ToDo_1 = new List<PolyQWithSign>();
            return new PolyQWithSign(r_k0.Sign, r_k0.Poly.Simplify());
        }
        else
        {
            Monom u = Monom.Div(r_k0.Poly.LM(), r.Poly.LM());
            if (r.Sign.Prod(u).IsBetterThan(r.Poly.sortWay, r_k0.Sign))
            {
                PolynomQ pol = new PolynomQ(r_k0.Poly.sortWay);
                pol.Add(u, new FieldQ(-1));
                r_k0.Poly = r_k0.Poly + (r.Poly * pol);
                ToDo_1 = new List<PolyQWithSign>();
                ToDo_1.Add(r_k0);
                return null;
            }
            else
            {
                K++;
                PolynomQ pol = new PolynomQ(r_k0.Poly.sortWay);
                pol.Add(u, new FieldQ(1));
```

```
                PolyQWithSign r_N = new PolyQWithSign(r.Sign.Prod(u), r.Poly * pol +
(-r_k0.Poly));
                AddRule(K, r_N);
                ToDo_1 = new List<PolyQWithSign>();
                ToDo_1.Add(r_N);
                ToDo_1.Add(r_k0);
                return null;
            }
        }

    }
    private static List<PolyQWithSign> Reduction(List<PolyQWithSign> ToDo,
List<PolyQWithSign> G_i, int k, List<PolynomQ> B_last)
    {
        List<PolyQWithSign> Done = new List<PolyQWithSign>();
        while (ToDo.Count != 0)
        {
            PolyQWithSign h = ToDo[0];
            ToDo.RemoveAt(0);
            PolyQWithSign phi_h = new PolyQWithSign(h.Sign, Grebner.Mod(h.Poly,
B_last.ToArray()));
            List<PolyQWithSign> ToDo_1 = new List<PolyQWithSign>();
            PolyQWithSign h_1 = TopReduction(phi_h, G_i.Union(Done).ToList(), k,
B_last, out ToDo_1);
            if (h_1 != null) Done.Add(h_1);
            ToDo.AddRange(ToDo_1);
            ToDo.Sort(new ComparePolyQSign());
        }
        return Done;
    }
    private static List<PolyQWithSign> F5(int i, PolynomQ f_i, List<PolyQWithSign>
G_last, List<PolynomQ> B_last)
    {
        List<PolyQWithSign> G_i = new List<PolyQWithSign>();
        int N = G_last[0].Poly.VarCnt;
        int Cnt = K + 1;
        PolyQWithSign r_i = new PolyQWithSign(new Signature(N, i), f_i);
        G_i.AddRange(G_last);
        G_i.Add(r_i);
        Nums.Add(r_i.Sign.PolynomIndex, r_i);
        Pols.Add(r_i, r_i.Sign.PolynomIndex);
        List<CritPairQ> P = new List<CritPairQ>();
        foreach (PolyQWithSign r in G_last)
        {
            CritPairQ cp = CritPairQ.Get(r_i, r, i, B_last);
            if (cp != null) P.Add(cp);
        }
        CompareCritPairQ Comp = new CompareCritPairQ();
        P.Sort(Comp);
        while (P.Count != 0)
        {
            int d = P[0].LCM.FullDeg();
            List<CritPairQ> P_d = P.FindAll(p => p.LCM.FullDeg() == d);
            P_d.Sort(Comp);
            P.RemoveAll(p => p.LCM.FullDeg() == d);
            List<PolyQWithSign> F = Spol(P_d);
            List<PolyQWithSign> R_d = Reduction(F, G_i, i, B_last);
            foreach (var r in R_d)
            {
                foreach (var p in G_i)
                {
                    CritPairQ cp = CritPairQ.Get(r, p, i, B_last);
                    if (cp != null) P.Add(cp);
                }
```

```
                G_i.Add(r);
                Nums.Add(Cnt, r);
                Pols.Add(r, Cnt);
                Cnt++;
            }
            P.Sort(Comp);
        }
        return G_i;
    }
    public static List<PolynomQ> MakeGrebnerBase(PolynomQ[] F)
    {
        int M = F.Length; ResetRules(M);
        int N = F[0].VarCnt; K = M - 1;
        List<PolyQWithSign>[] G = new List<PolyQWithSign>[M];
        G[M - 1] = new List<PolyQWithSign>();
        G[M - 1].Add(new PolyQWithSign(new Signature(N, M - 1), F[M - 1]));
        Nums.Add(K, G[M - 1][0]); Pols.Add(G[M - 1][0], K);
        List<PolynomQ> B_last = G[M - 1].Select(x => x.Poly).ToList();
        for (int i = M - 2; i >= 0; i--)
        {
            G[i] = F5(i, F[i], G[i + 1], B_last);
            B_last = new List<PolynomQ>();
            for (int j = 0; j < G[i].Count; j++)
            {
                B_last.Add(new PolynomQ(G[i][j].Poly));
            }
            Grebner.ReduceBasis(B_last);
        }
        return B_last;
    }
}
```

The module Faugere/F5C

There is versions of F5C's implementations for the fields Q and F_{2^n}

```
public static class FaugereF5CF_2N
{
    private static List<RuleF5>[] Rules;
    private static Dictionary<int, PolyF_2NWithSign> Nums;
    private static Dictionary<PolyF_2NWithSign, int> Pols;
    static int K;

    private static void ResetRules(int M)
    {
        Rules = new List<RuleF5>[M];
        Nums = new Dictionary<int, PolyF_2NWithSign>();
        Pols = new Dictionary<PolyF_2NWithSign, int>();
        for (int i = 0; i < M; i++)
        {
            Rules[i] = new List<RuleF5>();
        }
    }

    private static void AddRule(int K, PolyF_2NWithSign r_N)
    {
        Rules[r_N.Sign.PolynomIndex].Insert(0, new RuleF5(r_N.Sign.Coefficient, K));
    }

    private static Monom Rewritten(Monom u, PolyF_2NWithSign r_k, out int r_o)
    {
```

```
        List<RuleF5> L = Rules[r_k.Sign.PolynomIndex];
        for (int i = 0; i < L.Count; i++)
        {
            Monom ut = Monom.Prod(u, r_k.Sign.Coefficient);
            if (Monom.IsDivisible(ut, L[i].Monom))
            {
                Monom v = Monom.Div(ut, L[i].Monom);
                r_o = L[i].Index;
                return Monom.Div(ut, L[i].Monom);
            }
        }
        r_o = Pols[r_k];
        return u;
    }
    private static bool IsRewritten(Monom u, int r_k)
    {
        int r_l;
        Monom v = Rewritten(u, Nums[r_k], out r_l);
        return r_l != r_k;
    }
    private static List<PolyF_2NWithSign> Spol(List<CritPairF_2N> P)
    {
        List<PolyF_2NWithSign> F = new List<PolyF_2NWithSign>();
        for (int i = 0; i < P.Count; i++)
        {
            if (!IsRewritten(P[i].U_1, Pols[P[i].R_1]) && !IsRewritten(P[i].U_2,
Pols[P[i].R_2]))
            {
                var field = P[i].R_1.Poly.LC().Prim;

                PolynomF_2N pol1 = new PolynomF_2N(P[i].R_1.Poly.sortWay);
                pol1.Add(P[i].U_1, new FieldF_2N(field, FieldF_2N.Convert(1)));

                PolynomF_2N pol2 = new PolynomF_2N(P[i].R_2.Poly.sortWay);
                pol2.Add(P[i].U_2, new FieldF_2N(field, FieldF_2N.Convert(1)));

                K++;
                PolyF_2NWithSign r_N = new
PolyF_2NWithSign(P[i].R_1.Sign.Prod(P[i].U_1), P[i].R_1.Poly * pol1 + (P[i].R_2.Poly *
pol2));
                AddRule(K, r_N);
                F.Add(r_N);
            }
        }
        F.Sort(new ComparePolyF_2NSign());
        return F;
    }

    private static PolyF_2NWithSign IsReducible(PolyF_2NWithSign r_i0,
List<PolyF_2NWithSign> G, List<PolynomF_2N> B_last)
    {
        for (int i = 0; i < G.Count; i++)
        {
            bool a = Monom.IsDivisible(r_i0.Poly.LM(), G[i].Poly.LM());
            if (a)
            {
                Monom u = Monom.Div(r_i0.Poly.LM(), G[i].Poly.LM());
                PolynomF_2N pol = new PolynomF_2N(r_i0.Poly.sortWay);
                var field = r_i0.Poly.LC().Prim;
                pol.Add(Monom.Prod(u, G[i].Sign.Coefficient), new FieldF_2N(field,
FieldF_2N.Convert(1)));
                bool b = Grebner.Mod(pol, B_last.ToArray()).Equals(pol);
                if (b)
                {
```

```
                    Signature s = new Signature(Monom.Prod(u, G[i].Sign.Coefficient),
G[i].Sign.PolynomIndex);
                    bool c = s.Equals(r_i0.Sign);
                    if (c)
                    {
                        bool d = (!IsRewritten(u, Pols[G[i]]));
                        if (d) return G[i];
                    }
                }
            }
        }
        return null;
    }
    private static PolyF_2NWithSign TopReduction(PolyF_2NWithSign r_k0,
List<PolyF_2NWithSign> G, List<PolynomF_2N> B_last, out List<PolyF_2NWithSign> ToDo_1)
    {
        PolyF_2NWithSign r = IsReducible(r_k0, G, B_last);
        if (r == null)
        {
            ToDo_1 = new List<PolyF_2NWithSign>();
            return new PolyF_2NWithSign(r_k0.Sign, r_k0.Poly.Simplify());
        }
        else
        {
            Monom u = Monom.Div(r_k0.Poly.LM(), r.Poly.LM());
            if (r.Sign.Prod(u).IsBetterThan(r.Poly.sortWay, r_k0.Sign))
            {
                PolynomF_2N pol = new PolynomF_2N(r_k0.Poly.sortWay);
                var field = r_k0.Poly.LC().Prim;
                pol.Add(u, new FieldF_2N(field, FieldF_2N.Convert(1)));
                r_k0.Poly = r_k0.Poly + (r.Poly * pol);
                ToDo_1 = new List<PolyF_2NWithSign>();
                ToDo_1.Add(r_k0);
                return null;
            }
            else
            {
                K++;
                PolynomF_2N pol = new PolynomF_2N(r_k0.Poly.sortWay);
                var field = r_k0.Poly.LC().Prim;
                pol.Add(u, new FieldF_2N(field, FieldF_2N.Convert(1)));
                PolyF_2NWithSign r_N = new PolyF_2NWithSign(r.Sign.Prod(u), r.Poly *
pol + (-r_k0.Poly));
                AddRule(K, r_N);
                ToDo_1 = new List<PolyF_2NWithSign>();
                ToDo_1.Add(r_N);
                ToDo_1.Add(r_k0);
                return null;
            }
        }

    }
    private static List<PolyF_2NWithSign> Reduction(List<PolyF_2NWithSign> ToDo,
List<PolyF_2NWithSign> G_i, List<PolynomF_2N> B_last)
    {
        List<PolyF_2NWithSign> Done = new List<PolyF_2NWithSign>();
        while (ToDo.Count != 0)
        {
            PolyF_2NWithSign h = ToDo[0];
            ToDo.RemoveAt(0);
            PolyF_2NWithSign phi_h = new PolyF_2NWithSign(h.Sign, Grebner.Mod(h.Poly,
B_last.ToArray()));
            List<PolyF_2NWithSign> ToDo_1 = new List<PolyF_2NWithSign>();
            if (phi_h.Poly.list.Count == 0) continue;
```

```
            PolyF_2NWithSign h_1 = TopReduction(phi_h, G_i.Union(Done).ToList(),
B_last, out ToDo_1);
            if (h_1 != null) Done.Add(h_1);
            ToDo.AddRange(ToDo_1);
            ToDo.Sort(new ComparePolyF_2NSign());
        }
        return Done;
    }
    private static List<PolyF_2NWithSign> F5(int i, PolynomF_2N f_i,
List<PolyF_2NWithSign> G_last, List<PolynomF_2N> B_last)
    {
        List<PolyF_2NWithSign> G_i = new List<PolyF_2NWithSign>();
        int N = G_last[0].Poly.VarCnt;
        int Cnt = K + 1;
        PolyF_2NWithSign r_i = new PolyF_2NWithSign(new Signature(N, Cnt), f_i);
        G_i.AddRange(G_last);
        G_i.Add(r_i);
        Nums.Add(r_i.Sign.PolynomIndex, r_i);
        Pols.Add(r_i, r_i.Sign.PolynomIndex);
        List<CritPairF_2N> P = new List<CritPairF_2N>();
        foreach (PolyF_2NWithSign r in G_last)
        {
            CritPairF_2N cp = CritPairF_2N.Get(r_i, r, i, B_last);
            if (cp != null) P.Add(cp);
        }
        CompareCritPairF_2N Comp = new CompareCritPairF_2N();
        P.Sort(Comp);
        while (P.Count != 0)
        {
            int d = P[0].LCM.FullDeg();
            List<CritPairF_2N> P_d = P.FindAll(p => p.LCM.FullDeg() == d);
            P_d.Sort(Comp);
            P.RemoveAll(p => p.LCM.FullDeg() == d);
            List<PolyF_2NWithSign> F = Spol(P_d);
            List<PolyF_2NWithSign> R_d = Reduction(F, G_i, B_last);
            foreach (var r in R_d)
            {
                foreach (var p in G_i)
                {
                    CritPairF_2N cp = CritPairF_2N.Get(r, p, i, B_last);
                    if (cp != null) P.Add(cp);
                }
                G_i.Add(r);
                Cnt++;
                Nums.Add(Cnt, r);
                Pols.Add(r, Cnt);
            }
            P.Sort(Comp);
        }
        return G_i;
    }
    public static List<PolynomF_2N> MakeGrebnerBase(PolynomF_2N[] F)
    {
        int M = F.Length; ResetRules(M);
        int N = F[0].VarCnt; K = M - 1;
        List<PolyF_2NWithSign> G_old = new List<PolyF_2NWithSign>();
        G_old.Add(new PolyF_2NWithSign(new Signature(N, M - 1), F[M - 1]));
        Nums.Add(K, G_old[0]); Pols.Add(G_old[0], K);
        List<PolynomF_2N> B_last = G_old.Select(x => x.Poly).ToList();

        for (int i = M - 2; i >= 0; i--)
        {
            List<PolyF_2NWithSign> G_new = F5(K - 1, F[i], G_old, B_last);
            B_last = G_new.Select(x => x.Poly).ToList();
```

```
            Grebner.ReduceBasis(B_last);
            G_old = new List<PolyF_2NWithSign>();
            K = B_last.Count - 1;
            ResetRules(B_last.Count + 1);
            for (int j = K; j >= 0; j--)
            {
                PolyF_2NWithSign pol = new PolyF_2NWithSign(new Signature(N, K - j),
B_last[j]);
                G_old.Add(pol);
                Nums.Add(j, pol); Pols.Add(pol, j);
            }
        }
        return B_last;
    }
}

public static class FaugereF5CQ
{
    private static List<RuleF5>[] Rules;
    private static Dictionary<int, PolyQWithSign> Nums;
    private static Dictionary<PolyQWithSign, int> Pols;
    static int K;

    private static void ResetRules(int M)
    {
        Rules = new List<RuleF5>[M];
        Nums = new Dictionary<int, PolyQWithSign>();
        Pols = new Dictionary<PolyQWithSign, int>();
        for (int i = 0; i < M; i++)
        {
            Rules[i] = new List<RuleF5>();
        }
    }

    private static void AddRule(int K, PolyQWithSign r_N)
    {
        Rules[r_N.Sign.PolynomIndex].Insert(0, new RuleF5(r_N.Sign.Coefficient, K));
    }

    private static Monom Rewritten(Monom u, PolyQWithSign r_k, out int r_o)
    {
        List<RuleF5> L = Rules[r_k.Sign.PolynomIndex];
        for (int i = 0; i < L.Count; i++)
        {
            Monom ut = Monom.Prod(u, r_k.Sign.Coefficient);
            if (Monom.IsDivisible(ut, L[i].Monom))
            {
                Monom v = Monom.Div(ut, L[i].Monom);
                r_o = L[i].Index;
                return Monom.Div(ut, L[i].Monom);
            }
        }
        r_o = Pols[r_k];
        return u;
    }
    private static bool IsRewritten(Monom u, int r_k)
    {
        int r_l;
        Monom v = Rewritten(u, Nums[r_k], out r_l);
        return r_l != r_k;
    }
    private static List<PolyQWithSign> Spol(List<CritPairQ> P)
    {
        List<PolyQWithSign> F = new List<PolyQWithSign>();
```

```
        for (int i = 0; i < P.Count; i++)
        {
            if (!IsRewritten(P[i].U_1, Pols[P[i].R_1]) && !IsRewritten(P[i].U_2,
Pols[P[i].R_2]))
            {
                PolynomQ pol1 = new PolynomQ(P[i].R_1.Poly.sortWay);
                pol1.Add(P[i].U_1, new FieldQ(1));

                PolynomQ pol2 = new PolynomQ(P[i].R_2.Poly.sortWay);
                pol2.Add(P[i].U_2, new FieldQ(-1));

                K++;
                PolyQWithSign r_N = new PolyQWithSign(P[i].R_1.Sign.Prod(P[i].U_1),
P[i].R_1.Poly * pol1 + (P[i].R_2.Poly * pol2));
                AddRule(K, r_N);
                F.Add(r_N);
            }
        }
        F.Sort(new ComparePolyQSign());
        return F;
    }

    private static PolyQWithSign IsReducible(PolyQWithSign r_i0, List<PolyQWithSign>
G, List<PolynomQ> B_last)
    {
        for (int i = 0; i < G.Count; i++)
        {
            bool a = Monom.IsDivisible(r_i0.Poly.LM(), G[i].Poly.LM());
            if (a)
            {
                Monom u = Monom.Div(r_i0.Poly.LM(), G[i].Poly.LM());
                PolynomQ pol = new PolynomQ(r_i0.Poly.sortWay);
                pol.Add(Monom.Prod(u, G[i].Sign.Coefficient), new FieldQ(1));
                bool b = Grebner.Mod(pol, B_last.ToArray()).Equals(pol);
                if (b)
                {
                    bool c = (!IsRewritten(u, Pols[G[i]]));
                    if (c)
                    {
                        Signature s = new Signature(Monom.Prod(u,
G[i].Sign.Coefficient), G[i].Sign.PolynomIndex);
                        bool d = s.Equals(r_i0.Sign);
                        if (d) return G[i];
                    }
                }
            }
        }
        return null;
    }
    private static PolyQWithSign TopReduction(PolyQWithSign r_k0, List<PolyQWithSign>
G, List<PolynomQ> B_last, out List<PolyQWithSign> ToDo_1)
    {
        PolyQWithSign r = IsReducible(r_k0, G, B_last);
        if (r == null)
        {
            ToDo_1 = new List<PolyQWithSign>();
            return new PolyQWithSign(r_k0.Sign, r_k0.Poly.Simplify());
        }
        else
        {
            Monom u = Monom.Div(r_k0.Poly.LM(), r.Poly.LM());
            if (r.Sign.Prod(u).IsBetterThan(r.Poly.sortWay, r_k0.Sign))
            {
                PolynomQ pol = new PolynomQ(r_k0.Poly.sortWay);
```

```
                pol.Add(u, new FieldQ(-1));
                r_k0.Poly = r_k0.Poly + (r.Poly * pol);
                ToDo_1 = new List<PolyQWithSign>();
                ToDo_1.Add(r_k0);
                return null;
            }
            else
            {
                K++;
                PolynomQ pol = new PolynomQ(r_k0.Poly.sortWay);
                pol.Add(u, new FieldQ(1));
                PolyQWithSign r_N = new PolyQWithSign(r.Sign.Prod(u), r.Poly * pol +
(-r_k0.Poly));
                AddRule(K, r_N);
                ToDo_1 = new List<PolyQWithSign>();
                ToDo_1.Add(r_N);
                ToDo_1.Add(r_k0);
                return null;
            }
        }

    }
    private static List<PolyQWithSign> Reduction(List<PolyQWithSign> ToDo,
List<PolyQWithSign> G_i, List<PolynomQ> B_last)
    {
        List<PolyQWithSign> Done = new List<PolyQWithSign>();
        while (ToDo.Count != 0)
        {
            PolyQWithSign h = ToDo[0];
            ToDo.RemoveAt(0);
            PolyQWithSign phi_h = new PolyQWithSign(h.Sign, Grebner.Mod(h.Poly,
B_last.ToArray()));
            List<PolyQWithSign> ToDo_1 = new List<PolyQWithSign>();
            PolyQWithSign h_1 = TopReduction(phi_h, G_i.Union(Done).ToList(), B_last,
out ToDo_1);
            if (h_1 != null) Done.Add(h_1);
            ToDo.AddRange(ToDo_1);
            ToDo.Sort(new ComparePolyQSign());
        }
        return Done;
    }
    private static List<PolyQWithSign> F5(int i, PolynomQ f_i, List<PolyQWithSign>
G_last, List<PolynomQ> B_last)
    {
        List<PolyQWithSign> G_i = new List<PolyQWithSign>();
        int N = G_last[0].Poly.VarCnt;
        int Cnt = K + 1;
        PolyQWithSign r_i = new PolyQWithSign(new Signature(N, i), f_i);
        G_i.AddRange(G_last);
        G_i.Add(r_i);
        Nums.Add(r_i.Sign.PolynomIndex, r_i);
        Pols.Add(r_i, r_i.Sign.PolynomIndex);
        List<CritPairQ> P = new List<CritPairQ>();
        foreach (PolyQWithSign r in G_last)
        {
            CritPairQ cp = CritPairQ.Get(r_i, r, i, B_last);
            if (cp != null) P.Add(cp);
        }
        CompareCritPairQ Comp = new CompareCritPairQ();
        P.Sort(Comp);
        while (P.Count != 0)
        {
            int d = P[0].LCM.FullDeg();
            List<CritPairQ> P_d = P.FindAll(p => p.LCM.FullDeg() == d);
```

```
            P_d.Sort(Comp);
            P.RemoveAll(p => p.LCM.FullDeg() == d);
            List<PolyQWithSign> F = Spol(P_d);
            List<PolyQWithSign> R_d = Reduction(F, G_i, B_last);
            foreach (var r in R_d)
            {
                foreach (var p in G_i)
                {
                    CritPairQ cp = CritPairQ.Get(r, p, i, B_last);
                    if (cp != null) P.Add(cp);
                }
                G_i.Add(r);
                Nums.Add(Cnt, r);
                Pols.Add(r, Cnt);
                Cnt++;
            }
            P.Sort(Comp);
        }
        return G_i;
    }
    public static List<PolynomQ> MakeGrebnerBase(PolynomQ[] F)
    {
        int M = F.Length; ResetRules(M);
        int N = F[0].VarCnt; K = M - 1;
        List<PolyQWithSign> G_old = new List<PolyQWithSign>();
        G_old.Add(new PolyQWithSign(new Signature(N, M - 1), F[M - 1]));
        Nums.Add(K, G_old[0]); Pols.Add(G_old[0], K);
        List<PolynomQ> B_last = G_old.Select(x => x.Poly).ToList();

        for (int i = M - 2; i >= 0; i--)
        {
            List<PolyQWithSign> G_new = F5(K - 1, F[i], G_old, B_last);
            B_last = G_new.Select(x => x.Poly).ToList();
            Grebner.ReduceBasis(B_last);
            G_old = new List<PolyQWithSign>();
            K = B_last.Count - 1;
            ResetRules(B_last.Count + 1);
            for (int j = K; j >= 0; j--)
            {
                PolyQWithSign pol = new PolyQWithSign(new Signature(N, K - j),
B_last[j]);
                G_old.Add(pol);
                Nums.Add(j, pol); Pols.Add(pol, j);
            }
        }
        return B_last;
    }
}
```