Schedae Informaticae Vol. 25 (2016): 209–225
doi: 10.4467/20838476SI.16.016.6197

Complexity of the Inversion Algorithm of Polynomial Mappings

Paweł Bogdan
Chair of Effective Methods in Algebra,
Department of Computer Science and Computer Mathematics
Faculty of Mathematics and Computer Science
ul. Łojasiewicza 6, 30-348 Kraków, Poland
e-mail: *pawel.bogdan@doctoral.uj.edu.pl*

Abstract. In this paper we will recall the inversion algorithm described in [1]. The algorithm classifies polynomial automorphisms into two sets: *Pascal finite* and *Pascal infinite*. In this article the complexity of the inversion algorithm will be estimated. To do so, we will present two popular ways how Computer Algebra Systems (CASes) keep the information about multivariate polynomials. We will define the complexity as the amount of simple operations performed by the algorithm as a function of the size of the input. We will define simple operations of the algorithm. Then we will estimate complexity of checking that the polynomial map is not a polynomial automorphism. To do so we will use theorem 3.1 from [1].

Keywords: polynomial automorphism, differential Galois theory, computational complexity, computer algebra system

1. Introduction

Checking if a polynomial map is a polynomial automorphism is a very interesting problem investigated by many mathematicians. This problem is strongly connected to the Jacobian Conjecture. That problem was established by Ott-Heinrich Keller in his article *Ganze Cremona-Trasformationen* published in 1939. The problem stated by Keller is in fact a question:

Received: 25 November 2016 / Accepted: 20 December 2016

Question 1 (The Keller's problem) *Let $F_1, ..., F_n$ be polynomials such that $F_i \in \mathbb{Z}[X_1, ..., X_n]$ for each $i \in \{1, ..., n\}$ and the determinant $\det\left(\frac{\partial F_i}{\partial X_j}\right)_{1 \leq i,j \leq n}$ is equal to 1. Is it true that, X_i can be expressed as a polynomial with integer coefficients in the variables $F_1, ..., F_n$?*

Nowadays the Keller's problem is known as the Jacobian Conjecture:

Conjecture 2 (The Jacobian Conjecture) *Let $F = (F_1, ..., F_n) : K^n \to K^n$ be a polynomial map, where K is an algebraically closed field of characteristic 0 and n is an integer. The determinant of the Jacobian Matrix of the polynomial mapping F is a non-zero constant if and only if F is a polynomial automorphism.*

This hypothesis has been studied by many mathematicians who used a lot of different mathematical theories. One of them was L. Andrew Campbell, who in [2] stated the following theorem:

Theorem 3 *Let $F : \mathbb{C}^n \to \mathbb{C}^n$ be a polynomial map. The mapping F has a polynomial inverse if and only if the determinant of the Jacobian Matrix of F never vanishes and it induces a normal function field extension.*

The next significant result in the research of the Jacobian Conjecture is an article of Teresa Crespo and Zbigniew Hajto [3]. They generalized the approach of Campbell and described the hypothesis using the language of the differential Galois theory in the following theorem:

Theorem 4 *Let $\mathbb{K}$ be an algebraically closed field of a characteristic zero and let n be an integer. Let $F = (F_1, ..., F_n) : \mathbb{K}^n \to \mathbb{K}^n$ be a polynomial mapping such that the determinant of its Jacobian Matrix is a non-zero constant. Let $\delta_1, ..., \delta_n$ be the Nambu derivations which are defined by the formula:*

$$\begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix} = \left(J^{-1}\right)^T \begin{pmatrix} \frac{\partial}{\partial X_1} \\ \vdots \\ \frac{\partial}{\partial X_n} \end{pmatrix},$$

where J is the Jacobian Matrix of F. The following conditions are equivalent:

1. *The function F is invertible and its inverse is a polynomial function.*
2. *The differential field $(\mathbb{K}(X_1, ..., X_n), \{\delta_1, ..., \delta_n\})$ is a Picard-Vessiot extension of $\mathbb{K}$.*
3. *The finite field extension $\mathbb{K}(F_1, ..., F_n)$ of $\mathbb{K}(X_1, ..., X_n)$ is a Galois extension.*
4. *The elements $X_1, ..., X_n$ satisfy $W_{\delta_1,...,\delta_n}(X_1, ..., X_n) \neq 0$ and*

$$\frac{W_{\theta_1,...,\theta_n}(X_1, ..., X_n)}{W_{\delta_1,...,\delta_n}(X_1, ..., X_n)} \in \mathbb{K}(F_1, ..., F_n)$$

for n-tuples $\theta_1, ..., \theta_n$ of order not greater than $n + 1$ of the semi-group of differential operators generated by $\delta_1, ..., \delta_n$.

The fourth point of the theorem above gives a criterion to check if a polynomial mapping F is a polynomial automorphism. This criterion is called the **Wronskian**

Criterion. In [4] we reduced the number of the Wronskians which should be considered to be sure that a given polynomial mapping is a polynomial automorphism. During researching the Wronskian Criterion we used the Groebner Basis. As an alternative we invented the inversion algorithm. In [1] we described this algorithm, we discussed its properties and proved its correctness. The very natural question which appears after describing an algorithm and proving its correctness is the question about its computational complexity. This article answers this question.

The computational complexity estimation of calculating mathematical functions is discussed very often. For example Müller [5] estimated the complexity of computation of Taylor series. There are many articles about the computational complexity of algorithms computing the Groebner Basis. For example: [6] or [7].

The set of polynomial automorphisms which are inverted are called **Pascal finite**. The set of *Pascal finite* polynomial automorphisms has very interesting properties, which are discussed in [8]. Hence, our algorithm defines the new classification of the polynomial automorphisms. This classification can be useful to study the Jacobian Conjecture.

In the section 2 there is a description of our inversion algorithm and example of implementation of this algorithm. In the section 3, one can find the most common methods to store multivariate polynomials in computer programmes. The fourth section contains the definition of the computational complexity and the estimation of the complexity of the inversion algorithm. In the section 5, we recall Theorem 3.1 from [1] which let us estimate the number of steps that should be performed by the algorithm in order to obtain an inverse mapping. In the section 6 one can find the conclusion, which compares our algorithm with other ones.

2. The inversion algorithm

In [1] we presented the following algorithm, constructed in order to obtain the inverse of a given polynomial mapping F.

INPUT :

- Polynomial map $F \ : \ K^n \to K^n$
- Polynomial $P \in K[X] = K[X_1, ..., X_n]$

OUTPUT :

- Polynomial $G \in K[F] = K[F_1, ..., F_n]$ such that: $G(F(X)) - P(X) = 0$

This algorithm finds the polynomial $R \in K[F]$ for a given polynomial $P \in K[X]$ whenever it exists. If we put one of the variables $X_1, ..., X_n$ as polynomial P, we obtain the inverse if F, if it exist.

STEP 0 : $P_0(X_1, ..., X_n) = P(X_1, ..., X_n)$

STEP 2 : $P_1(X_1, ..., X_n) = P_0(F_1, ..., F_n) - P_0(X_1, ..., X_n)$

...

STEP k : $P_k(X_1, ..., X_n) = P_{k-1}(F_1, ..., F_n) - P_{k-1}(X_1, ..., X_n)$

STOP : if $P_k(X_1, ..., X_n) = 0$ for some $k \in \mathbb{N}_+$ then algorithm stops and polynomial R can be obtained. The corollary (formulated and proved in [1]) below describes how the polynomial R can be found.

Corollary 5 *Let F be a polynomial map. If $P_k = 0$, then F is invertible and the inverse map R if F is given by:*

$$G(X) = \sum_{l=0}^{k-1} (-1)^l P_l(X)$$

An implementation of this algorithm in Sage[1] can be found below:

```
def inversion_algorithm(R1, R2, F, P):
    '''
    :param R1: Ring K[X_1,...,X_n]
    :param R2: Ring K[F_1,...,F_n]
    :param F: Polynomial mapping F = (F_1,...,F_n)
    :param P: Polynomial in ring K[X_1,...,X_n] to inverse
    :return: Polynomial G in ring K[F_1,...,F_n] such that G(F)-P(X)=0
    '''
    Fs = R2.gens()
    G = 0
    while P != 0:
        G = P(Fs) - G
        P = P(F) - P
    return G
```

This version of the algorithm is very easy to implement, but also inconvenient to estimate time complexity. Let us introduce new notation. For the convenience of the reader we follow the notation established in [1].

$$I = (X_1, ..., X_n) \tag{1}$$

P will denote the polynomial mapping: $P : K^n \to K^n$. We define a polynomial ring endomorphism σ:

$$\sigma_F(P) = P(F_1, ..., F_n) = P \circ F \in K[X_1, ..., X_n] \tag{2}$$

[1] To see more information about Sage see [9].

Using the notation given above we can write steps of the algorithm, assuming that P is a polynomial map instead of a polynomial.

$$\begin{aligned}
P_0 &= I \\
P_1 &= \sigma_F(P_0) - P_0 \\
P_2 &= \sigma_F(P_1) - P_1 = \sigma_F\left(\sigma_F(P_0) - P_0\right) - \left(\sigma_F(P_0) - P_0\right) \\
&= \sigma_F\left(\sigma_F(P_0)\right) - \sigma_F(P_0) - \sigma_F(P_0) + P_0 \\
&= \sigma_F^2(P_0) - 2\sigma(P_0) + P_0 \\
P_3 &= \sigma_F(P_2) - P_2 = \sigma_F\left(\sigma_F^2(P_0) - 2\sigma(P_0) + P_0\right) - \left(\sigma_F^2(P_0) - 2\sigma(P_0) + P_0\right) \\
&= \sigma_F^3(P_0) - 2\sigma_F^2(P_0) + \sigma(P_0) - \sigma_F^2(P_0) + 2\sigma(P_0) - P_0 \\
&= \sigma_F^3(P_0) - 3\sigma_F^2(P_0) + 3\sigma(P_0) - P_0 \\
&... \\
P_k &= \sum_{i=0}^{k} \binom{k}{i} (-1)^i \sigma_F^{k-i}(P_0)
\end{aligned}$$

We have an exact formula for the k-th polynomial. To compute the k-th polynomial we need to:

1. compute all Newton symbols $\binom{k}{i}$ for $i = 0, ..., k$
2. compute all compositions $\sigma_F(P_0), \sigma_F^2(P_0), ..., \sigma_F^k(P_0)$

Remark: Coefficients in formula for kth polynomial are elements in kth row of Pascal Triangle. Hence, the set of polynomial automorphisms, such that are inverted by the inversion algorithm, are called *Pascal Finite*.

3. How computer remembers multivariate polynomials?

If we want to estimate time complexity of our algorithm, we need to know how a polynomial is stored in memory and what exactly is done during performing algorithm.

We have decided to use Sage software [9] to perform calculations. It is an open source software, so we can exactly control what is happening when our algorithm is performed.
Sage can keep multivariate polynomials in two different ways:

tree – a tree of symbolic expressions. The expression is divided into the simplest operations and the valuation tree is constructed. Each F_i is represented by its own tree. Hence, operation of σ is in fact very simple. Each node that keeps variable X_1 is replaced by the tree which represents the polynomial P_i for each $i = 1, ..., n$. However, answering if the polynomial is constantly equal to 0 is not so easy. To do so, a transformation of the tree has to be done.

dictionary – a dictionary is a very popular data structure, widely used by programmers. It keeps information as a set of pairs. The first item of the pair is called key, the second one is called value. To keep the multivariate polynomial in a dictionary, number and names of variables have to be known before defining the first polynomial. Each monomial of such a multivariate polynomial is kept as a separate pair of key and value. The key is a n-tuple of powers of variables in the monomial, value is coefficient of this monomial. The polynomial is kept in form which allows us to check if polynomial is equal to 0 very quickly, but operation σ is a little more complicated. The usage of memory is noticeably less when we choose to keep a polynomial mapping as a dictionary.

3.1. Example

Let us consider an example of a polynomial mapping: $F\colon \mathbb{C}^2 \to \mathbb{C}^2$

$$F = (F_1, F_2),$$

where

$$\begin{aligned} F_1(x,y) &= x + a_1x^3 + a_2x^2y + a_3xy^2 + a_4y^3, \\ F_2(x,y) &= y + b_1x^3 + b_2x^2y + b_3xy^2 + b_4y^3. \end{aligned}$$

3.1.1. Tree of symbolic expression

The mapping F can be stored by tree of expression:

```
sage: a_1, a_2, a_3, a_4 = var('a_1 a_2 a_3 a_4')
sage: b_1, b_2, b_3, b_4 = var('b_1 b_2 b_3 b_4')
sage: x, y = var('x y')
sage: F_1 = x + a_1*x**3 + a_2*x**2*y + a_3*x*y**2 + a_4*y**3; F_1
a_1*x^3 + a_2*x^2*y + a_3*x*y^2 + a_4*y^3 + x
sage: F_2 = y + b_1*x**3 + b_2*x**2*y + b_3*x*y**2 + b_4*y**3; F_2
b_1*x^3 + b_2*x^2*y + b_3*x*y^2 + b_4*y^3 + y
```

There is the built-in function to print an expression in form of the tree: _dbgprinttree(), but its output can be inconvenient to read, because it contains too much unnecessary information, eg. addresses of pointers in the memory. Using the simple recursive function we can do the same in much more readable form:

```
def get_value(root):
    if root.operator() is None:
        return root
    else:
```

```
        a,b = var('a b')
        nops = len(root.operands())
        if root.operator() == (a+b).operator():
            return 'addtition,     nops = {0}'.format(nops)
        elif root.operator() == (a*b).operator():
            return 'multiplication,     nops = {0}'.format(nops)
        else:
            return 'power'

def print_tree(root, prefix = '+'):
    print prefix,get_value(root)
    for o in root.operands():
        print_tree(o, '|    ' + prefix)
```

Output for F_1:

```
+ addtition,     nops = 5
|    + multiplication,     nops = 2
|    |    + a_1
|    |    + power
|    |    |    + x
|    |    |    + 3
|    + multiplication,     nops = 3
|    |    + a_2
|    |    + power
|    |    |    + x
|    |    |    + 2
|    |    + y
|    + multiplication,     nops = 3
|    |    + a_3
|    |    + x
|    |    + power
|    |    |    + y
|    |    |    + 2
|    + multiplication,     nops = 2
|    |    + a_4
|    |    + power
|    |    |    + y
|    |    |    + 3
|    + x
```

Output for F_2:

```
+ addtition,     nops = 5
|    + multiplication,     nops = 2
|    |    + b_1
|    |    + power
|    |    |    + x
|    |    |    + 3
```

```
|    + multiplication,     nops = 3
|    |    + b_2
|    |    + power
|    |    |    + x
|    |    |    + 2
|    |    + y
|    + multiplication,     nops = 3
|    |    + b_3
|    |    + x
|    |    + power
|    |    |    + y
|    |    |    + 2
|    + multiplication,     nops = 2
|    |    + b_4
|    |    + power
|    |    |    + y
|    |    |    + 3
|    + y
```

Using Sage we can obtain $\sigma_F(F_1)$:

```
sage: sigma_F_1 = F_1.substitute({x: F_1, y: F_2})
sage: sigma_F_1
(a_1*x^3 + a_2*x^2*y + a_3*x*y^2 + a_4*y^3 + x)^3*a_1 +
(a_1*x^3 + a_2*x^2*y + a_3*x*y^2 + a_4*y^3 + x)^2*
(b_1*x^3 + b_2*x^2*y + b_3*x*y^2 + b_4*y^3 + x)*a_2
+ (a_1*x^3 + a_2*x^2*y + a_3*x*y^2 + a_4*y^3 + x)*
(b_1*x^3 + b_2*x^2*y + b_3*x*y^2 + b_4*y^3 + x)^2*a_3 +
(b_1*x^3 + b_2*x^2*y + b_3*x*y^2 + b_4*y^3 + x)^3*a_4 +
a_1*x^3 + a_2*x^2*y + a_3*x*y^2 + a_4*y^3 + x
```

The tree for $\sigma_F(F_1)$:

```
+ addtition,     nops = 9
|    + multiplication,     nops = 2
|    |    + power
|    |    |    + addtition,     nops = 5
|    |    |    |    + multiplication,     nops = 2
|    |    |    |    |    + a_1
|    |    |    |    |    + power
|    |    |    |    |    |    + x
|    |    |    |    |    |    + 3
|    |    |    |    + multiplication,     nops = 3
|    |    |    |    |    + a_2
|    |    |    |    |    + power
|    |    |    |    |    |    + x
|    |    |    |    |    |    + 2
|    |    |    |    |    + y
|    |    |    |    + multiplication,     nops = 3
```

```
|    |    |    |    |    + a_3
|    |    |    |    |    + x
|    |    |    |    |    + power
|    |    |    |    |    |    + y
|    |    |    |    |    |    + 2
|    |    |    |    + multiplication,     nops = 2
|    |    |    |    |    + a_4
|    |    |    |    |    + power
|    |    |    |    |    |    + y
|    |    |    |    |    |    + 3
|    |    |    |    + x
|    |    |    + 3
|    |    + a_1
......
```

As we said before, it is easy to count σ_F in that way – some nodes are replaced by trees. However, as we can see – the size of the tree raises rapidly – reduction of some similar parts or determining if the expression is equal to zero is complicated. There is the operation expand which transforms the tree to form:

```
+ addtition,     nops = 210
|    + multiplication,     nops = 2
|    |    + power
|    |    |    + a_1
|    |    |    + 4
|    |    + power
|    |    |    + x
|    |    |    + 9
|    + multiplication,     nops = 4
|    |    + power
|    |    |    + a_1
|    |    |    + 2
|    |    + a_2
|    |    + b_1
|    |    + power
|    |    |    + x
|    |    |    + 9
|    + multiplication,     nops = 4
|    |    + a_1
|    |    + a_3
|    |    + power
|    |    |    + b_1
|    |    |    + 2
|    |    + power
|    |    |    + x
|    |    |    + 9
........
```

After the expand operation, we have sum of 210 monomials and we conclude that the tree is much simpler.

3.1.2. Dictionary of powers

The documentation of the function expand says:

```
Note: If you want to compute the expanded form of a polynomial
arithmetic operation quickly and the coefficients of the polynomial
all lie in some ring, e.g., the integers, it is vastly faster to
create a polynomial ring and do the arithmetic there.
```

We will do that indeed.

```
sage: a_1, a_2, a_3, a_4 = var('a_1 a_2 a_3 a_4')
sage: b_1, b_2, b_3, b_4 = var('b_1 b_2 b_3 b_4')
sage: R.<x,y> = PolynomialRing(SR, 2)
sage: F_1 = x + a_1*x**3 + a_2*x**2*y + a_3*x*y**2 + a_4*y**3; F_1
a_1*x^3 + a_2*x^2*y + a_3*x*y^2 + a_4*y^3 + x
sage: F_2 = y + b_1*x**3 + b_2*x**2*y + b_3*x*y**2 + b_4*y**3; F_2
b_1*x^3 + b_2*x^2*y + b_3*x*y^2 + b_4*y^3 + y
```

Polynomials are stored in memory as dictionaries:

```
sage: F_1.dict()
{(0, 3): a_4, (1, 0): 1, (1, 2): a_3, (2, 1): a_2, (3, 0): a_1}
sage: F_2.dict()
{(0, 1): 1, (0, 3): b_4, (1, 2): b_3, (2, 1): b_2, (3, 0): b_1}
```

Counting σ_F is a little bit more complicated when we keep a polynomial as a dictionary, because a multiplying of polynomials needs to be perform, but the multiplication is a simple operation. The best form of a polynomial for estimating the complexity is a sum of monomials. This form (a sum of monomials) is also more convenient for saving memory and execution time.

```
sigma_F_1 = F_1(F_1,F_2)
sage: sigma_F_1.dict()
{(0, 3): 2*a_4,
 (0, 5): a_3*a_4 + 3*a_4*b_4,
 (0, 7): a_2*a_4^2 + 2*a_3*a_4*b_4 + 3*a_4*b_4^2,
 (0, 9): a_1*a_4^3 + a_2*a_4^2*b_4 + a_3*a_4*b_4^2 + a_4*b_4^3,
 (1, 0): 1,
 (1, 2): 2*a_3,
 (1, 4): (a_3 + 2*b_4)*a_3 + 2*a_2*a_4 + 3*a_4*b_3,
......
```

4. Complexity estimation

In this section we will define the complexity, just like Winkler did in [10]. After that we will present two well known forms of polynomial maps. Later we will define the simplest operations performed by the algorithm. Then we will estimate complexity of our algorithm. Question "how long the program works for specified input" is very natural, and estimating computational complexity can answer this question.

4.1. Definitions

Definition 6 (Definition 1.5.1 in [10]) *Let A be an algorithm whose set of inputs is a set X. Let $P = \{X_j\}_j$ be* ***partition*** *of the set of inputs, such that. Then:*

$t_A^-(j) := \min\{t_A(x) \ : \ x \in X_j\}$ *denotes* ***minimum time complexity function***[2] *of A*

$t_A^+(j) := \max\{t_A(x) \ : \ x \in X_j\}$ *denotes* ***maximum time complexity function***[3] *of A*

$t_A^*(j) := \sum_{x \in X_j} t_A(x)/\#X_j$ *denotes* ***average time complexity function***[4] *of A*

where t_A denotes the number of the basic steps performed by computer during execution of the algorithm A.

We will use *maximum time complexity function*, because we would like to know how many basic operations have to be performed in the most pessimistic situation. When we consider the complexity, it is very convenient to use notation 'great O':

Definition 7 (Definition 1.5.2 in [10]) *Let f and g be functions from a given set S to set $\mathbb{R}^+$.*

$$f \in O(g) \quad :\Leftrightarrow \quad \exists_c \forall_{s \in S} \quad f(s) \leq c \cdot g(s)$$

Lemma 8 (Lemma 1.5.1 in [10]) *Let $f, g \ S \to \mathbb{R}^+$. If $f(s) \leq c \cdot g(s)$ for $c \in \mathbb{R}^+$ and for all but a finite number of $s \in S$ then $f \in O(g)$*

Lemma 9 (Lemma 1.5.1 in [10]) *Let $f, g \ S \to \mathbb{R}^+$. If $f \in O(g)$, then $g + f \in O(g)$ and $g \in O(g + f)$*

In [11] van der Essen described a very significant result which was developed independently by Aleksandr Yagzhev [12] and Hyman Bass [13]. Their successfully reduced the Jacobian Conjecture. This reduction is known as the form of Bass because Yagzhev published his article in Russian, so anyone outside of Russia did not hear about this paper. According to their articles it is enough to prove the Jacobian Conjecture for polynomial mappings $F = (F_1, ..., F_n)$, where $F_i = X_i + H_i$ for each

[2] or **minimum computing time function**
[3] or **maximum computing time function**
[4] or **average computing time function**

$i \in \{1, ..., n\}$, where H_i is a homogeneous polynomial of degree three in variables $X_1, ..., X_n$.

Another significant result described by van den Essen was published by Ludwik Drużkowski [14]. He stated and proved, that the hypothesis would be proved in general case if it was proved for the polynomial mappings $F = (F_1, ..., F_n)$, where $F_i = X_i + H_i^3$ for each $i \in \{1, ..., n\}$, where H_i is a linear form in variables $X_1, ..., X_n$. Drużkowski has used notation:

$$F(\bar{X}) = \bar{X} + \left(A\bar{X}\right)^{*3},$$

where A is a matrix of coefficients of linear forms and V^{*3} means rising to the third power each element separately.

In obvious way n determines the size of the input. For the given n the maximum number of monomials in a polynomial map in a Drużkowski form is smaller than the maximum number of monomials in a polynomial map in a Bass form. We choose the Bass form, since we want to count *maximum time complexity function*, we would like to consider as many monomials in a polynomial map as possible. Bass form is more general, so we will focus on that form.

If we have a polynomial map in the Bass form defined by n polynomials in n-variable, each of these polynomials has one monomial of degree one, and at most $\binom{n+2}{3}$ monomials[5] of degree three.

We will use dictionary form during our calculations.

4.2. The basic operation

Bass form has two kinds of monomials:

1. Monomial of degree 1,
2. Monomial of degree 3.

We can perform σ_F separately for each monomial of the polynomial map.

Let us denote the number of monomials in the polynomial map $F = (F_1, ..., F_n)$ by

$$\#F,$$

where each F_i is n-variable polynomial.

If we want to perform operation $\sigma_F(M_1)$ where $M_1 = X_i$ is a monomial of degree one, we just need to create $\#F_i$ entries in the result dictionary. The complexity of putting the pair of key and value to the dictionary is constant – we will say the complexity is $O(1)$.

If we want to perform operation $\sigma_F(M_3)$ where M_3 is monomial of degree three, we need to multiply coefficient of M_3 and three polynomials (let us denote them by F_i, F_j and F_k). So, we need to perform at most $\#F_i \cdot \#F_j \cdot \#F_k$ blocks of operations:

[5] Each monomial defined as multiplication of three-element-combination with repetitions of n-element set of variables

- Check if resulting key exists in resulting dictionary
- If it exists:
  - Add a new value to the old one
- If it does not exist:
  - Create new (key, value) pair in the resulting dictionary.

Each new value is a result of multiplication of 4 coefficients of polynomial

Let us sum up the basic operation used by our algorithm:

- Creating the new entry in the dictionary
- Checking if the dictionary has specified key
- Adding two coefficients of polynomials
- Multiplying at least two coefficients of polynomial

We can assume each operation can be performed in the constant time. We need to know how many of these operations should be performed during counting $\sigma_F(F)$ (depending on size of polynomial map F).

4.3. Complexity of $\sigma_F^k(X)$

Firstly, we are going to estimate the time complexity of the calculation $\sigma_F(F)$ depending on the size of the polynomial map F. As we said before, the operation σ_F for the monomial of degree one $M_1 = X_i$ needs as many insertion operations as many monomials F_i has.

We need to create at most $n \cdot \left(\binom{n+2}{3} + 1\right) \in O(n^4)$ new entries in the result polynomial map which has $n \cdot \binom{n+2}{3} \in O(n^4)$ monomials of degree 3.

Each polynomial F_i has at most $\binom{n+2}{3}$ monomials of degree three and one monomial X_i of degree one. $\sigma_F(X_i)$ takes $\binom{n+2}{3} + 1$ operations of insertions. Let's denote by M_i monomial of degree 3 for $i = 1, ..., \binom{n+2}{3}$. Counting $\sigma_F(M_i)$ takes $\left(\binom{n+2}{3} + 1\right) \cdot \left(\binom{n+2}{3} + 1\right) \cdot \left(\binom{n+2}{3} + 1\right)$ operation of insertion or addition and $3 \cdot \binom{n+2}{3} \cdot \binom{n+2}{3} \cdot \binom{n+2}{3}$ multiplication. So to calculate $\sigma_F(M_i)$ we need to perform $3 \cdot \binom{n+2}{3} \cdot \binom{n+2}{3} \cdot \binom{n+2}{3} + \left(\binom{n+2}{3} + 1\right) \cdot \left(\binom{n+2}{3} + 1\right) \cdot \left(\binom{n+2}{3} + 1\right)$ simple operations. Hence, to calculate $\sigma_F(F_i)$ we perform at most

$$\left(\binom{n+2}{3} + 1\right) + \binom{n+2}{3} \cdot \left[3 \cdot \left[\binom{n+2}{3}\right]^3 + \left[\binom{n+2}{3} + 1\right]^3\right] \in O(n^{12}).$$

As a result we obtain sum of $\binom{n+2}{3} \cdot \left[\binom{n+2}{3}\right]^3$ monomials of degree 9 and some monomials of lower degree.

To sum up, after $\sigma_F^2(X)$ we obtain[6]:

[6] where X should be understood as tuple of n variables of the polynomial map F, or in other words as an identity map

- $O(n^{13})$ monomials of the highest degree,
- The degree of polynomials defining the result polynomial map is equal to 9,
- We need to perform $O(n^{13})$ basic operations.

If we want to use notation O, we can omit monomials of lower degree from previous step. So we can limit ourselves to consider the situation when we have the polynomial map with $O(n^{13})$ monomials of degree 9. Let us denote these monomials by M_i. To calculate $\sigma_F(M_i)$ we need to perform $\left[\binom{n+2}{3}+1\right]^9$ insertions to the dictionary or additions and $9 \cdot \left[\binom{n+2}{3}\right]^9$ multiplications. The complexity of counting $\sigma_F^3(X)$ is estimated by $O(n^{40})$. The polynomial map obtained by $\sigma_F^3(X)$ has $O(n^{40})$ monomials of degree 27.

Let us summarize our estimations:

k	$\deg(\sigma_F^k(X))$	The amount of monomials of highest degree	The complexity
1	3	$O(n^4)$	$O(n^4)$
2	9	$O(n^{13})$	$O(n^{13})$
3	27	$O(n^{40})$	$O(n^{40})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

As we can see, the complexity of the calculating (understood as the amount of basic operations) and the amount of monomials of the highest degree are in the same class of O notation. We can extract recurrent formula for the complexity:

$$\begin{aligned} t_1^+ &= O(n^4), \\ t_k^+ &= t_{k-1}^+ \cdot \left(O(n^3)\right)^{3^{k-1}}. \end{aligned}$$

This recurrence can be formulated as the exact formula (we omit O notation for readability):

$$\begin{aligned} t_1^+ &= n \cdot n^3 = n^4, \\ t_2^+ &= n \cdot n^3 \cdot \left(n^3\right)^3 = n^{13}, \\ t_3^+ &= n \cdot n^3 \cdot \left(n^3\right)^3 \cdot \left(n^3\right)^9 = n \cdot \left(n^3\right)^{1+3+9} = n^{40}, \\ t_3^+ &= n \cdot n^3 \cdot \left(n^3\right)^3 \cdot \left(n^3\right)^9, \cdot \left(n^3\right)^{27} = n \cdot \left(n^3\right)^{1+3+9+27} = n^{121}, \\ &\dots \\ t_k^+ &= n \cdot \left(n^3\right)^{1+3+\ldots+3^k} = n \cdot \left(n^3\right)^{\frac{3^k-1}{2}}. \end{aligned}$$

We have just proved, the complexity of the counting $\sigma_F^k(X)$ is in class $O\left(n \cdot \left(n^3\right)^{\frac{3^k-1}{2}}\right)$.

4.4. Complexity of counting kth step of algorithm

As we presented in section 1, k-th step of our algorithm can be calculated using the formula:

$$P_k = \sum_{i=0}^{k} \binom{k}{i} (-1)^i \sigma_F^{k-i}(X)$$

According to the Pascal's rule which is used to build the Pascal Triangle, we can calculate the Newton symbol $\binom{k}{i}$ for each $i = 0, ..., k$ in time $O(k^2)$. We need to know each $\binom{k}{i}$ for $i = 0, ..., k$. Calculating all Newton symbols takes $O(k^3)$ time. We also needs to calculate every $\sigma_F^i(X)$ for $i = 0, ..., k$. We use O notation, so we can consider the complexity of calculating $\sigma_F^k(X)$ as the complexity of calculating P_k, because calculating the Newtons symbols and calculating the other $\sigma_F^i(X)$ for $i = 0, ..., k-1$ needs much less basic operations.

5. Verifying if map is polynomial automorphism

According to [1] algorithm does not stop for every polynomial automorphism. If algorithm stops for a polynomial map F, the polynomial map F is polynomial automorphism and it is called *Pascal finite*. As examples show not all polynomial automorphisms are *Pascal finite*. Two of those examples can be found in [1]:

$$F(X_1, X_2) = \left(X_1 + \left(X_2 + X_1^3\right)^2, X_2 + X_1^3\right)$$

and also:

$$F(X_1, X_2, X_3, X_4) = \left(X_1 + pX_4, X_2 - pX_3, X_3 + X_4^3, X_4\right),$$

where $p = X_1X_3 + X_2X_4$.

However, using Theorem 3.1 from [1] we can compute an inverse mapping even if the algorithm does not stop.

Theorem 10 *Let F be a polynomial map of the form:*

$$\begin{aligned} F_1(X_1, ..., X_n) &= X_1 + H_1(X_1, ..., X_n), \\ &... \\ F_n(X_1, ..., X_n) &= X_n + H_n(X_1, ..., X_n), \end{aligned}$$

where $H_i(X_1, ..., X_n)$ is a polynomial in $X_1, ..., X_n$ of degree D_i and lower degree d_i, with $d_i \geq 2$ for $i = 1, ..., n$, such that the determinant of the Jacobi matrix of the mapping F is equal to 1. Let $d = \min d_i$ and $D = \max D_i$. The following conditions are equivalent:

1. *F is invertible*

2. *For $i = 1, ..., n$ and $m = \left\lfloor \frac{D^{n-1}-d}{d-1} + 1 \right\rfloor + 1$ we have*

$$\Sigma_{j=0}^{m-1}(-1)^j P_j^i(X) = G_i(X) + R_m^i(X).$$

Where $G_i(X)$ is a polynomial of degree $\leq D^{n-1}$, and $R_m^i(X)$ is a polynomial satisfying $R_m^i(X) = (-1)^{m+1} P_m^i(X)$

Moreover the inverse G of F is given by:

$$G_i(Y_1, ..., Y_n) = \Sigma_{l=0}^{m-1}(-1)^l \widehat{P}_l^i(Y_1, ..., Y_n),$$

where $\widehat{P}_l^i$ is the sum of homogeneous summands of P_l^i of degree ≤ 2 and m is an integer $> \frac{D^{n-1}-d_i}{d-1} + 1$

Using theorem given above we are able to estimate the computational complexity of checking if a given polynomial mapping F is not *Pasal finite* polynomial automrphism. The complexity of these calculations is estimated by

$$O\left(n^{-\frac{1}{2}+\frac{1}{2}\cdot 3^{1+\left\lfloor \frac{1}{6}\cdot 3^n+\frac{1}{2}\right\rfloor}}\right). \tag{3}$$

6. Conclusions

The estimated complexity seems to be very big, but this is the estimation of the maximum time complexity function. In fact, the complexity is usually much smaller and the algorithm can be used in practice. According to [6] the worst-case computational complexity of Groebner basis algorithm is double exponential, but this method can be used for special cases in practice. We did not compare those two methods. In my opinion the inversion algorithm is much simpler then the Buchberger algorithm.

Another way to inverse polynomial mappings is the Taylor Series.The inversion algorithm has a very important advantage over the Taylor series – it does not perform any division, hence it can be used to study polynomial mappings with coefficients in finite fields.

In spite of big complexity of a pessimistic computational complexity, the inversion algorithm is very useful, and can be used for many examples with coefficients in various fields and even algebras with nilpotent elements.

7. References

[1] Adamu E., Bogdan P., Crespo T., Hajto Z., *An effective study of polynomial maps*. Journal of Algebra and Its Applications, 2017, 16(5).

[2] Campbell L.A., *A condition for a polynomial map to be invertible*. Mathematische Annalen, 1973, 205(3), pp. 243–248.

[3] Crespo T., Hajto Z., *Picard-vVessiot theory and the Jacobian problem*. Israel Journal of Mathematics, 2012, 186(1), pp. 401–406.

[4] Adamus E., Bogdan P., Hajto Z., *An effective approach to Picard-Vessiot theory and the Jacobian Conjecture*, 2015, submitted.

[5] Müller N.T., *Uniform Computational Complexity of Taylor Series*. In: *14th International Colloquium on Automata, Languages and Programming*, London, UK, UK, Springer-Verlag, 1987, pp. 435–444.

[6] Bardet M., Faugère J.C., Salvy B., *On the complexity of the F5 Gr'obner basis algorithm*. Journal of Symbolic Computation, 2014, 70, pp. 1–24.

[7] Faugère J.C., Safey El Din M., Spaenlehauer P.J., *Gröbner Bases of Bihomogeneous Ideals Generated by Polynomials of Bidegree (1,1): Algorithms and Complexity*. Journal of Symbolic Computation, 2011, 46(4), pp. 406–437 Available online 4 November 2010.

[8] Adamus E., Bogdan P., Crespo T., Hajto Z., *An effective study of polynomial maps*, 2016, submitted.

[9] Developers T.S., *Sage Mathematics Software (Version 7.1)*. 2015.

[10] Winkler F., *Polynomial Algorithms in Computer Algebra*. Texts & Monographs in Symbolic Computation. Springer Vienna, 1996.

[11] van den Essen A., *Polynomial Automorphisms: and the Jacobian Conjecture (Progress in Mathematics)*. Birkhuser, 2000.

[12] Yagzhev A.V., *Keller's problem*. Siberian Mathematical Journal, 1980, 21(5), pp. 747–754.

[13] Bass H., Connell E.H., Wright D., *The Jacobian conjecture: Reduction of degree and formal expansion of the inverse*. Bulletin of the AMS – American Mathematical Society (NS), 1982, 7(2), pp. 287–330.

[14] Druzkowski L.M., *An Effective Approach to Keller's Jacobian Conjecture*. Mathematische Annalen, 1983, 264, pp. 303–314.