An Implementation of Faugère's F_4 Algorithm for Computing Gröbner Bases

by Daniel Cabarcas

Committee Chair: Prof. Dr. Dieter Schmidt

A thesis submitted to the Graduate School of the

University of Cincinnati

in partial fulfillment of the requirements for the degree of

Master of Science

in the Department of Computer Science

of the College of Engineering

May 2010

Abstract

Preface

Gröbner bases are the single most important tool of applied algebraic geometry. Both, inside mathematics, as standard bases for ideals that make trivial the ideal membership problem, and outside mathematics, as an intermediate step to solve systems of polynomial equations.

Finding a Gröbner basis is a difficult task, that requires lots of computational resources. Algorithms to compute Gröbner bases have evolved a great deal since the first one was proposed in 1965 by Bruno Buchberger. A significant jump in performance was achieved in 1999 with the introduction of the F_4 algorithm by Jean-Charles Faugère.

Progressed slowed down after 99, a decade passed by with barely marginal improvements. In fact, only a few implementations of the F_4 algorithm are known. We believe that the lack of an independent implementation has restrained research in this area. That is why we have engaged in the challenge of producing an efficient, independent, and open implementation of the F_4 algorithm, hoping to thrust the development of Gröbner basis computation.

F_4 is a rich algorithm that involves complicated operations over large amounts of data, thus its implementation required a fine balance between good software engineering practices and a close attention to optimization details.

Despite the importance of Gröbner bases, and despite the recognized efficiency of the F_4 algorithm, very little has been written about it. In particular, it is hard to tell what exactly makes it fast. Through an independent implementation we try to answer this question in the following pages.

In the process, we give our own vision of the algorithm, of its theoretical ground and an intuitive perception of the mechanism. We embrace the challenge of making the exposition accessible to people not familiar with algebraic geometry, while delivering a mathematically sound discourse.

The thesis is organized as follows. Chapter 1 is a brief introduction to the topic of Gröbner bases. Chapter 2 provides a presentation, both intuitive and theoretical, of the F_4 algorithm. In Chapter 3 we present our implementation of the F_4 algorithm. Chapter 4 exhibits experimental results about our implementation and other Gröbner bases algorithms. In Chapter 5 we state some conclusions and future work related with possible improvements to the F_4 algorithm.

Contents

List of Tables

List of Algorithms

Chapter 1

Gröbner Bases

Buchberger discovered Gröbner bases in 1965. Their importance is well recognized today. Every algebra software provides the functionality to compute Gröbner bases, and every book of applications of algebraic geometry has Gröbner bases as one of its main features.

The main objects of study of Algebraic geometry are a set of polynomials in n variables, $f_1, \ldots, f_m$ and the set of all n-tuples solution to the corresponding system of equations

$$f_1(x_1, \ldots, x_n) = 0,\ f_2(x_1, \ldots, x_n) = 0,\ \ldots\ , f_m(x_1, \ldots, x_n) = 0.$$

Formally, the concept of Gröbner basis is defined for an ideal of a polynomial ring over a field. Intuitively, given a finite set of polynomials, Gröbner basis generalize the concept of triangular basis from the linear case to polynomials of any degree and also generalize the concept of least common multiple from the univariate case to multiple variables.

Similar to a triangular basis for a system of linear equations, a Gröbner basis for a set of polynomials, preserves some of the algebraic properties of the original set, while it reveals im-

portant information. Consider for instance a system of m polynomial equations in n variables $f_1 = 0, \ldots, f_m = 0$. Some of the most remarkable problems that a Gröbner basis allows to solve are:

Ideal Membership: given any polynomial g, do there exist polynomials $h_1, \ldots, h_m$ such that $g = \sum h_i f_i$.

Existence of solution: does there exist any n-tuple that satisfies all the equations?

Dimension of solution space: are there finite or infinitely many solutions to the system of equations? What is the dimension of the solution space?

Solving Equations: Find all common solutions to the system of equations.

Applications in different areas of science and technology abound since polynomials are a common modeling tool. In celestial mechanics for example, recent advances in the *N-Body problem* were possible with large computations of Gröbner bases using algebraic processors. Only few exact solutions exist when $N > 2$. One type of solution is called relative equilibria where the N bodies appear stationary in a rotating coordinate system. Related to it are central configurations, which occur when several bodies collide simultaneously. Central configurations can be found by solving polynomial equations via Gröbner basis. Nevertheless, even for $N = 4$ these equations are too complicated so that a complete classification of all relative equilibria has not yet been achieved.

Gröbner baseshave also been used in robotics, where the configurations of a robot arm can be described by polynomial equations where the variables are the position of the hand and the configuration of the joints. Gröbner bases can be used to solve the so called *inverse kinematic*

problem, given a position of the hand, determine all configurations of the joints of the arm that place the arm at such position.

Also in Cryptology Gröbner bases are playing an important role, since most cryptographic function can be written as polynomial functions, and attacks can be stated as systems of polynomial equations. In some cases, using Gröbner bases to solve the system may be faster that a brute force attack.

However, many of this potential applications are limited by the speed and memory requirements of algorithms and implementations for Gröbner bases computation.

In the rest of this chapter, we define formally a Gröbner basis, then present the original algorithm to compute a Gröbner basis and the most important improvements to the algorithm before 1999.

1.1 Definition

Let us start by stating some basic vocabulary to describe polynomials. Most of our notation is in accordance with [9].

We denote by K an arbitrary *field*. If the reader is not familiar with the formal definition of a field, he or she may think of a field as any set where addition, subtraction, multiplication and division are defined and obey the usual properties. Given some natural number n, we denote by $\underline{x}$ the *sequence of variables* $x_1, \ldots, x_n$. We denote by $K[x_1, \ldots, x_n]$ or $K[\,\underline{x}\,]$ the *ring of polynomials* in n variables.

We will call a *monomial*, any product of the form $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, where $\alpha := (\alpha_1, \ldots, \alpha_n)$ is any n-tuple of natural numbers. For simplicity, we may denote such monomial simply by $\underline{x}^{\alpha}$ and

we will denote arbitrary monomials with lowercase letters s or t. The *degree* of $\underline{x}^\alpha$ is defined as $\deg(\underline{x}^\alpha) := \sum_{i=1}^n \alpha_i$. The *set of all monomials* will be denoted by M,

$$M := \{\underline{x}^\alpha \mid \alpha \in \mathbb{N}_0^n\}.$$

With this notation, any element $f \in K[\underline{x}]$ is a linear combination of monomials

$$f = \sum_{s \in M} a_s s, \quad a_s \in K$$

where finitely many coefficients a_s are non-zero. If $a_s \neq 0$, we will refer to $a_s s$ as a *term* in f and to a_s as the *coefficient* of s in f.

The function lcm (least common multiple), and the relations of being multiple of and being divisible by are defined in the natural way in the set of monomials M. Given $s, t \in$ M we say that s divides t ($s \mid t$) if and only if there exist $s' \in M$ such that $s \cdot s' = t$; we say that t is a multiple of s if and only if s divides t; $\operatorname{lcm}(s,t)$ is the multiple of both s and t that divides any other common multiple of s and t.

An ideal is a general concept in algebra and it is important for our purposes, but we'll try to keep it as concrete and simple as possible. We refer the reader to any abstract algebra text book ([14] for instance) for a more complete treatment.

Definition 1.1. The ideal generated by m polynomials $f_1, \ldots, f_m \in K[\underline{x}]$, denoted by $\langle f_1, \ldots, f_m \rangle$, is the set of all sums of polynomials of the form $h_i f_i$ where h_i is any polynomial and $1 \leq i \leq m$, i.e.

$$\langle f_1, \ldots, f_m \rangle := \left\{ \sum_{i=1}^m h_i f_i \mid h_i \in K[\underline{x}] \right\}.$$

The definition of a Gröbner basis starts with the establishment of an adequate order among monomials, in particular, Gröbner bases are different depending on the chosen order. First we need to precisely state what we mean by adequate.

Definition 1.2. A *monomial order* is a relation $<$ on M satisfying

1. $<$ is a total order (i.e. transitive, anti-symmetric and for all $s, t \in M$, $s = t$ or $s < t$ or $t < s$).
2. For all $s, t_1, t_2 \in M$, if $t_1 < t_2$ then $s \cdot t_1 < s \cdot t_2$.
3. $<$ is a well ordering (i.e. every non-empty set has a smallest element).

There are many different orders that do satisfy this conditions, and some stand out for different reasons. For example, the *lexicographic order* is useful to solve systems of polynomial equations [6] and the *graded reverse lexicographic order* is usually the fastest to compute a Gröbner basis for [1]. There are algorithms to transform a Gröbner basis in one order to a Gröbner basis in another order, such as FGLM [17] and Gröbner Walk [8]. For simplicity, in our examples we will stick to a single monomial order, the *graded lexicographic order*, which is fast yet useful, easy to define and intuitive.

Definition 1.3. Given $\alpha, \beta \in \mathbb{N}_0^n$ we say that $\underline{x}^\alpha < \underline{x}^\beta$ in *graded lexicographic order* if

- $\deg(\underline{x}^\alpha) < \deg(\underline{x}^\beta)$ or
- $\deg(\underline{x}^\alpha) = \deg(\underline{x}^\beta)$ and, in $\beta - \alpha$, the left most nonzero entry is positive

So, for example, $x_1^3 x_2^2 < x_2 x_3^5$ due to the degree difference but $x_1^3 x_2 x_3 < x_1^3 x_2^2$ because in the difference of the corresponding tuples of exponents. In $(3, 2, 0) - (3, 1, 1) = (0, 1, -1)$ the left most nonzero entry is positive.

With a monomial order in place, we can sort the monomials of a polynomial. The largest is specially important thus we give it a distinctive name as follows.

Definition 1.4. Let $f = \sum_{s \in M} a_s s \in K[\underline{x}]$, $a_s \in K$ and $<$ a monomial order

- The set of *monomials* of f is $\mathrm{M}(f) := \{s \mid a_s \neq 0\}$.
- The *leading monomial* of f is $\mathrm{LM}(f) := \max(\mathrm{M}(f))$.
- The *leading coefficient* of f is the coefficient of its leading monomial. $\mathrm{LC}(f) = a_{LM(f)}$
- The *leading term* of f is $\mathrm{LT}(f) := \mathrm{LC}(f)\,\mathrm{LM}(f)$.

Now we're ready to define what a Gröbner basis is.

Definition 1.5. Let $f_1, \ldots, f_m$ be polynomials in $K[\underline{x}]$ and $I := \langle f_1, \ldots, f_m \rangle$ be the ideal generated by them. We say that $\{g_1, \ldots, g_r\} \subseteq I$ is a ***Gröbner basis*** for the ideal I with respect to (w.r.t.) a monomial order $<$ if

$$\mathrm{LM}(I) \subset \langle \mathrm{LM}(g_1), \ldots, \mathrm{LM}(g_r) \rangle \tag{1.1}$$

It is easy to show that the ideal generated by $g_1, \ldots, g_r$ is the same as I. So, in a sense, a Gröbner basis is a more complete set of generators for the same ideal, one that contains enough leading monomials.

There is extensive literature on Gröbner bases. It has been proved that Gröbner bases always exist and that they are unique if an extra condition is added to the definition (usually named *reduced Gröbner basis*). There are abundant characterizations and important properties. There are also extensions to other algebraic structures. However, the presentation of those results goes beyond the scope of this work. We refer the reader to the original work by Buchberger [4] which

has been translated to English and republished, also to a survey by the same author [6] and a book by T. Becker, Kredel and Weispfenning [2] for more information.

1.2 Buchberger's Original Algorithm

Bruno Buchberger not only introduced the concept of Gröbner basis, proved its existence and showed important properties, but he also presented the first algorithm to compute a Gröbner basis. With the goal of definition 1.5 in mind, the algorithm proceeds by enlarging the set of generators with elements from I that contribute new leading monomials. The polynomials added are carefully chosen to guarantee correctness and termination of the algorithm using the so call S-polynomials and an algorithm `normal_form`.

Definition 1.6. Let $f, g \in K[\underline{x}]$. The *S-polynomial* of f and g is defined by

$$\text{S-poly}(f,g) := \frac{t}{\text{LT}(f)} \cdot f - \frac{t}{\text{LT}(g)} \cdot g$$

where $t = \text{lcm}(\text{LM}(f), \text{LM}(g))$.

The intuition behind this definition is that, aiming at canceling the leading terms of f and g to generate new leading monomials, we multiply each by the smallest term such that the results have equal leading term and then subtract. For instance,

$$\begin{aligned}
\text{S-poly}(3x^2y - 1,\ 5xy^2 - 1) &= \frac{x^2y^2}{3x^2y}(3x^2y - 1) - \frac{x^2y^2}{5xy^2}(5xy^2 - 1) \\
&= y/3(3x^2y - 1) - x/5(5xy^2 - 1) \\
&= (x^2y^2 - y/3) - (x^2y^2 - x/5) \\
&= x/5 - y/3
\end{aligned}$$

Given polynomials f and g we say that f *reduces* g if $\mathrm{LM}(f) \mid \mathrm{LM}(g)$. The reduction refers to the action of canceling the leading term of g by subtracting a suitable multiple of f, that is, $\frac{\mathrm{LT}(g)}{\mathrm{LT}(f)} \cdot f$, to obtain $h := g - \frac{\mathrm{LT}(g)}{\mathrm{LT}(f)} \cdot f$. Then, we say that g reduces to h modulo f. This reduction step can be viewed as a step in a sort of division algorithm. Algorithm 1.1 follows this idea.

Algorithm 1.1 `normal_form`(g, F)

Require: F is a finite subset of $K[\underline{x}]$
Require: $g \in K[\underline{x}]$
1: $h := g$
2: **while** there exist $f \in F$ such that $\mathrm{LM}(f) \mid \mathrm{LM}(h)$ **do**
3: choose $f \in F$ such that $\mathrm{LM}(f) \mid \mathrm{LM}(h)$
4: $h := h - \frac{\mathrm{LT}(h)}{\mathrm{LT}(f)} \cdot f$
5: **return** h

Algorithm 1.2 presents the Buchberger algorithm for computing Gröbner bases as stated in [6].

Algorithm 1.2 `buchberger`(F)

Require: F is a finite subset of $K[\underline{x}]$
1: $G := F$
2: $B := \{\{g_1, g_2\} \mid g_1, g_2 \in G, g_1 \neq g_2\}$
3: **while** $B \neq \emptyset$ **do**
4: let $\{g_1, g_2\}$ be an element of B
5: $B := B \setminus \{\{g_1, g_2\}\}$
6: $h := \text{S-poly}(g_1, g_2)$
7: $r := \texttt{normal\_form}(h, G)$
8: **if** $r \neq 0$ **then**
9: $B := B \cup \{\{g, r\} \mid g \in G\}$
10: $G := G \cup \{r\}$
11: **return** G

The correctness of the algorithm relies on the following theorem proved by Buchberger in a more general statement in [4].

Theorem 1.1. *Let $G = \{g_1, \ldots, g_m\}$ be a subset of $K[\underline{x}]$ and let I be the ideal generated by G.*

Then G is a Gröbner basis for I if and only if for all $i \neq j$ it holds that

$$\mathit{normal\_form}(\text{S-poly}(g_i, g_j), G) = 0$$

.

Given a pair $\{g_1, g_2\}$ of elements from G, $h := \text{S-poly}(g_1, g_2)$, and $r := \texttt{normal\_form}(h, G)$, the idea behind Buchberger's algorithm is to append r to G in order to ensure that $\texttt{normal\_form}(h, G) = 0$. Termination of Algorithm 1.2 is guaranteed by Hilbert's theorem on ascending chains of ideals in $K[\underline{x}]$, that states that any strictly ascending chain of ideals must be finite. In our case, the ideals generated by the leading monomials of G after line 10 constitute an ascending chain, hence the algorithm must terminate.

1.3 Improvements until 1999

During the decades after the discovery of Gröbner bases, especially during the eighties and beginning of the nineties, Buchberger himself and other researchers around the world, directed great efforts toward speeding up the Buchberger algorithm. At the beginning of the eighties, there was no implementation efficient enough to solve non trivial problems. Three main improvements were vastly explored:

(1) The order of selection of pairs (line 4, Algorithm 1.2). Although it has no influence on the final outcome, it has great impact on the complexity. Buchberger proposed the *normal strategy* [4] which chooses a pair with minimal lcm with respect to the monomial order. Giovini et. al. proposed to choose a minimal pair with respect to its *sugar flavor* [19], which is the degree

it would have after being homogenized, and they claim this is a more stable measure for non-graded orders.

(2) When a new element is adjoined to the basis (line 10, algorithm 1.2) this new element may reduce other elements in the basis. Thus, as said in [6], "such reductions may initiate a whole cascade of reductions and cancellations." This idea however complicates the management of the pairs to guarantee correctness, since a change in the basis G, forces changes in the set of pairs B. The solution is roughly, to treat reduced elements from G as new elements, thus eliminate old pairs involving them and form new pairs with them.

(3) Criteria to discard pairs a-priori that are known to reduce to zero. Notably, the Buchberger criteria, introduced in [5] and the Gebauer-Möller criteria introduced in [18]. Also, under this category, lay the attempts by Möller, Mora and Traverso to maintain a set of relations (mathematically known as syzygies) to avoid as many useless pairs as possible [29].

Algorithm 1.3 presents an improved version of the Buchberger algorithm that includes these ideas. It is attributed to Gebauer-Möller [18], and presented in a clear manner in [2], where the reader can also find a proof of correctness.

Algorithm 1.4 shows the `update` function, which has become a crucial part of the Buchberger algorithm and later became part of the $F4$ algorithm. This function is in charge of inserting new elements into the set G and new pairs in B while avoiding redundancies.

A special chapter of the race to improve Gröbner bases computation was written by Daniel Lazard. In 1979 he presented an independent approach for solving systems of polynomial equations based on multivariate resultants [23]. His approach turned out to be closely related to Gröbner

Algorithm 1.3 `groebner`(F)

Require: F is a finite subset of $K[\underline{x}]$

1: $G := \emptyset$
2: $B := \emptyset$
3: **for all** f in F **do**
4: $\quad (G, B) :=$ `update`(G, B, f)
5: **while** $B \neq \emptyset$ **do**
6: $\quad$ let $\{g_1, g_2\}$ be an element of B
7: $\quad B := B \setminus \{\{g_1, g_2\}\}$
8: $\quad h :=$ S-poly(g_1, g_2)
9: $\quad r :=$ `normal_form`(h, G)
10: $\quad$ **if** $r \neq 0$ **then**
11: $\quad\quad (G, B) :=$ `update`(G, B, r)
12: **return** G

bases [24], but his perspective of the problem yielded great success years later when his student Jean-Charles Faugère proposed the F_4 algorithm.

Algorithm 1.4 `update`(G, B, h)

Require: G is a finite subset of $K[\underline{x}]$
Require: B is a finite set of pairs of elements from $K[\underline{x}]$.
Require: $0 \neq h \in K[\underline{x}]$
1: $C := \{\{h, g\} \mid g \in G\}$
2: **for all** $\{h, g_1\} \in C$ **do**
3: **if** (LM(h) and LM($g1$) are NOT disjoint) **and**
(there exist $\{h, g_2\} \in C \setminus \{\{h, g_1\}\}$ **s.t.**
$\mathrm{lcm}(\mathrm{LM}(h), \mathrm{LM}(g_2)) \mid \mathrm{lcm}(\mathrm{LM}(h), \mathrm{LM}(g_1)))$ **then**
4: $C := C \setminus \{h, g_1\}$
5: **for all** $\{h, g\} \in C$ **do**
6: **if** LM(h) and LM(g) are disjoint **then**
7: $C := C \setminus \{h, g\}$
8: **for all** $\{g_1, g_2\} \in B$ **do**
9: **if** $(\mathrm{LM}(h) \mid \mathrm{lcm}(\mathrm{LM}(g_1), \mathrm{LM}(g_2))$ **and**
$(\mathrm{lcm}(\mathrm{LM}(g_1), \mathrm{LM}(h)) \neq \mathrm{lcm}(\mathrm{LM}(g_1), \mathrm{LM}(g_2)))$ **and**
$(\mathrm{lcm}(\mathrm{LM}(h), \mathrm{LM}(g_2)) \neq \mathrm{lcm}(\mathrm{LM}(g_1), \mathrm{LM}(g_2)))$ **then**
10: $B := B \setminus \{g_1, g_2\}$
11: $B := B \cup C$
12: **for all** $g \in G$ **do**
13: **if** $\mathrm{LM}(h) \mid \mathrm{LM}(g)$ **then**
14: $G := G \setminus \{g\}$
15: **return** G, B

Chapter 2

Faugère's F_4 algorithm

In 1999 the field of applicable algebraic geometry received a huge thrust with the publication by Jean-Charles Faugère of [15], where the F_4 algorithm to compute Gröbner bases was introduced. By combining some of the best techniques described in section 1.3 with well established matrix techniques, Faugère was able to take Gröbner bases computation to a new level. In this chapter we describe the algorithm. As we did with Buchberger algorithm in Chapter 1, first we will present a basic version, and then a more complete one.

2.1 Basic F_4

The F_4 algorithm is similar to the Buchberger algorithm –1.2– in that it traverses the set of all pairs of elements from the basis, but it differs in that it selects and process many pairs at a time instead of a single one like the Buchberger algorithm does. The ideas of an S-polynomial and that of a normal form are present in F_4, but hidden behind symbolic constructions and matrix

computations. We shall try to show the connection between F_4 and Buchberger's, as we present the algorithm. For this purpose we deviate a bit from the notation used by Faugère.

Algorithm 2.1 `basic_`$F_4(F)$

Require: F is a finite subset of $K[\underline{x}]$

1: $G := F$
2: $B := \{\{g_1, g_2\} \mid g_1, g_2 \in G, g_1 \neq g_2\}$
3: **while** $B \neq \emptyset$ **do**
4: let B^* be a nonempty subset of B
5: $B := B \setminus B^*$
6: $L := \left\{ \frac{\text{lcm}(\text{LM}(f),\text{LM}(g))}{\text{LM}(f)} \cdot f \mid \{f, g\} \in B^* \right\}$
7: $H :=$ `basic_symb_pre_proc`(L, G)
8: $\widetilde{H} :=$ a row echelon form of H
9: $\widetilde{H}^+ := \{h \in \widetilde{H} \mid \text{LM}(h) \notin \text{LM}(H)\}$
10: $G := G \cup \widetilde{H}^+$
11: $B := B \cup \{\{h, g\} \mid h \in \widetilde{H}^+, g \in G, h \neq g\}$
12: **return** G

Algorithm 2.1 shows pseudo-code for the F_4 algorithm. Note the similarities with Buchberger's algorithm. The initialization steps (lines 1 and 2) are identical, as well as the while loop condition. In line 4 a subset of pairs is selected, instead of a single element. Lines 6 through 9 are the equivalent to the S-poly and `normal_form` instructions in the Buchberger algorithm. Since may pairs were processed simultaneously, then, many new polynomials are produced, which are appended to the basis in line 10, and then all pairs of these new elements with old and new elements are appended to B in line 11.

The key of F_4 is the way it processes simultaneously many pairs. The set L, constructed in line 6 of Algorithm 2.1, groups the building blocks of S-polynomials from pairs in B^*. Note that a S-polynomial of f and g (definition 1.6) can be written as

$$\frac{1}{\text{LC}(f)} \cdot \frac{\text{lcm}(\text{LM}(f), \text{LM}(g))}{\text{LM}(f)} \cdot f - \frac{1}{\text{LC}(g)} \cdot \frac{\text{lcm}(\text{LM}(f), \text{LM}(g))}{\text{LM}(g)} \cdot g$$

hence, all the S-polynomials from elements in B^* belong to the linear span of L.

The set H, formed in line 7, appends to L every polynomial that may be useful to linearly reduce polynomials in L. The `basic_symb_pre_proc` algorithm –2.2– describes how this is accomplished. After initializing H with L, the algorithm traverses every monomial t in H and searches for a polynomial $g \in G$ that reduces t. Note that the polynomial $h = \frac{t}{\mathrm{LM}(g)} * g$, appended in line 8, has t as its leading monomial. Note also, that after inserted in H, the polynomial h changes its role from being chosen to reduce others to be subject of reduction, as the set $\mathrm{M}(H)$ increases for the condition of the while loop $\mathrm{M}(H) \neq done$; this is crucial to guarantee termination of the F_4 algorithm as we will see below.

Algorithm 2.2 `basic_symb_pre_proc`(L, G)

Require: L and G are finite subsets of $K[\underline{x}]$
1: $H := L$
2: $done := \mathrm{LM}(H)$
3: **while** $\mathrm{M}(H) \neq done$ **do**
4: let t be an element of $(\mathrm{M}(H) \setminus done)$
5: $done = done \cup \{t\}$
6: **if** there exist $g \in G$ s.t. $\mathrm{LM}(g) \mid t$ **then**
7: choose $g \in G$ s.t. $\mathrm{LM}(g) \mid t$
8: $H := H \cup \{\frac{t}{\mathrm{LM}(g)} * g\}$
9: **return** H

The row echelon form (line 8 of Algorithm 2.1) is a massive reduction of the polynomials of H via linear combinations, that can be done using Gaussian elimination on a matrix. We define it as follows.

Definition 2.1. Let F be a finite subset of $K[\underline{x}]$ and $<$ a monomial order.

- F is said to be in *row echelon form* with respect to $<$, if the elements of F are non-zero and have pairwise different leading monomials.

- F is said to be in *row reduced echelon form* with respect to $<$, if every element of F is non-zero and its leading monomial is not a monomial of any other element of F.

- A finite subset $\widetilde{F}$ of $K[\underline{x}]$ is a *row (reduced) echelon form* of F w.r.t. $<$, if $\text{span}(\widetilde{F}) = \text{span}(F)$ and $\widetilde{F}$ is in row (reduced) echelon form.

Finally, line 9 of Algorithm 2.1, is nothing else than a purge of the result of the massive reduction in order to guarantee that only polynomials that contribute new leading monomials are appended to the basis. This is guaranteed by the fact that every polynomial $h \in H$ is of the form $h = t \cdot g$ where $g \in G$ and $t \in \text{M}$, so that $\text{LM}(h) \in \langle \text{LM}(G) \rangle$, and by the `basic_symb_pre_proc` algorithm as we will see in the proof of Lemma 2.2.

We present the proofs of correctness and termination of the algorithm in a slightly different fashion than in [15]. The following lemmas lead to the main result.

Lemma 2.1. *If $F \subset K[\underline{x}]$ is in row echelon form and $r \in \text{span}(F)$ then either $r = 0$ or else $\text{LM}(r) \in \text{LM}(F)$.*

Proof. Given $a, b \in K$ and $f, g \in F$, since F is in row echelon form, either $f = g$ or f and g have distinct leading monomials, thus either $af + bg = 0$ or else the leading monomial of $af + bg$ coincides with that of f or g. The result follows then by induction. □

Next lemma is the key for the termination of the algorithm. It states that polynomials in $\widetilde{H}^+$ after line 9 of Algorithm 2.1 posses leading monomials that are not in G, thus their insertion into G expands the ideal generated by the leading monomials of G.

Lemma 2.2. *Let G be a finite subset of $K[\underline{x}]$, L a finite subset of $\{tg \mid t \in M, g \in G\}$, $H :=$*

basic_symb_pre_proc(L, G), $\widetilde{H}$ *a row echelon form of* H *and* $\widetilde{H}^+ := \{h \in \widetilde{H} \mid \mathrm{LM}(h) \notin \mathrm{LM}(H)\}$. *Then for all* $h \in \widetilde{H}^+$, $\mathrm{LM}(h) \notin \langle \mathrm{LM}(G) \rangle$.

Proof. Assume for a contradiction that $h \in \widetilde{H}^+$ and $\mathrm{LM}(h) \in \langle \mathrm{LM}(G) \rangle$. Then, there exist $g \in G$, such that $\mathrm{LM}(g) \mid \mathrm{LM}(h)$. Note that $\mathrm{LM}(h) \in \mathrm{M}(\widetilde{H}^+) \subseteq \mathrm{M}(\widetilde{H}) \subseteq \mathrm{M}(H)$. Hence, when $t = \mathrm{LM}(h)$ is considered in the while loop of `basic_symb_pre_proc`, a polynomial $h' = \frac{t}{\mathrm{LM}(g)} * g$ with $g \in G$ is inserted into H, hence $t = \mathrm{LM}(h') \in \mathrm{LM}(H)$. This contradicts the definition of $\widetilde{H}^+$. □

We borrow the following definition, theorem and corollary from the general theory of Gröbner bases.

Definition 2.2. Let F be a finite subset of $K[\underline{x}]$, $0 \neq f \in \langle F \rangle$ and $t \in M$. A representation

$$f = \sum_{i=1}^{m} a_i t_i f_i$$

with $a_i \in K$, $t_i \in \mathrm{M}$, and $f_i \in F$ is called a t-representation of f w.r.t. F if $\mathrm{LM}(t_i f_i) \leq t$ for $i = 1, \ldots, m$. A $\mathrm{LM}(f)$-representation of f w.r.t. F is called a *standard representation.*

Theorem 2.3. *[2] Let* G *be a finite subset of* $K[\underline{x}]$ *with* $0 \notin G$, *and assume that for all* $f, g \in G$, $\text{S-poly}(f, g)$ *equals zero or has a* t*-representation w.r.t.* G *for some* $t < \mathrm{lcm}(\mathrm{LM}(f), \mathrm{LM}(g))$. *Then* G *is a Gröbner basis.*

The following corollary is obvious from the fact that $\mathrm{LM}(\text{S-poly}(f, g)) < \mathrm{lcm}(\mathrm{LM}(f), \mathrm{LM}(g))$.

Corollary 2.4. *[2] Let* G *be a finite subset of* $K[\underline{x}]$ *with* $0 \notin G$, *and assume that for all* $f, g \in G$, $\text{S-poly}(f, g)$ *equals zero or has a standard representation w.r.t.* G. *Then* G *is a Gröbner basis.*

The following lemma is the key for the correctness of the F_4 algorithm since it guarantees that polynomials in $\mathrm{span}(H)$, after line 6 of Algorithm 2.1, have a standard representation w.r.t G after

$\widetilde{H}^+$ is inserted in line 8. In particular, as we will see later on, S-polynomials of pairs in B^* belong to the span of $L \subset H$.

Lemma 2.5. *Let G be a finite subset of $K[\underline{x}]$, H a finite subset of $\{tg \mid t \in M, g \in G\}$, $\widetilde{H}$ a row echelon form of H and $\widetilde{H}^+ := \{h \in \widetilde{H} \mid \mathrm{LM}(h) \notin \mathrm{LM}(H)\}$. Then*

i) $\widetilde{H}^+ \subset \langle G \rangle$,

ii) for all $f \in \mathrm{span}(H)$, f has a standard representation w.r.t. $G \cup \widetilde{H}^+$, and

iii) for all $h \in H$, there exist a unique $h' \in \widetilde{H}$, such that $\mathrm{LM}(h) = \mathrm{LM}(h')$ and $r := h - \frac{\mathrm{LC}(h)}{\mathrm{LC}(h')}h'$ satisfies that $\mathrm{LM}(r) < \mathrm{LM}(h)$ and r has a standard representation w.r.t. $G \cup \widetilde{H}^+$.

Proof. i) Note that $H \subset \{tg \mid t \in M, g \in G\}$ and that $\mathrm{span}(H) = \mathrm{span}(\widetilde{H})$ so, clearly $\widetilde{H}^+ \subseteq \widetilde{H} \subset \langle G \rangle$.

ii) Note that by definition of $\widetilde{H}^+$, $\mathrm{LM}(\widetilde{H}) = \mathrm{LM}(H \cup \widetilde{H}^+)$. Let $f \in \mathrm{span}(H)$. Clearly $f \in \mathrm{span}(\widetilde{H})$, and by Lemma 2.1, $f = 0$ or $\mathrm{LM}(f) \in \mathrm{LM}(\widetilde{H}) = \mathrm{LM}(H \cup \widetilde{H}^+)$. If $f = 0$ we are done, so assume $f \neq 0$, thus, we can choose $g_1 \in H \cup \widetilde{H}^+$ such that $\mathrm{LM}(g_1) = \mathrm{LM}(f)$ and define

$$f_1 := f - \frac{\mathrm{LC}(f)}{\mathrm{LC}(g_1)} g_1.$$

Then, $\mathrm{LM}(f_1) < \mathrm{LM}(f)$ and $f_1 \in \mathrm{span}(H \cup \widetilde{H}) = \mathrm{span}(H)$. We can then repeat the process and because each time $\mathrm{LM}(f_{i+1}) < \mathrm{LM}(f_i)$ and $<$ is a well ordering, eventually, for some m we must have that $f_m = 0$. Then we get that

$$f = \sum_{i=1}^{m} b_i g_i \tag{2.1}$$

with $b_i \in K$, $g_i \in H \cup \widetilde{H}^+$ and by noting that $H \subset \{tg \mid t \in M, g \in G\}$ we conclude that (2.1) is a standard representation of f w.r.t. $\widetilde{H}^+ \cup G$.

iii) Given $h \in H$, since $h \in \text{span}(H) = \text{span}(\widetilde{H})$ and $\widetilde{H}$ is in row echelon form, by lemma 2.1, there exist a unique $h' \in \widetilde{H}$ such that $\text{LM}(h) = \text{LM}(h')$. Let $r := h - \frac{\text{LC}(h)}{\text{LC}(h')}h'$. Clearly by construction $\text{LM}(r) < \text{LM}(h)$ and since $r \in \text{span}(H) = \text{span}(\widetilde{H})$, by part *ii)* of the lemma it must have a standard representation w.r.t. $G \cup \widetilde{H}^+$.

□

Parts *i)* and *ii)* of the previous lemma will be used to prove the main theorem of this section below, while part *iii)* will be used in next section to prove an improvement of the F_4 algorithm.

Theorem 2.6. *Given a finite set of polynomials F, the algorithm* `basic_F4` *computes a Gröbner basis G of $\langle F \rangle$.*

Proof. Let $G_0 = F$ and for $i \geq 0$ $G_{i+1} = G_i \cup \widetilde{H}_i^+$, where $\widetilde{H}_i^+$ is the i^{th} nonempty set produced by `basic_F4` (Algorithm 2.1). Let $B_0 = \{\{f_1, f_2\} \mid f_1, f_2 \in G, f_1 \neq f_2\}$ and for $i \geq 0$ $B_{i+1} = B_i \cup \{\{h, g\} \mid h \in \widetilde{H}_i^+, g \in G_{i+1}, h \neq g\}$.

Termination: By Lemma 2.2, for $i \geq 0$, $\langle \text{LM}(G_i) \rangle$ is a proper subset of $\langle \text{LM}(G_{i+1}) \rangle$. By Hilbert's Theorem on ascending chains of ideals in $K[\underline{x}]$, that states that any strictly ascending chain of ideals must be finite, the sequence $G_0, G_1, \ldots$ must be finite, hence, $\widetilde{H}^+$ after line 9 must be empty from some point on, thus the set B does not grow anymore and since B^* is nonempty, B will shrink on every subsequent pass through the while loop, thus eventually becoming empty, which causes the while loop to end.

Let $G_0, G_1, \ldots, G_N$ be the complete sequence, so that $G_{out} := G_N$ is the final value of G returned by Algorithm 2.1.

Correctness: By Lemma 2.5 *i)*, for $i \geq 0$ $\widetilde{H}_i^+ \subset \langle G_i \rangle$, thus $\langle G_i \rangle = \langle G_{i+1} \rangle$, hence $\langle G_{out} \rangle =$

$\langle G_0 \rangle = \langle F \rangle$.

In order to prove that G_{out} is a Gröbner basis, we will show that for all $g_1, g_2 \in G_{out}$, S-poly(g_1, g_2) has a standard representation w.r.t. G_{out}. Let $g_1, g_2 \in G_{out}$. Note that $B_i = \{\{f, g\} \mid f, g \in G_i, f \neq g\}$, thus at some point the pair $\{g_1, g_2\}$ must be chosen to be part of B^*. In that pass throw the while loop, $L = \left\{t \cdot f \mid \{f, g\} \in B^*,\ t \in \mathrm{M},\ t = \frac{\mathrm{lcm}(\mathrm{LM}(f),\mathrm{LM}(g))}{\mathrm{LM}(f)}\right\}$, thus $t_1 g_1, t_2 g_2 \in L$ with $t_i = \frac{\mathrm{lcm}(\mathrm{LM}(g_1),\mathrm{LM}(g_2))}{\mathrm{LM}(g_i)}$ for $i = 1, 2$. Hence,

$$\text{S-poly}(g_1, g_2) = \frac{\mathrm{lcm}(\mathrm{LM}(g_1), \mathrm{LM}(g_2))}{\mathrm{LT}(g_1)} \cdot g_1 - \frac{\mathrm{lcm}(\mathrm{LM}(g_1), \mathrm{LM}(g_2))}{\mathrm{LT}(g_2)} \cdot g_2 = \frac{t_1}{\mathrm{LC}(g_1)} \cdot g_1 - \frac{t_2}{\mathrm{LC}(g_2)} \cdot g_2 \in \mathrm{span}(L).$$

Therefore, by Lemma 2.5 *ii)*, S-poly(g_1, g_2) has a standard representation w.r.t. G_{out}, and Corollary 2.4 completes the proof. □

2.2 Improved F_4 Algorithm

In [15], Faugère presents an improved F_4 algorithm that combines several ideas. Here, we separate the improvements into 5 different independent points, some mentioned explicitly on the paper, and some derived from it. The first three are related to improvements previously studied for Buchberger's algorithm while the last two are particular to F_4.

(1) The selection strategy for pairs, i.e. the way B^* is chosen in line 4 of algorithm 2.1, is critical for the efficiency of the algorithm.

(2) When a new element is adjoined to the basis, this new element may reduce other elements in the basis.

(3) We can use criteria to discard pairs a-priori that are known to reduce to zero.

(4) In the symbolic pre-processing algorithm –2.2–, in line 7, there is an obvious choice that we can use to our advantage.

(5) It is possible to substitute the product $t \cdot f$, ($t = \frac{\text{lcm}(\text{LM}(f),\text{LM}(g))}{\text{LM}(f)}$) of line 6 of the F_4 algorithm –2.1–, by a different product $t'f'$ under certain conditions. We can use this flexibility to our advantage.

For improvement (1), Faugère suggests using the normal strategy for F_4 which consists on choosing all the pairs in B of minimum degree, where the *degree of a pair* is the degree of the least common multiple of the leading monomials. Improvement (2) is not explicitly mentioned in [15] but used by the commercial software Magma [3] and tested in [26]. For improvement (3), Faugère suggests using an `update` function like the one presented in section 1.3.

Improvements (4) and (5) are treated as one in [15]. We consider them separately here. A discussion on (4) is delayed to section 5.1.1. For the rest of this section we discuss the theoretical foundation of improvement (5), and present the improved F_4 algorithm as described in [15].

Consider Algorithm 2.3 which introduces some modifications to the basic F_4 algorithm –2.3–. The function $\texttt{mult} : M \times K[\underline{x}] \to K[\underline{x}]$ is defined by $\texttt{mult}(t, f) = tf$ and Algorithm 2.4 defines the `simplify` function. Besides the change in line 7, the other changes have no effect on the algorithm. we are simply keeping a counter j and store H and $\widetilde{H}$ for each pass through the loop with unique names according to the counter.

The `simplify` algorithm –2.4– recursively changes a monomial-polynomial pair (t, f) into another pair $(t/u, p)$ where u is a non-trivial divisor of t, $uf \in H_j$ for some j, and p is the unique polynomial in $\widetilde{H}_j$ such that $\text{LM}(p) = \text{LM}(uf)$. `simplify`clearly terminates, since each recursive

Algorithm 2.3 `improved_`$F_4(F)$

Require: F is a finite subset of $K[\underline{x}]$
1: $G := F$
2: $B := \{\{f_1, f_2\} \mid f_1, f_2 \in G, f_1 \neq f_2\}$
3: $j := 1$
4: **while** $B \neq \emptyset$ **do**
5: let B^* be a nonempty subset of B
6: $B := B \setminus B^*$
7: $L := \left\{\texttt{mult}(\texttt{simplify}(t, f, (H_1, \ldots, H_{j-1}), (\widetilde{H}_1, \ldots, \widetilde{H}_{j-1}))) \mid \{f, g\} \in B^*, t = \frac{\text{lcm}(\text{LM}(f), \text{LM}(g))}{\text{LM}(f)}\right\}$
8: $H_j := H := \texttt{basic\_symb\_pre\_proc}(L, G)$
9: $\widetilde{H}_j := \widetilde{H} :=$ a row echelon form of H
10: $\widetilde{H}^+ := \{h \in \widetilde{H} \mid \text{LM}(h) \notin \text{LM}(H)\}$
11: $G := G \cup \widetilde{H}^+$
12: $B := B \cup \{\{h, g\} \mid h \in \widetilde{H}^+, g \in G, h \neq g\}$
13: $j := j + 1$
14: **return** G

call reduces the monomial of the pair. We shall prove that this substitution does not affect the correctness of the algorithm. A discussion about the effectiveness of this substitution is postpone to Chapter 5. Our C++ implementation for this algorithm can be found in Appendix A.

Algorithm 2.4 `simplify`$(t, f, (H_1, \ldots, H_J), (\widetilde{H}_1, \ldots, \widetilde{H}_J))$

Require: $f \in K[\underline{x}], t \in \text{M}$
Require: H_j be a finite subset of $K[\underline{x}]$, for $j = 1, \ldots, J$.
Require: $\widetilde{H}_j$ be a row echelon form of H_j, for $i = 1, \ldots, I$.
1: **if** there exist $1 \neq u \in \text{M}$, $j \in \{1, \ldots, J\}$ s.t. $u \mid t$ and $uf \in H_j$ **then**
2: choose $1 \neq u \in \text{M}$, $j \in \{1, \ldots, J\}$ s.t. $u \mid t$ and $uf \in H_j$
3: Let $p \in K[\underline{x}]$ be the unique element of $\widetilde{H}_j$ such that $\text{LM}(p) = \text{LM}(uf)$
4: **return** `simplify`$(t/u, p, (H_1, \ldots, H_J), (\widetilde{H}_1, \ldots, \widetilde{H}_J))$
5: **else**
6: **return** (t, f)

The following lemma provides a useful property of the `simplify` Algorithm. We shall use it to prove `improved_`F_4's correctness.

Lemma 2.7. *Given a finite set of polynomials F, consider the execution of* `improved_`$F_4(F)$

–2.3–. Suppose that right before line 13 is executed, $\{f, g\} \in B^*$, $t = \frac{\mathrm{lcm}(\mathrm{LM}(f),\mathrm{LM}(g))}{\mathrm{LM}(f)}$ *and*

$$(t', f') = \texttt{simplify}(t, f, (H_1, \ldots, H_{j-1}), (\widetilde{H}_1, \ldots, \widetilde{H}_{j-1})).$$

Then $r := tf - \frac{\mathrm{LC}(f)}{\mathrm{LC}(f')} t'f'$ *has a* v*-representation w.r.t.* G *for some* $v < \mathrm{LM}(tf)$.

Proof. For simplicity we assume that all polynomials in G are monic ($\mathrm{LC}(f) = 1$ always). We proceed by induction on the substitutions made by `simplify`. Consider the sequence $(t, f) = (t_0, f_0), \ldots, (t_s, f_s) = (t', f')$ produced by the call to `simplify`. Then for $k > 0$, there exist $u_k \in \mathrm{M}$ and $j_k \in \{1, \ldots, j-1\}$, such that, $u_k \mid t_k$, $u_k f_k \in H_{j_k}$ and $f_{k+1} \in K[\underline{x}]$ is the unique element of $\widetilde{H}_{j_k}$ such that $\mathrm{LM}(f_{k+1}) = \mathrm{LM}(u_k f_k)$ and $t_{k+1} = t_k/u_k$.

Suppose that for some $0 < k \leq s$, $r_k := tf - t_k f_k$ has a v_k-representation w.r.t. G for some $v_k < \mathrm{LM}(tf)$. By Lemma 2.5 *iii)*

$$r' := u_k f_k - f_{k+1}$$

satisfies that $\mathrm{LM}(r') < \mathrm{LM}(u_k f_k)$ and r' has a standard representation w.r.t. G. Then, multiplying by $t_{k+1} = t_k/u_k$ we obtain

$$t_{k+1} r' = t_k f_k - t_{k+1} f_{k+1}$$

that satisfies $\mathrm{LM}(t_{k+1} r') < \mathrm{LM}(t_k f_k) = \mathrm{LM}(tf)$ and $t_{k+1} r'$ has a standard representation w.r.t. G. Then,

$$\begin{aligned} r_k :=& tf - t_k f_k \\ =& tf - \left(t_{k+1} r' + t_{k+1} f_{k+1}\right) \end{aligned}$$

hence

$$r_{k+1} := tf - t_{k+1} f_{k+1} = r_k - t_{k+1} r'$$

satisfies that $\mathrm{LM}(r_{k+1}) < \mathrm{LM}(t_k f_k)$ and r_{k+1} has a v_{k+1}-representation w.r.t. G, where

$$v_{k+1} = \max(v_k, \mathrm{LM}(t_{k+1} r')) < \mathrm{LM}(tf).$$

The result follows by induction on k. □

We can now prove the correctness of the `improved_`F_4 algorithm.

Theorem 2.8. *Given a finite set of polynomials F, algorithm* `improved_`F_4 *computes a Gröbner basis G of $\langle F \rangle$.*

Proof. For simplicity we assume that all polynomials in G are monic (leading coefficient is 1). We rely on the proof of Theorem 2.6. Termination follows the same way. For correctness, we must show that for all $g_1, g_2 \in G_{out}$, S-poly(g_1, g_2) has a v-representation w.r.t. G_{out} for some v strictly less than $\mathrm{lcm}(\mathrm{LM}(g_1), \mathrm{LM}(g_2))$. Let $g_1, g_2 \in G_{out}$. There exist $J_1, J_2 \geq 0$, $t_1', t_2' \in \mathrm{M}$, $g_1', g_2' \in K[\underline{x}]$ such that

$$(t_i', g_i') = \texttt{simplify}(t_i, g_i, (H_1, \ldots, H_{J_i}), (\widetilde{H}_1, \ldots, \widetilde{H}_{J_i})), \text{ for } i = 1, 2.$$

By Lemma 2.7, $r_i := t_i g_i - t_i' g_i'$ has a v_i representation w.r.t. G_{out} for some $v_i < \mathrm{LM}(t_1 g_1) = \mathrm{LM}(t_2 g_2)$. Then,

$$\begin{aligned} \text{S-poly}(g_1, g_2) &= t_1 g_1 - t_2 g_2 \\ &= (t_1 g_1 - t_1' g_1') + (t_1' g_1' - t_2' g_2') - (t_2 g_2 - t_2' g_2') \\ &= r_1 + \text{S-poly}(g_1', g_2') - r_2 \end{aligned}$$

where S-poly(g_1', g_2'), being a linear combination of $t_1' g_1'$ and $t_2' g_2'$, has a standard representation w.r.t. G_{out} (as explained in the proof of Theorem 2.6). Therefore, S-poly(g_1, g_2) has a v-

representation w.r.t. G_{out} where $v = \max(v_1, v_2, \mathrm{LM}(\text{S-poly}(g_1', g_2')))$ is strictly less than $\mathrm{LM}(t_1 g_1) = \mathrm{lcm}(\mathrm{LM}(g_1) m \mathrm{LM}(g_2))$. By Theorem 2.3, G_{out} is a Gröbner basis. □

We conclude this chapter with the F_4 algorithm to compute Gröbner bases as proposed by Faugère in [15]. Algorithm 2.5 is a pseudo-code description with slightly different notation of Faugère's algorithm. Our C++ implementation for this algorithm can be found in Appendix A.

Algorithm 2.5 $F_4(F)$

Require: F is a finite subset of $K[\underline{x}]$

1: $G := \emptyset$
2: $B := \emptyset$
3: $j := 1$
4: **for all** $f \in F$ **do**
5: $\quad (G, B) := \texttt{update}(G, B, f)$
6: **while** $B \neq \emptyset$ **do**
7: $\quad$ let $B^* = \{b \in B \mid \deg(b) \text{ is minimal } \}$
8: $\quad B := B \setminus B^*$
9: $\quad L := \left\{(t, f) \mid \{f, g\} \in B^*, t = \frac{\mathrm{lcm}(\mathrm{LM}(f), \mathrm{LM}(g))}{\mathrm{LM}(f)}\right\}$
10: $\quad H_j := H := \texttt{symb\_pre\_proc}(L, G, (H_1, \ldots, H_{j-1}), (\widetilde{H}_1, \ldots, \widetilde{H}_{j-1}))$
11: $\quad \widetilde{H}_j := \widetilde{H} :=$ a row echelon form of H
12: $\quad \widetilde{H}^+ := \{h \in \widetilde{H} \mid \mathrm{LM}(h) \notin \mathrm{LM}(H)\}$
13: $\quad$ **for all** $h \in \widetilde{H}^+$ **do**
14: $\quad\quad (G, B) := \texttt{update}(G, B, h)$
15: $\quad j := j + 1$
16: **return** G

Note that in lines 5 and 14, the function `update` is called for each new element. Note also that in line 9, monomial-polynomial pairs are inserted into the set L. Then a new version of `symb_pre_proc` function is called in line 10 which is described in Algorithm 2.6. Our C++ implementation for this algorithm can be found in Appendix A.

The function `symb_pre_proc` defined in 2.6 is identical to the one defined in 2.2, except for the calls to `simplify` (see Algorithm 2.4) before inserting elements into H.

Algorithm 2.6 `symb_pre_proc`$(L, G, (H_1, \ldots, H_J), (\widetilde{H}_1, \ldots, \widetilde{H}_J))$

Require: L a finite subset of $M \times K[\underline{x}]$
Require: G a finite subsets of $K[\underline{x}]$
1: $H := \{\texttt{mult}(\texttt{simplify}(t, f, (H_1, \ldots, H_J), (\widetilde{H}_1, \ldots, \widetilde{H}_J))) \mid (t, f) \in L\}$
2: $done := \mathrm{LM}(H)$
3: **while** $\mathrm{M}(H) \neq done$ **do**
4: let t be an element of $(\mathrm{M}(H) \setminus done)$
5: $done = done \cup \{t\}$
6: **if** there exist $g \in G$ s.t. $\mathrm{LM}(g) \mid t$ **then**
7: choose $g \in G$ s.t. $\mathrm{LM}(g) \mid t$
8: $H := H \cup \left\{\texttt{mult}(\texttt{simplify}(\frac{t}{\mathrm{LM}(g)}, g, (H_1, \ldots, H_J), (\widetilde{H}_1, \ldots, \widetilde{H}_J)))\right\}$
9: **return** H

Chapter 3

Implementation of the F_4 algorithm

Besides the evolution of Gröbner bases algorithms, there have been huge efforts at the implementation level. As stated by Lazard in a recent survey on polynomial system solving [25], "Although it is rather easy to implement Buchberger algorithm crudely, the huge amount of data generated and the length of the computation makes it very difficult to obtain an efficient implementation." In this chapter we describe our implementation of the F_4 algorithm, explaining the challenges faced and the solutions taken.

We start with a detailed example that should help motivate some of the design and implementation decisions. In Section 3.2 we briefly state the requirements of the application. Then, in Section 3.3, we describe the general design, and in section 3.4 we go into details of the most critical data structure, a matrix, and related procedure, row reduction. In Section 3.5 we explain some of the means used to optimize the implementation. C++ code can be found in the Appendices.

3.1 Motivational example

In order to motivate our design we present a toy example that illustrates the operation of the F_4 algorithm and gives us insight of the most time consuming processes. Let us assume that our base field is $GF(2^8)$, that a is a primitive elements of the field and that we want to compute a Gröbner basis for the set F given by

$$\{f_1 = a^{100}x_1^2 + a^{226}x_1x_2 + a^{214}x_1x_3 + a^7x_2^2 + a^{208}x_2x_3 + a^{206}x_3^2 + a^{33}x_1 + a^{32}x_2 + a^{202}x_3 + a^{249},$$

$$f_2 = a^{80}x_1^2 + a^{101}x_1x_2 + a^{137}x_1x_3 + a^{165}x_2^2 + a^{177}x_2x_3 + a^{202}x_3^2 + a^{229}x_1 + a^{107}x_2 + a^{249}x_3 + a^{91},$$

$$f_3 = x_1^2 + a^{11}x_1x_2 + a^{67}x_1x_3 + a^{233}x_2^2 + a^{53}x_2x_3 + a^{58}x_3^2 + a^{244}x_1 + a^{230}x_2 + a^{50}x_3 + a^{50}\}.$$

We follow the steps of a call to F_4 as specified by Algorithm 2.5. The first call to `update` inserts f_1 to G; the second one inserts f_2 to G, the pair $\{f_1, f_2\}$ to B and removes f_1 form G because $\mathrm{LM}(f_2) \mid \mathrm{LM}(f_1)$; the third inserts f_3 to G, the pair $\{f_2, f_3\}$ to B and removes f_2 form G because $\mathrm{LM}(f_3) \mid \mathrm{LM}(f_2)$. Thus at the beginning of the while loop $G = \{f_3\}$ and $B = \{\{f_1, f_2\}, \{f_2, f_3\}\}$.

It is important to note at this point that polynomials that have been removed from G may still be part of some pairs in B, thus in an implementation, we must keep this elements stored somewhere.

Since the degree of both pairs is 2, in the first pass through the while loop $B^* = B = \{\{f_1, f_2\}, \{f_2, f_3\}\}$, and $L = \{(1, f_1), (1, f_2), (1, f_3)\}$. Note that f_2 is part of 2 pairs in B^* but in an implementation we need a mechanism to avoid duplications in L. After `symb_pre_proc` $H_1 = H = \{f_1, f_2, f_3\}$ because no leading monomial of G divides any non-leading monomial of L.

The most natural way to find a row echelon form of H is to use a matrix that represents H where the coefficients are the entries of the matrix, rows correspond to polynomials and columns corre-

spond to monomials, ordered according to the given monomial order. The matrix corresponding to H is

$$\begin{pmatrix} a^{100} & a^{226} & a^{214} & a^{7} & a^{208} & a^{206} & a^{33} & a^{32} & a^{202} & a^{249} \\ a^{80} & a^{101} & a^{137} & a^{165} & a^{177} & a^{202} & a^{229} & a^{107} & a^{249} & a^{91} \\ 1 & a^{11} & a^{67} & a^{233} & a^{53} & a^{58} & a^{244} & a^{230} & a^{50} & a^{50} \end{pmatrix}$$

whose row echelon form is

$$\begin{pmatrix} a^{100} & 0 & 0 & a^{4} & a^{46} & a^{227} & a^{160} & a^{102} & a^{43} & a^{143} \\ 0 & a^{94} & 0 & a^{54} & a^{114} & a^{33} & a^{227} & a^{84} & a^{69} & a^{110} \\ 0 & 0 & a^{246} & a^{120} & a^{32} & a^{180} & a^{159} & a^{137} & a^{198} & a^{91} \end{pmatrix}$$

which corresponds to the set of polynomials

$$\begin{aligned} \widetilde{H}_1 = \widetilde{H} = \{ & f_4 = a^{100}x_1^2 + a^4x_2^2 + a^{46}x_2x_3 + a^{227}x_3^2 + a^{160}x_1 + a^{102}x_2 + a^{43}x_3 + a^{143}, \\ & f_5 = a^{94}x_1x_2 + a^{54}x_2^2 + a^{114}x_2x_3 + a^{33}x_3^2 + a^{227}x_1 + a^{84}x_2 + a^{69}x_3 + a^{110}, \\ & f_6 = a^{246}x_1x_3 + a^{120}x_2^2 + a^{32}x_2x_3 + a^{180}x_3^2 + a^{159}x_1 + a^{137}x_2 + a^{198}x_3 + a^{91}\}. \end{aligned}$$

Since $\mathrm{LM}(f_4) = \mathrm{LM}(f_1) \in \mathrm{LM}(H)$, $\widetilde{H}^+ = \{f_5, f_6\}$. Next, `update`$(G, B, f_5)$ is called, which inserts f_5 to G and the pair $\{f_3, f_5\}$ to B, and subsequently `update`(G, B, f_6) is called, which inserts f_6 to G and the pairs $\{f_3, f_6\}$, $\{f_5, f_6\}$ to B. At the end of this $G = \{f_3, f_5, f_6\}$ and $B = \{\{f_3, f_5\}, \{f_3, f_6\}, \{f_5, f_6\}\}$. It is important to note that no pairs are formed with elements that have been removed form the set G in previous steps.

The condition of the while loop is satisfied, thus we choose all pairs of the minimum degree, in this case all pairs are of degree 3, hence $B^* = B$ and

$$L = \{(x_2, f_3), (x_1, f_5), (x_3, f_3), (x_1, f_6), (x_3, f_5), (x_2, f_6)\}.$$

In the calls to `simplify` for each of the elements of L in the first step of `symb_pre_proc`, (x_2, f_3) is replaced by (x_2, f_4) and (x_3, f_3) by (x_3, f_4). Note that the logic behind these replacements is that f_4 is a reduced representative of f_3 via row-echelon form, so it is more efficient to use it instead.

Through the `symb_pre_proc` loop, the following elements are inserted to H

$$x_2 f_5 = a^{94} x_1 x_2^2 + \dots$$

$$x_3 f_6 = a^{246} x_1 x_3^2 + \dots$$

$$f_4 = a^{100} x_1^2 + \dots$$

$$f_5 = a^{94} x_1 x_2 + \dots$$

$$f_6 = a^{246} x_1 x_3 + \dots$$

So $H = H_2 = \{x_2 f_4, x_1 f_5, x_3 f_4, x_1 f_6, x_3 f_5, x_2 f_6, x_2 f_5, x_3 f_6, f_4, f_5, f_6\}$ and the corresponding matrix is

	$x_1^2x_2$	$x_1^2x_3$	$x_1x_2^2$	$x_1x_2x_3$	$x_1x_3^2$	x_2^3	$x_2^2x_3$	$x_2x_3^2$	x_3^3	x_1^2	x_1x_2	x_1x_3	x_2^2	$\cdots$
x_2f_4	a^{100}	0	0	0	0	a^4	a^{46}	a^{227}	0	0	a^{160}	0	a^{102}	$\cdots$
x_1f_5	a^{94}	0	a^{54}	a^{114}	a^{33}	0	0	0	0	a^{227}	a^{84}	a^{69}	0	$\cdots$
x_3f_4	0	a^{100}	0	0	0	0	a^4	a^{46}	a^{227}	0	0	a^{160}	0	$\cdots$
x_1f_6	0	a^{246}	a^{120}	a^{32}	a^{180}	0	0	0	0	a^{159}	a^{137}	a^{198}	0	$\cdots$
x_3f_5	0	0	0	a^{94}	0	0	a^{54}	a^{114}	a^{33}	0	0	a^{227}	0	$\cdots$
x_2f_6	0	0	0	a^{246}	0	a^{120}	a^{32}	a^{180}	0	0	a^{159}	0	a^{137}	$\cdots$
x_2f_5	0	0	a^{94}	0	0	a^{54}	a^{114}	a^{33}	0	0	a^{227}	0	a^{84}	$\cdots$
x_3f_6	0	0	0	0	a^{246}	0	a^{120}	a^{32}	a^{180}	0	0	a^{159}	0	$\cdots$
f_4	0	0	0	0	0	0	0	0	0	a^{100}	0	0	a^4	$\cdots$
f_5	0	0	0	0	0	0	0	0	0	0	a^{94}	0	a^{54}	$\cdots$
f_6	0	0	0	0	0	0	0	0	0	0	0	a^{246}	a^{120}	$\cdots$

This matrix illustrates the behavior of `symb_pre_proc` from a matrix perspective. The first 6 rows correspond to the 6 polynomials of L which correspond to the 3 pairs in B^*. Columns 1,2 and 4 are done because of the leading monomials of the elements in L. Next the algorithm traverses every column not done searching for a leading monomial in G that divides the corresponding monomial.

For example, column 3 corresponds to the monomial $x_1x_2^2$, is not done and $\mathrm{LM}(f_5) = x_1x_2 \mid x_1x_2^2$, thus x_2f_5 is inserted. Similarly, x_3f_6 is inserted to reduce the column corresponding to $x_1x_3^2$, and so on.

Note that there are many zeros in the matrix, and as the degree gets higher the matrices produced are sparser and sparser. Also note that the matrix is close to being in row-echelon form. After row-echelon form, we obtain

$$\begin{aligned}
\widetilde{H}_2 = \widetilde{H} = \{&f_7 = a^{100}x_1^2x_2 + \cdots, \quad f_8 = a^{75}x_2^2x_3 + \cdots, \quad f_9 = a^{100}x_1^2x_3 + \cdots, \\
&f_{10} = a^{142}x_2x_3^2 + \cdots, \quad f_{11} = a^{223}x_2^3 + \cdots, \quad f_{12} = a^{246}x_1x_2x_3 + \cdots, \\
&f_{13} = a^{94}x_1x_2^2 + \cdots, \quad f_{14} = a^{246}x_1x_3^2 + \cdots, \quad f_{15} = a^{100}x_1^2 + \cdots, \\
&f_{16} = a^{94}x_1x_2 + \cdots, \quad f_{17} = a^{246}x_1x_3 + \cdots\}
\end{aligned}$$

and after eliminating redundant elements, $\widetilde{H}^+ = \{f_8, f_{10}, f_{11}\}$. The interaction of the `update` function for the insertion of these new elements is more interesting. First consider the call $\texttt{update}(G, B, f_8)$. We consider the new pairs $\{f_3, f_8\}, \{f_5, f_8\}, \{f_6, f_8\}$. The pair $\{f_5, f_8\}$ is discarded by the first loop because

$$\mathrm{lcm}(\mathrm{LM}(f_8), \mathrm{LM}(f_6)) = x_1x_2^2x_3 = \mathrm{lcm}(\mathrm{LM}(f_9), \mathrm{LM}(f_5)).$$

Also, the pair $\{f_3, f_8\}$ is discarded by the second loop because the leading monomials are disjoint. Then, f_8 is appended to G and the pair $\{f_6, f_8\}$ is appended to B.

Next consider the call $\texttt{update}(G, B, f_{10})$. We consider the new pairs $\{f_3, f_{10}\}, \{f_5, f_{10}\}, \{f_6, f_{10}\}$ and $\{f_8, f_{10}\}$. The pair $\{f_5, f_{10}\}$ is discarded by the first loop because

$$\mathrm{lcm}(\mathrm{LM}(f_{10}), \mathrm{LM}(f_6)) = x_1x_2x_3^2 = \mathrm{lcm}(\mathrm{LM}(f_{10}), \mathrm{LM}(f_5)).$$

Also, the pair $\{f_3, f_{10}\}$ is discarded by the second loop because the leading monomials are disjoint. Then, f_{10} is appended to G and the pairs $\{f_6, f_{10}\}, \{f_8, f_{10}\}$ to B.

Next consider the call `update`(G, B, f_{11}). We consider the new pairs $\{f_3, f_{11}\}$, $\{f_5, f_{11}\}$, $\{f_6, f_{11}\}$, $\{f_8, f_{11}\}$ and $\{f_{10}, f_{11}\}$. The pair $\{f_{10}, f_{11}\}$ is discarded by the first loop because

$$\mathrm{lcm}(\mathrm{LM}(f_{11}), \mathrm{LM}(f_8)) = x_2^3 x_3 \mid x_2^3 x_3^2 = \mathrm{lcm}(\mathrm{LM}(f_{11}), \mathrm{LM}(f_{10})).$$

Also, the pairs $\{f_3, f_{11}\}$ and $\{f_6, f_{11}\}$ are discarded by the second loop because the leading monomials are disjoint. Then, f_{11} is appended to G and the pairs $\{f_5, f_{11}\}, \{f_8, f_{11}\}$ to B.

After all that

$$G = \{f_3, f_5, f_6, f_8, f_{10}, f_{11}\}$$

and

$$B = \{\{f_6, f_8\}, \{f_6, f_{10}\}, \{f_8, f_{10}\}, \{f_5, f_{11}\}, \{f_8, f_{11}\}\}.$$

In the next round, all pairs in B are of degree 4, so $B^* = B$, and

$$L = \{(x_2^2, f_6), (x_1, f_8), (x_2, x_3, f_6), (x_1, f_{10}), (x_3, f_8), (x_2, f_{10}), (x_2^2, f_5), (x_1, f_{11}), (x_2, f_8), (x_3, f_{11})\}.$$

This monomial-polynomial pairs are simplified respectively to

$$(x_2, f_{12}), (x_1, f_8), (x_2, f_{14}), (x_1, f_{10}), (x_3, f_8), (x_2, f_{10}), (x_2, f_{13}), (x_1, f_{11}), (x_2, f_8), (x_3, f_{11})$$

and then multiplied and appended to H which is then enlarged in the while loop of `symb_pre_proc` with

$$\{x_3 f_{14}, x_3 f_{10}, f_{13}, f_{12}, f_{14}, f_{11}, f_8, f_{10}, f_4, f_5, f_6\}.$$

after row-echelon form we obtain

$$\begin{aligned}
\widetilde{H}_3 = \widetilde{H} = \{ & f_{18} = a^{246}x_1x_2^2x_3+, && f_{19} = a^{192}x_1x_2 + \cdots, && f_{20} = a^{246}x_1x_2x_3^2 + \cdots, \\
& f_{21} = a^{65}x_1x_3 + \cdots, && f_{22} = a^{75}x_2^2x_3^2 + \cdots, && f_{23} = a^{168}x_3^4 + \cdots, \\
& f_{24} = a^{94}x_1x_2^3 + \cdots, && f_{25} = a^{79}x_2^2x_3 + \cdots, && f_{26} = a^{223}x_2^3x_3 + \cdots, \\
& f_{27} = a^{246}x_1x_3^3 + \cdots, && f_{28} = a^{142}x_2x_3^3 + \cdots, && f_{29} = a^{94}x_1x_2^2 + \cdots, \\
& f_{30} = a^{246}x_1x_2x_3, && f_{31} = a^{246}x_1x_3^2 + \cdots, && f_{32} = a^{223}x_2^3 + \cdots, \\
& f_{33} = a^{193}x_2x_3^2 + \cdots, && f_{34} = a^{100}x_1^2 + \cdots \}.
\end{aligned}$$

Only one element makes it to $\widetilde{H}^+$, $f_{23} = a^{168}x_3^4 + \cdots$, and only two pairs make it through the filters of the update function, $\{f_6, f_{23}\}$ and $\{f_{10}, f_{23}\}$. Therefore $G = \{f_3, f_5, f_6, f_8, f_{10}, f_{11}, f_{23}\}$ and $B = \{\{f_6, f_{23}\}, \{f_{10}, f_{23}\}\}$. In the next round, all pairs in B are of degree 5, so $B^* = B$, and after `symb_pre_proc` we obtain

$$H_4 = H = \{x_3f_{27}, x_1f_{23}, x_3f_{28}, x_2f_{23}, f_{27}, f_{28}, f_{23}, f_{13}, f_{12}, f_{14}, f_{11}, f_8, f_{10}, f_4, f_5, f_6\}$$

which is row reduced to

$$\begin{aligned}
\widetilde{H}_4 = \widetilde{H} = \{ & f_{35} = a^{246}x_1x_3^4 + \cdots, && f_{36} = a^{168}x_2x_3^4 + \cdots, && f_{37} = a^{246}x_1x_3^3 + \cdots, \\
& f_{38} = a^{142}x_2x_3^3 + \cdots, && f_{39} = a^{168}x_3^4 + \cdots, && f_{40} = a^{94}x_1x_2^2 + \cdots, \\
& f_{41} = a^{246}x_1x_2x_3 + \cdots, && f_{42} = a^{246}x_1x_3^2 + \cdots, && f_{43} = a^{223}x_2^3 + \cdots, \\
& f_{44} = a^{75}x_2^2x_3 + \cdots, && f_{45} = a^{142}x_2x_3^2 + \cdots, && f_{46} = a^{100}x_1^2 + \cdots, \\
& f_{47} = a^{94}x_1x_2 + \cdots, && f_{48} = a^{246}x_1x_3 + \cdots \}.
\end{aligned}$$

In this case all the leading monomials are present in H thus $\widetilde{H}^+ = \emptyset$ and hence `update` does not

append any new elements to G or B and hence $B = \emptyset$ when the condition for the while loop is checked and the algorithms terminates returning $G = \{f_3, f_5, f_6, f_8, f_{10}, f_{11}, f_{23}\}$.

3.2 Requirements

We aim at an application that computes a Gröbner basis for the ideal generated by a set of polynomials over a field, using the F_4 algorithm, as specified in Chapter 2. Efficiency is the most important goal, both in time and in memory usage.

Flexibility and maintainability are also important. The number of variables must be a parameter that can be modified. We aim at Galois Field 256 to be the base field, and the graded lexicographic monomial order, however, both base field and monomial order, should be easy to change. It should also be possible to change the selection strategy for pairs and the row echelon form algorithm.

3.3 Design

The design of this application must favor time and memory efficiency without losing maintainability. The F_4 algorithm is big enough to require the use of high level design, yet a naive design may yield speed and memory less than acceptable. We explain in this section the basic structures used in our design, hoping to provide insight on the decisions taken.

The Class Diagram in Figure 3.1 provides an overview of the general design of the application. Class `Field` provides the functionality of the base field, Class `Matrix` provides the functionality of a Matrix of base field values, and Class `Monomial` provides the functionality of a monomial in n variables. Class `List_poly` provides the functionality of a list of polynomials. Classes `Signature` and

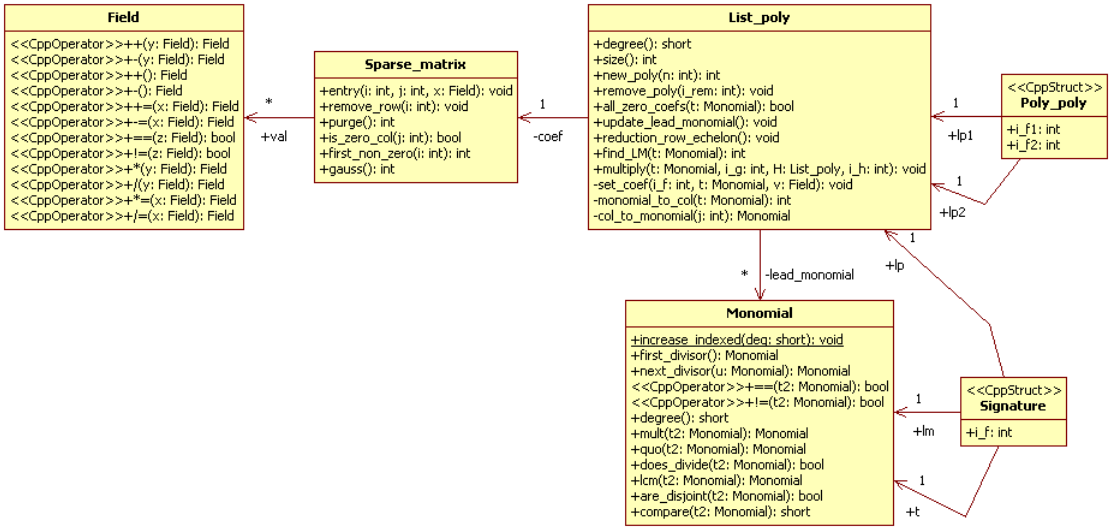

Figure 3.1: Class Diagram.

`Poly_poly` are simple structures to hold pairs of monomial-polynomial and polynomial-polynomial respectably.

In the following sub-sections we discuss the details of each of these classes and their interaction.

3.3.1 List of Polynomials

The example of Section 3.1 illustrates the waste of space in writing the monomials of each polynomial over and over again. It also shows the waste of space in writing zeros in the matrices that represent polynomials. In order to face this two issues, a list of polynomials is stored as a matrix of its coefficients using a sparse representation. Each row corresponds to a polynomial and each column to a monomial. The correspondence between columns and monomials is given by member functions `column_to_monomial` and `monomial_to_column`.

The example of Section 3.1 also shows that the leading monomials are accessed repeatedly so a

vector of leading monomials is maintained for quick access. Public member functions include:

- `new_poly(n: int): int`. Inserts `n` zero polynomials.
- `remove_poly(i: int)`. Removes the `i`-th polynomial.
- `all_zero_coefs(t: Monomial): bool`. Returns `true` if for all polynomials, the coefficient of `t` is zero, or `false` otherwise.
- `update_leading_monomial()`. Updates the leading monomials.
- `reduction_row_echelon()`. Reduces this list of polynomails to row echelon form.
- `find_LM(t: Monomial): int`. Searches for a polynomial with leading monomial `t` and returns its index or -1 if not found.
- `multiply(t: Monomial, i_g: int, H: List_poly, i_h: int)`. Multiplies the polynomial in `i_g` position of this List, by the monomial `t` and stores the result in the `i_h` position of the list `H`.
- `set_coef(i: int, t: Monomial, v: Field)`. Sets the coefficient of `t` in the `i`-th polynomial to `v`.

Our C++ declaration for this Class can be found in Appendix A.

3.3.2 Monomial

In order to allow fast conversion from monomials to columns and vice versa, without redundant storage, we designed a Monomial with two interchangeable representations. one as a map from

Degree	0	1			2						3									
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Monomial	1	x_3	x_2	x_1	x_3^2	x_2x_3	x_2^2	x_1x_3	x_1x_2	x_1^2	x_3^3	$x_2x_3^2$	$x_2^2x_3$	x_2^3	$x_1x_3^2$	$x_1x_2x_3$	$x_1x_2^2$	$x_1^2x_3$	$x_1^2x_2$	x_1^3
$\times x_1$	3	7	8	9	14	15	16	17	18	19										
$\times x_2$	2	5	6	8	11	12	13	15	16	18										
$\times x_3$	1	4	5	7	10	11	12	14	15	17										

Table 3.1: Monomials in 3 variables up to degree 3 indexed according to graded lexicographic order and corresponding multiplication table. Row 1 shows the degree of the monomials. Row 2 has the indexes and row 3 the monomials sorted in increasing graded lexicographic order. Rows 4 to 6 constitute the multiplication table. Its entries should be interpreted as the index of the monomial that results of multiplying the given monomial in the column with the variable in the row.

variables to degrees, e.g. $x_4^2x_7x_9^5$ is stored as $\{4 \mapsto 2, 7 \mapsto 1, 9 \mapsto 5\}$, and another as an integer. All monomials up to certain degree are stored sorted according to the given monomial order in a static vector, and the integer representation is simply an index to that vector. The main advantage of using the integer representation has to do with the design of a list of polynomials. The integer representation of monomials allows fast conversion from monomials to columns and vice versa. It is also fast to compare monomials according to the monomial order and easy to change the monomial order.

As it was illustrated in the example of Section 3.1, multiplication of monomials is a very recurrent operation. In order to make multiplication as fast for the integer representation as for the map representation, we use a multiplication table. Table 3.1 exemplifies monomials indexed according to graded lexicographic order and the corresponding multiplication table.

The problem with the integer representation is the obvious bound on the degree of the monomials that requires extra care from the part of the programmer. A function call to the static function `increase_indexed(d: int)` changes the degree up to which monomials are indexed to `d`, and updates the multiplication table. The user of the `Monomial` class must be aware at all times of this bound, or otherwise, should implement expensive checks. The normal strategy for selecting

pairs in the F_4 algorithm (see section 2.2) favors the awareness of a degree bound.

Changing from map to integer representation for a monomial has worse case complexity log of the number of monomials indexed (which is approximately `d`), while changing from index to map takes constant time. Public members of the class `Monomial` include:

- `first_divisor(): Monomial`. Returns the smallest divisor according to the monomial order.
- `next_divisor(t: Monomial): Monomial`. Given that `t` is a divisor of this monomial, returns the smallest divisor of this monomial larger than `t`, according to the monomial order.
- `operator==(t: Monomial): bool`. Returns `true` if equal to `t`, and `false` otherwise.
- `operator!=(t: Monomial): bool`. Returns `false` if equal to `t`, and `true` otherwise..
- `degree(): int`. Returns the degree of the monomial.
- `mult(t: Monomial): Monomial`. Returns the product of this Monomial with `t`.
- `quo(t: Monomial): Monomial`. Returns the quotient of this Monomial with `t`. It is assumed that `t` divides this monomial.
- `does_divide(t: Monomial): bool`. Returns `true` if `t` divides this monomial and `false` otherwise.
- `lcm(t: Monomial): Monomial`. Returns the least common multiple between this monomial and `t`.
- `are_disjoint(t: Monomial): bool`. Returns `true` if `t` and this monomial and are disjoint and `false` otherwise.

Some of these functions should have independent implementations for the two representations, others can have a single implementation for the map representation and use the constant time conversion from integer to map representation. Our C++ declaration for this Class can be found in Appendix A.

3.3.3 Matrix

The `Matrix` Class provides the functionality of a typical matrix of field elements. Although we defer a detailed discussion to Section 3.4, here we present the basic blue print of what a `Matrix` class should provide. The basic member functions are.

- `entry(i: int, j: int, Field x)`. Sets the `(i,j)` entry to `x`.
- `remove_row(i: int)`. Sets all entries in row `i` to zero.
- `purge()`. Removes redundancy of the matrix.
- `is_zero_col(j: int): bool`. Returns `true` if for all entries in column `j` are zero, or `false` otherwise.
- `first_non_zero(i: int): int`. Returns the column of the first non-zero entry of row `i`.
- `echelon(): int`. Transforms the matrix to a row echelon form, and returns the rank of the matrix.

Changing the row echelon reduction algorithm is as simple as changing a member function of the class, and changing the storage scheme requires the replacement of the `Matrix` class.

3.3.4 Base Field

The base field is the Galois field with $2^8 = 256$ elements. Its implementation was adapted from an implementation by Schmidt in an algebraic processor known as POLYPAK [31]. The class provides basic operations such as addition, multiplication, subtraction and division. A change in the base field can be easily accomplished by simply substituting this class.

3.3.5 Signature and Poly_poly

`Signature` and `Poly_poly` are simple data holder structures. `Poly_poly` is used to store a pair of polynomials, which are fundamental in the F_4 algorithm as was shown in Section 2. In order to avoid the burden of redundancy, pointers to existing polynomials are the best choice, rather than actual copies. It may seem tempting to store the least common multiple of the leading monomials or some other information that is used often, however our choice was to keep pairs as light as possible. Instead, upon creation of a pair, we compute its degree and store pairs according to their degree. Because of the normal strategy for F_4 (Section 2.2), the degree is the only information required before the pair is chosen for processing in line 7 of Algorithm 2.5.

The class `Signature` is used to store a pair of a monomial and a polynomial, which becomes important with the `simplify` algorithm –2.4–. Besides a pointer to a polynomial p and a monomial t it contains the monomial s that results of multiplying the leading monomial of p with the monomial t, $s = \mathrm{LM}(p)t$. This decision was taken because the creation of a signature often computes s even before t, thus it would be a waste of time to compute it again later. Moreover, signatures are temporary objects, so that it is not a burden to store this redundant information.

3.4 Echelon Form and Matrix Storage Scheme

A matrix is the main storage structure in our implementation, thus it is paramount to store it wisely. Moreover, finding a row echelon form is the most time consuming part of the F_4 algorithm, and memory is a big limitation in computation of large Gröbner bases. So we need a storage scheme that is memory efficient yet allows us to find row echelon forms very fast.

We tried to replicate what was suggested by Faugère in [15], however, this was not an easy task since many details are left out of the explanation. After several attempts we decided for a straight forward Gaussian elimination using a simplified Markowitz criteria for choosing pivots, and we use a coordinate scheme as data structure with access by rows and columns via simply linked lists. The main considerations that lead us to make this decisions were:

- As we saw in Section 3.1, matrices tend to get bigger and bigger, yet sparser and sparser, as the F_4 algorithm progress.
- The sparsity of a matrix depends on the sparsity of previous matrices, so ultimately it depends on our ability to preserve sparsity in the reduction process.
- The shape of the matrices that appear in the process are by no means random, they tend to have a very specific shape exemplified in Figure 3.2.
- Strictly speaking, column swapping is not allowed, although some tricks are possible.
- The base field is of medium size, storing a single field element takes one Byte.

In the next two sub-section we describe the Algorithm used to compute row echelon forms and the Data structure used to hold matrices.

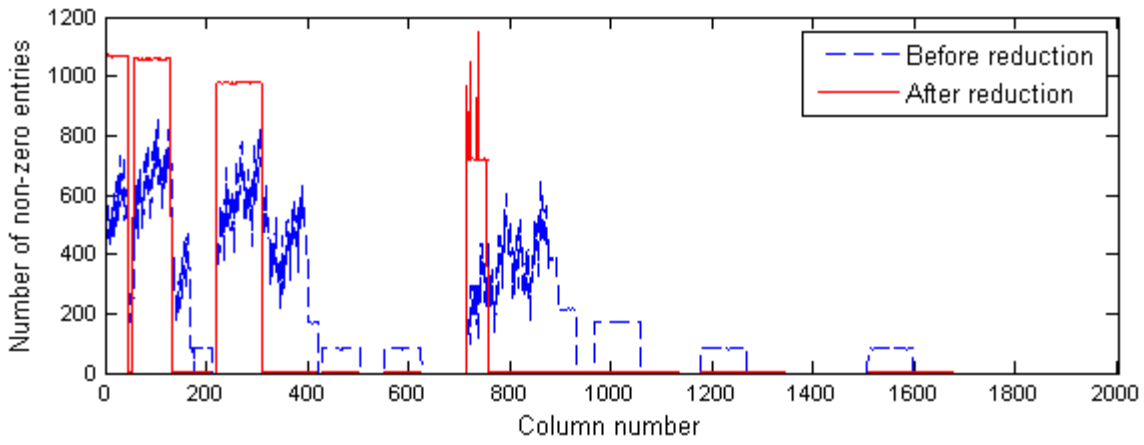

Figure 3.2: Number of non-zero entries for each column in a run of the F_4 algorithm. The example is a random system of 10 equations in 9 variables. The graph deploys the matrix of the fourth pass through the while loop, before and after row reduction.

3.4.1 Echelonization Algorithm

The problem of efficient computation of row echelon forms for a sparse matrix has been widely studied. In a very general setting, Duff [13], or Davis [10] have researched in the area for decades. For finite fields, a survey by LaMacchia and Odlyzko [22], provides a general perspective of the different alternatives, advantages and disadvantages. Unfortunately, most methods aim at solving linear systems, while our goal is to find a row echelon form of a matrix. Although the two problems are related, some adaptation is necessary.

For example, in [15] Section 3.1, Faugère suggests to "first extract a square sub matrix and to put the remaining columns into the right-hand side." The statement is less than precise and leaves some room for interpretation. We interpret it as, given a $m \times n$ matrix M of rank r, if $r = n < m$ choose any r linearly independent rows of M to form a square non-singular matrix; if $r = m < n$,

choose the leftmost r linearly independent columns of M. Then, this first step can be as hard as finding the row echelon form. In order to overcome this, in [15] Section 3.1, Faugère suggests to use a fast mod p algorithm. However, in our case, since the base field is GF(256) the field operations take constant time and there is no significant advantage in doing that.

[15] also suggests structured Gaussian elimination, a method explained in [22, 30] as an adaptation of standard techniques used to minimize fill-ins during Gaussian elimination to be applied to matrices arising from integer factorization. Based on Duff's classification of ordering strategies for controlling fill-ins (see [13] Chapters 7 and 8), Structured Gaussian Elimination is a combination of a simplified Markowitz criterion with some kind of ordered objective form. We found direct application of this method unfeasible because of the column interchanges implied.

We used a simplified version of structured Gaussian elimination, or to be more precise we used a simplified Markowitz criterion and a simple objective form. Columns are split into two sets, light columns and dense ones. The next pivot is chosen in the left-most non-zero column. The row for the pivot is chosen to be one with minimum number of non-zero entries among the light columns. This pivoting strategy aims at minimizing the number of fill-ins in the light columns, while guarantees a row echelon form upon termination.

For example, suppose that the following matrix is left after some steps of Gaussian elimination.

$$\left|\begin{array}{cccccccccccccccccc|l}
* & * & 0 & 0 & 0 & * & * & * & * & * & * & 0 & 0 & 0 & 0 & * & * & * & * & * & \leftarrow 2 \\
* & * & * & 0 & 0 & 0 & * & * & * & * & 0 & 0 & 0 & * & 0 & * & * & * & * & * & \leftarrow 2 \\
* & * & 0 & 0 & 0 & * & * & * & * & * & 0 & 0 & 0 & 0 & * & * & * & * & * & * & \leftarrow 2 \\
* & * & 0 & 0 & 0 & 0 & * & * & * & * & * & 0 & * & 0 & 0 & * & * & * & * & * & \leftarrow 2 \\
* & * & 0 & 0 & * & 0 & * & * & * & * & 0 & 0 & 0 & 0 & 0 & * & * & * & * & * & \leftarrow 1 \\
* & * & * & * & 0 & 0 & * & * & * & * & 0 & 0 & 0 & 0 & 0 & * & * & * & * & * & \leftarrow 2 \\
* & * & 0 & 0 & 0 & 0 & * & * & * & * & 0 & 0 & * & 0 & 0 & * & * & * & * & * & \leftarrow 1 \\
* & * & 0 & * & 0 & 0 & * & * & * & * & 0 & 0 & 0 & 0 & * & * & * & * & * & * & \leftarrow 2 \\
* & * & 0 & 0 & 0 & 0 & * & * & * & * & 0 & 0 & 0 & 0 & 0 & * & * & * & * & * & \leftarrow 0 \\
* & * & 0 & 0 & 0 & 0 & * & * & * & * & 0 & * & 0 & 0 & 0 & * & * & * & * & * & \leftarrow 1
\end{array}\right|$$

Suppose that columns 3,4,5,6,11,12,13, 14 and 15 are declared light while the others are declared heavy. The next pivot is chosen among the non-zero entries of the first column. The row is chosen to be one with the least number of nonzero entries among the light columns. At the right hand side of the matrix, we indicate this counts, so the next pivot would be chosen to be in row 9 column 1.

We compute row reduce echelon forms. Experimentally, we found that this was faster in the long run than computing just any row echelon form.

3.4.2 Data Structure

The data structure used to hold matrices is based on a description in [13]. A matrix is stored as an unordered set of triplets (a_{ij}, i, j). For any given entry there is a row link (and a column link) that connects the entry to another entry in the same row (and column), allowing traversal of the row (and the column) in an arbitrary but fixed order. A vector specifies the first entry of each row and each column. As an example, the matrix

$$\begin{pmatrix} a^{100} & a^{226} & 0 & 0 \\ a^{80} & 0 & 0 & 0 \\ a^{2} & 0 & a^{67} & a^{233} \end{pmatrix} \tag{3.1}$$

can be stored as shown in Table 3.2. If we want to traverse the first column of the matrix, we look at the first j-col-start entry, 3, which points at to the entry a^{80} in row 2 column 1. Then we look at link-col[3]=6 which points at the entry a^2 in row 3 column 1, which has a corresponding link-col[6]=2 which points at the entry a^{100} in row 1 column 1. We know that there are no more entries in this column because link-col[2]=-1 meaning the end of the list.

Index	1	2	3	4	5	6
Value	a^{67}	a^{100}	a^{80}	a^{233}	a^{226}	a^2
i	3	1	2	3	1	3
j	3	1	1	4	2	1
link-row	-1	-1	-1	1	2	4
link-col	-1	-1	6	-1	-1	2
i-Row-Start	5	3	6			
j-Col-Start	3	5	1	4		

Table 3.2: Matrix of Equation (3.1) stored using a coordinate scheme, with rows and columns simply linked.

3.5 Optimization

Although a good design and algorithms are fundamental to achieve efficiency, it is also extremely important to pay close attention to the details of the implementation. In the case of our implementation of F_4, depuration of the code accounted for up to a factor of 100 reduction in time and memory consumption in some subroutines.

We do not pretend to provide a comprehensive study of code optimization, but rather to relate our experience optimizing our C++ implementation of the F_4 algorithm, hoping to provide some insight of the process.

Perhaps the most rewarding optimizations are done by a good compiler, so they require no work from yourself. However, there were some instances where we did have to get our hands dirty. For example, during Gaussian elimination as described in Section 3.4.1 it was very important to store dense columns in a dense representation by rows, in order to favor data alignment when adding rows among those columns. Also, it is important to store the counts of non-zero entries for each row, and to update it during row addition.

Most of the memory consumption comes from storing matrices. A key for saving memory is

to analyze where the peek consumption happens. For example, at some stage of our code, the peek consumption was happening at the point where we stored a copy of the already reduced matrix, because we were creating a new copy while keeping the old matrix and other unnecessary information from previous steps. By releasing storage before this step, and storing a light version of the matrix we were able to move the peek to another point in the execution. A light version of the matrix was possible because after reduction, the matrix does not require much of the information that helps speed up Gaussian elimination, like links across the columns and the row coordinate of each entry.

In order to save memory, without affecting the efficiency of Gaussian elimination, we keep a vector of deleted elements from the coordinate scheme, that we use when fill-ins occur. Of course, the light copy of the matrix is completely compact and has no holes in the vector of non-zero entries.

Other optimization include the use of ansi-c arrays instead of std vectors, the use of a boolean vector to distinguish $\widetilde{H}$ from $\widetilde{H}^+$ and the elements deleted from the basis by the `update`. We also tried to avoid trivial checks like division by zero when unnecessary.

Chapter 4

Experimental Results

In this chapter we present experimental data from our C++ implementation of F_4. First, we present a raw comparisons to other Gröbner basis implementations. Magma's implementation of F_4 is widely recognized as one of the best, so we present a more detailed comparison with it. Finally, by means of an example, we discuss the time consumption of different parts of the F_4 algorithm.

We have limited our examples to sets of m degree 2 polynomials in n variables with coefficients chosen randomly in GF(2^8). They provide a general perspective but do not constitute a comprehensive study. Higher degree polynomials can be transformed into degree 2 polynomials by adding additional variables. Also, for fixed m and n, the behavior of Gröbner basis algorithms on such a system is extremely stable, thus we can regard one instance as a generic sample of the family of sets of m degree 2 polynomials in n variables.

All Gröbner bases were computed in graded lexicographic order. All the experiments were run in a Dell Inspiron 530 Intel(R) core(TM)2 Duo CPU E6550 @2.33GHz with 2GB of Ram running Windows XP.

m	n	F_4 by Magma	Singular	Macaulay2	F_4 by Cabarcas
10	7	0.015	0.03	0.187	0.078
9	7	0.015	0.09	0.344	0.094
8	7	0.046	0.28	0.86	0.266
7	7	0.203	1.22	5.344	0.891
11	8	0.031	0.27	1.312	0.25
10	8	0.125	0.35	3.187	0.719
9	8	0.343	1.34	10.078	1.86
8	8	1.625	12.89	83.781	5.469

Table 4.1: Comparison between different algorithms to compute Gröbner bases.

4.1 Comparison with Other Software Systems

Table 4.1 shows the time it took for different software systems to compute some Gröbner bases.

Magma [3] is a commercial proprietary software system for computational algebra developed by the Computational Algebra Group at the University of Sydney. Its development goes back to the early 1970's when the group developed the Cayley system for group theory. A major upgrade of Cayley lead to the first version of Magma at the end of 1993 which by 1995 was running at approximately 200 sites in about 35 countries, and has continue to grow since then. Magma has an efficient implementation of F_4 since the mid 2000's. We used version 2.14-9 of Magma and we computed Gröbner baseswith flags `Al:="Direct", HomogeneousWeights := false, ReduceByNew := false`. This options were chosen in order to resemble the original algorithm as much as possible.

Singular [21] is an open source and free Computer Algebra System for polynomial computations, developed by the SINGULAR team at the Department of Mathematics of the University of Kaiserslautern. Singular descended from the Buchmora system developed at the end of the 1980's, and continues to grow and provide support today. Singular does not include an implementation of F_4. We run version 3-1-1 on a Cygwin shell and we computed Gröbner bases with the `std` option,

which uses a "(more or less) straight-forward implementation of the classical Buchberger (resp. Mora) algorithm" according to its manual.

Macaulay2 [20] is a free proprietary software system devoted to supporting research in algebraic geometry and commutative algebra, whose creation has been funded by the National Science Foundation since 1992. It includes an implementation of the Buchberger algorithm for computing Gröbner bases. We run version 1.3.1 on a Cygwin shell.

4.1.1 Close comparison with Magma

Magma posses one of the only implementations of F_4. We compare our implementation with Magma's. Using the Verbose option of Magma set to `("Faugere",3)` we were able to obtain details about intermediate steps.

Table 4.2 shows intermediate information for a run example of Magma and the corresponding values for our implementation. The example is a set of 11 degree 2 polynomials in 10 variables, with coefficients chosen randomly in GF(256). For each pass through the while loop of the algorithm, we report

- *Basis Length:* Number of polynomials in the basis G.
- *Queue Length:* Number of unprocessed pairs in B.
- *Degree:* Degree of the chosen pairs in B^*.
- *No. pairs:* Number of pairs selected to be processed in B^*.
- *Rows:* Number of polynomials in H before row reduction.

Step	Basis Length	Queue Length	Degree	No. pairs	Rows	Columns	nnz	New polys
F_4 by Magma								
1	11	10	2	10	11	66	724	10
2	21	46	3	46	120	285	6798	46
3	67	291	4	291	761	916	97946	116
4	183	864	5	864	2356	2149	549796	195
5	378	1573	6	1573	5039	4145	1563930	217
6	595	1832	7	1832	7856	6486	2607470	132
7	727	1320	6	1320	4570	3580	1329708	165
8	892	1650	5	1650	3399	1914	671062	110
9	1002	1100	4	1100	1863	818	175740	44
10	1046	440	3	440	703	274	23088	10
11	1056	55	2	55	120	66	1282	1
F_4 by Cabarcas								
1	1	10	2	10	11	66	724	10
2	11	46	3	46	119	284	6641	46
3	57	291	4	291	760	915	124546	116
4	173	864	5	864	2325	2118	747516	195
5	368	1573	6	1573	4389	3495	1941988	217
6	585	1832	7	1832	5376	4006	2396985	132
7	500	1320	6	1320	3586	2596	1380563	165
8	338	1650	5	1650	3211	1726	844075	110
9	167	1100	4	1100	1859	814	235458	44
10	55	440	3	440	703	274	28778	10
11	10	55	2	55	120	66	1272	1

Table 4.2: Detailed comparison between F_4 by Magma and F_4 by Cabarcas.

- *Columns:* Number of monomials in the polynomials of H before row reduction.
- *nnz:* Number of non-zero entries in a matrix representation of H before row reduction.
- *New polys:* Number of new polynomials after row reduction.

All columns are exactly the same except for Basis length, the number of rows and columns and the number of non-zero entries. Basis length values are different because they measure different sets, while Magma's verbose counts all elements in the basis including those marked as unnecessary to make pairs, we only report necessary elements. The difference in rows, columns and nnz, on the other hand, are due to the distinct row reduction functions used, and due to the choices made in the `simplify` function.

Next, we push both Magma's and our own implementation to the limit. In Table 4.3 we report

		F_4 by Magma		F_4 by Cabarcas	
m	**n**	Time	Memory	Time	Memory
12	9	0.359	8.4	1.453	8.172
11	9	0.531	8.7	2.422	11.11
10	9	2.078	12.3	8.563	24.28
9	9	15.14	37.4	33.515	128.9
13	10	1.593	11.2	6.985	20.22
12	10	5.078	19.1	17.672	44.3
11	10	18.656	37.1	62.219	98.56
10	10	133.812	152.5	293.375	564.1
14	11	13.796	32.2	43.344	79.54
13	11	29.14	41.3	105.782	142.5
12	11	126.593	100.2	426	353.2

Table 4.3: Time and memory comparison between F_4 by Magma and F_4 by Cabarcas.

a comparison on the running time and memory usage for both algorithms and many combinations of m and n.

Our implementation of F_4 is between 2 and 4 times slower and it consumes between 1 and 4 times more memory than Magma's implementation.

4.1.2 Inside F_4

In this section we analyze how the time is spent in different parts of the F_4 algorithm. The information comes from our own implementation. Table 4.4 reports the amount of time of different parts of the algorithm at each step for the same sample run as in the previous section, 11 polynomials in 10 variables. The subroutines timed were

- *L-R:* Computation of monomial-polynomial pairs from polynomial pairs. Line 9 in Algorithm 2.5.

- *Symb:* Symbolic preprocessing. Algorithm 2.6.

Step	L-R	Symb	Echelon	Update	Indexing
1	0	0	0	0	0
2	0	0	0.016	0.015	0.016
3	0	0.016	0.406	0.203	0.031
4	0.016	0.156	3.688	1.219	0.109
5	0.016	0.641	13.532	4.094	0.281
6	0.031	1.234	12.797	4.375	0
7	0.031	0.859	3.219	5.25	0
8	0.031	0.469	3.094	3.563	0
9	0.015	0.094	0.562	0.875	0
10	0	0.016	0.031	0.141	0
11	0	0	0	0.015	0
Total	0.14	3.485	37.345	19.75	0.437
% of Total	0.23%	5.70%	61.06%	32.29%	0.71%

Table 4.4: Time spent in different subroutines of the F_4 algorithm for each step in a run.

- *Echelon:* Computation of row echelon form. Line 11 of Algorithm 2.5.
- *Update:* Update function. Algorithm 1.4.
- *Indexing:* Monomial indexation and construction of multiplication table as described in section 3.3.2.

Note that more than 90% of the time is spent in row reducing matrices and updating. In particular, more than 25% of the time is spent row reducing the matrices of steps 5 and 6. All the symbolic computaion in columns L-R and Symb account for less than 6% of the total time. Indexing, which is not part of the F_4 algorithm but an implementation trick, acounts to less than 1% of the total time.

Chapter 5

Conclusions and Future Work

The F_4 algorithm is an excellent tool to compute Gröbner bases. It lies on a solid mathematical ground. Its efficiency relies on fine implementation details.

We have achieved an efficient implementation of the F_4 algorithm. There is a lot of room for improvement, especially in the row reduction algorithm. But the framework for managing monomials, polynomials and pairs is competitive with commercial software.

We believe that the last word about F_4 has not been said yet. There are choices that open the door for improvement. It is crucial, to understand better, what makes the algorithm fast. We believe it is a combination of fine tunning of sparse linear algebra with a symbolic process that promotes sparsity.

We will keep working in improving the F_4 algorithm and its implementation. We hope that this thesis will encourage other researchers to work in this area, where we believe there is enormous potential. Below are some specific points where we believe the F_4 algorithm can be improved.

5.1 Future Work

Major areas of improvement are the `simplify` function which stimulates the sparsity of matrices, the `update` function which purges basis set and the set of pairs, and matrix row reduction which carries most of the computation. We discuss each of this points in what follows.

5.1.1 The `simplify` Function

The `simplify` function –2.4– is very important for the efficiency of the F_4 algorithm, however, very little has been written about it. In [15] references to `simplify` are limited to 2 comments. Before presenting the improved algorithm Faugère writes

> "In the previous version of the algorithm we used only some rows of the reduced matrix (the sets $\widetilde{F}^+$), rejecting the rows which were already in the original matrix (the sets TF). In the new version of the algorithm we keep these useless rows, and we try to replace some products occurring in the rows of the "matrix" F by a new "equivalent" product $m'f'$ with $m \geq m'$. This is the task of the function `simplify`."

and later he remarks

> "Experimental observation establishes that the effect of `simplify` is to return, in 95% of the cases, a product (x_i, p) where x_i is a variable (and frequently the product (x_n, p)). This technique is very similar to the FGLM algorithm for computing normal forms by using matrix multiplications."

showing that the `simplify` function is a heuristic transformation, and not, as the name suggests, a reduction with a measurable goal. Why is a product of the form (x_i, p) desirable? why $i = n$ is better? as implied by the quotations above.

Answering these questions is important to fully understand the power of the F_4 algorithm, to weight its features according to usefulness, and to be able to improve on it. Similar strategic decisions are discussed in [28] and [7] in pondering about *partial enlargement*.

The `simplify` function is also not well defined. If we look back to Algorithm 2.4, different choices of divisor may yield different results. More than a mistake, this is a source for improvement. If we understand better the goal of the `simplify` function, we can take a more educated decision there.

In Section 2.2 (see Improvements (4) and (5)) we mentioned that the two uses of the `simplify` function can be understood differently thus different approaches are possible. Its first use, in line 1 of Algorithm 2.6, is related with the correctness of the algorithm, whereas the second use, in line 8 of Algorithm 2.6, is related with the termination of the algorithm.

Going back to the `basic_symb_pre_proc` algorithm –2.2–, we can see that in line 7, any monomial-polynomial pair that satisfies $\mathrm{LM}(sg) = t$ can be used. Thus we have more leverage than what the `simplify` function provides. We are investigating how to make a more educated decision.

In summary, the `simplify` function is a good heuristic method, however there is room for improvement. Understanding the insight of the choices made by this function may help improve the F_4 algorithm.

5.1.2 The `update` Function

The `update` function is in charge of discarding useless pairs, inserting new elements to the basis, and forming new pairs. The `update` function used in our implementation dates from 1988, and lots

of research has focus on this issue since.

A possible source of improvement is in the detection of redundant pairs. For example, Möller, Mora and Traverso syzygies tracker [29] or Faugère's F_5 algorithm [16], can be adapted to work together with F_4.

Another, is in the insertion of new elements. In [12], Ding et. al. explore a way to keep the set of basis elements G smaller by distinguishing *Mutant Polynomials* [11]. The idea is to avoid forming pairs with larger degree elements when smaller degree elements are available. This idea can also be used to update the `update` function.

This latest idea can work best if combined with a mutant based termination condition for the F_4 algorithm. In [27] Mohamed et. al. introduced a termination criterion based on Mutant polynomials to be used to detect a Gröbner basis in the XL algorithm. A similar approach is likely for the F_4 algorithm.

5.1.3 Linear Algebra

As it was shown in Section 4.1.2, most of the running time of the F_4 algorithm is spent row reducing large sparse matrices. Thus, it is obvious that improvements in the linear algebra would greatly impact the efficiency of the algorithm. In Section 3.4 we briefly discussed some alternative algorithms, but many more are available, spread over different scientific areas where large sparse matrix appear.

Also, the linear algebra module is specially sensitive to fine optimization. In Section 3.4.1 the particular issue of data alignment was mentioned. For such large amount of data, as those handled by the F_4 algorithm, data alignment can reduce by a factor of 10 the running time of the algorithm.

Also, linear algebra techniques have the potential of parallelization. For example, the Wiedemann algorithm for solving systems of equations can be easily parallelized, and even a parallel Gaussian Elimination may yield some gain. This is specially important, at a time when the computing industry, at large, is switching to parallel architectures.

Bibliography

[1] D. Bayer and M. Stillman. A Theorem on Refining Division Orders by the Reverse Lexicographic Order. *Duke Mathematical Journal*, 55(2):321–328, 1987. 1.1

[2] T. Becker, H. Kredel, and V. Weispfenning. *Gröbner Bases: a Computational Approach to Commutative Algebra*. Springer-Verlag, London, UK, April 1993. 1.1, 1.3, 2.3, 2.4

[3] W. Bosma, J. Cannon, and C. Playoust. The Magma Algebra System. I. The User Language. *J. Symbolic Computation*, 24(3-4):235–265, 1997. 2.2, 4.1

[4] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal)*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. (English translation in Journal of Symbolic Computation, 2004). 1.1, 1.2, 1

[5] B. Buchberger. A Criterion for Detecting Unnecessary Reductions in the Construction of Gröbner Bases. In *Proceedings of the EUROSAM 79 Symposium on Symbolic and Algebraic*

Manipulation, volume 72, pages 3–21, London, UK, 1979. Johannes Kepler University Linz, Springer, Berlin - Heidelberg - New York. 3

[6] B. Buchberger. *Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory.* Reidel Publishing Company, Dodrecht - Boston - Lancaster, 1985. 1.1, 1.1, 1.2, 2

[7] J. Buchmann, D. Cabarcas, J. Ding, and M. S. E. Mohamed. Flexible Partial Enlargement to Accelerate Gröbner Basis Computation over $\mathbb{F}_2$. In *Proceedings of The Third International Conference on Cryptology in Africa, (AfricaCrypt2010)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2010. Accepted for publication. 5.1.1

[8] S. Collart, M. Kalkbrener, and D. Mall. Converting Bases with the Gröbner walk. *J. Symb. Comput.*, 24(3-4):465–469, 1997. 1.1

[9] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer New York, 2007. 1.1

[10] T. A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006. 3.4.1

[11] J. Ding. Mutants and its Impact on Polynomial Solving Strategies and Algorithms. Privately distributed research note, University of Cincinnati and Technical University of Darmstadt, 2006. 5.1.2

[12] J. Ding, D. Carbarcas, D. Schmidt, J. Buchmann, and S. Tohaneanu. Mutant Gröbner Basis Algorithm. In *Proceedings of the 1st international conference on Symbolic Computation and Cryptography (SCC08)*, pages 23 – 32, Beijing, China, April 2008. LMIB. 5.1.2

[13] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, New York, NY, USA, 1989. 3.4.1, 3.4.2

[14] D. Dummit and R. Foote. *Abstract Algebra*. John Wiley and Sons, Inc., 2004. 1.1

[15] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases (F_4). *Pure and Applied Algebra*, 139(1-3):61–88, June 1999. 2, 2.1, 2.2, 2.2, 2.2, 3.4, 3.4.1, 5.1.1

[16] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases Without Reduction to Zero (F_5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation (ISSAC)*, pages 75 – 83, Lille, France, July 2002. ACM. 5.1.2

[17] J. C. Faugère, P. Gianni, D. Lazard, and T. Mora. Efficient Computation of Zero-Dimensional Gröbner Bases by Change of Ordering. *J. Symb. Comput.*, 16(4):329–344, 1993. 1.1

[18] R. Gebauer and H. Möller. On an Installation of Buchberger's Algorithm. *Journal of Symbolic Computation*, 6(2-3):275–286, 1988. 3, 1.3

[19] A. Giovini, F. Mora, G. Niesi, L. Robbiano, and C. Traverso. "One Sugar Cube, Please" or Selection Strategies in the Buchberger Algorithm. In *ISSAC '91: Proceedings of the 1991 international symposium on Symbolic and algebraic computation*, pages 49–54. ACM Press, 1991. 1

[20] D. R. Grayson and M. E. Stillman. Macaulay2, a Software System for Research in Algebraic Geometry. Available at http://www.math.uiuc.edu/Macaulay2/. 4.1

[21] G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 3-1-1 — A Computer Algebra System for Polynomial Computations, 2010. http://www.singular.uni-kl.de. 4.1

[22] B. A. LaMacchia and A. M. Odlyzko. Solving Large Sparse Linear Systems over Finite Fields. In *CRYPTO*, pages 109–133, 1990. 3.4.1

[23] D. Lazard. Resolution des Systemes d'Equations Algebriques. *Theoretical Computer Science*, 15(1):77 – 110, 1981. 1.3

[24] D. Lazard. Gröbner-Bases, Gaussian Elimination and Resolution of Systems of Algebraic Equations. In *EUROCAL '83: Proceedings of the European Computer Algebra Conference on Computer Algebra*, pages 146–156, London, UK, 1983. Springer-Verlag. 1.3

[25] D. Lazard. Thirty Years of Polynomial System Solving, and Now? *J. Symb. Comput.*, 44(3):222–231, 2009. 3

[26] C. E. McKay. An Analysis of Improvements to Buchberger's Algorithm for Gröbner Basis Computation. Master's thesis, University of Maryland, College Park, 2004. 2.2

[27] M. S. E. Mohamed, D. Cabarcas, J. Ding, J. Buchmann, and S. Bulygin. MXL3: An efficient algorithm for computing Gröbner bases of zero-dimensional ideals. In *Proceedings of The 12th international Conference on Information Security and Cryptology, (ICISC 2009)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, December 2009. Accepted for publication. 5.1.2

[28] M. S. E. Mohamed, W. S. A. E. Mohamed, J. Ding, and J. Buchmann. MXL2: Solving Polynomial Equations over GF(2) using an Improved Mutant Strategy. In *Proceedings of The Second international Workshop on Post-Quantum Cryptography, (PQCrypto08)*, Lecture

Notes in Computer Science, pages 203–215, Cincinnati, USA, October 2008. Springer-Verlag, Berlin. 5.1.1

[29] H. Möller, F. Mora, and C. Traverso. Gröbner Bases Computation Using Syzygies. In *ISSAC 92*, pages 320–328, New York, NY, USA, 1992. ACM Press. 3, 5.1.2

[30] C. Pomerance and J. W. Smith. Reduction of Huge, Sparse Matrices over Finite Fields Via Created Catastrophes. *Experiment. Math*, 1:89–94, 1992. 3.4.1

[31] D. Schmidt. *POLYPAK: an Algebraic Processor for Computations in Celestial Mechanics*, pages 111–120. Chudnovsky and Jenks Editors, Marcel Deker, 1989. 3.3.4

Appendix A

Source Code

We include in the following pages an excerpt of the source code of our implementation of the F_4 algorithm. Only the main skeleton is displayed here, due to space limitations. For the same reason, many code comments were suppressed, but we hope the thesis itself should serve as code documentation.

```
void F4(const List_poly &F,  //< input sequence
        List_poly &G      //< output GB of <F>.
        )
{
    //Start F4
    vector<list<Signature>> HH;  //< Stores signatures of H after symbolic preprocessing on each step
    vector<List_poly> GG;        //< Stores Htilde after row reduction on each step
    vector<list<Poly_poly>*> BB(MAX_DEG, NULL); //< Stores all unprocessed pairs discriminated by degree
    vector<vector<bool>> ncsry; //< ncsry[step][i]==true iff the i-th poly in GG[step] is necessary
    //reserve space for HH GG and unnecessary
    HH.reserve(MAX_STEP);
    GG.reserve(MAX_STEP);
    ncsry.reserve(MAX_STEP);
    //step zero
    HH.push_back(list<Signature>());            //< HH[0] = \empty
    GG.push_back(F);                            //< GG[0] = F
    ncsry.push_back(vector<bool>(F.size(),true)); //< create necessary vector for new elts F. All true
    update(GG, &BB, &ncsry);
    //prepare for step 1
    short deg = select(BB);            //< degree of the step
    Monomial::increase_indexed(deg);   //< indexes monomials up to degree deg
    while( deg != -1 && GG.size() <= MAX_STEP)  //< while there are still pairs in BB and not too many steps
{
        list<Signature> L;      //< A list of monomial-polynomial pairs
        left_right(*(BB[deg]), GG, &L);
        delete BB[deg]; BB[deg] = NULL; //<delete the pairs used
        List_poly H(deg);       //< A list of polynomials of degree up to deg
        symbolic_preprocessing(GG, ncsry, HH, &H, &L);
        HH.push_back(L);    //< signatures of H
        L.clear();
        //make a copy of the leading monomials of H before gauss
        set<Monomial::Index> lead_monomial_H(H.lead_monomial_begin(), H.lead_monomial_end());
        reduction_row_echelon(&H);
        GG.push_back(H);    //< store permanently H in GG.
        H.clear();
        ncsry.push_back(vector<bool>(GG.back().size(),true)); //< create necessary vector for new elts H
        //mark as unnecessary if leading monomial appears in the leading monomials of H before gauss
        remove_if_LM_in(GG.back(), &(ncsry.back()), lead_monomial_H);
        lead_monomial_H.clear();
        //prepare for next step
        update(GG, &BB, &ncsry);
        deg = select(BB);
        Monomial::increase_indexed(deg);   //< indexes monomials up to degree deg
}
    //Prepare Output. Move necessary to G
    deg = 0;
    for(unsigned short s = 0; s < GG.size(); s++)
        deg = MAX( deg, GG[s].degree() );
    G.reset(deg);
    for(unsigned short s = 0; s < GG.size(); s++)
    {
        for(int i_f = 0; i_f < GG[s].size(); i_f++)
        {
            if(ncsry[s][i_f])
            {
                int i_g = G.new_poly();
                multiply(0, GG[s], i_f, &G, i_g);
            }
        }
    }
}
```

```
///implements the normal selection strategy for F4
/** returns the minimum degree of any pair in B
    the degree of a pair(f,g) is the degree of lcm(LM(f),LM(g)).
    \param B a list of pairs to select form
    \return the minimum degree of elements from B
*/
short select(vector<list<Poly_poly>*> &BB)
{
    short i;
    for(i = 0; i < MAX_DEG; i++)
    {
        if(BB[i])
        {
            if(BB[i]->size() == 0)
            {
                delete BB[i];
                BB[i] = NULL;
            }
            else
                return i;
        }
    }
    return -1;
 }

/// Stores in L the left and right monomial-polys products of the pairs in Bstar
/// and their corresponding multiplication-simplification in H
/**
    L = { (t,f) : {f,g}\in B^*, t=lcm(LM(f),LM(g))/LM(f) }
*/
void left_right(const list<Poly_poly> &Bstar,   ///< Input, selected pairs
                const vector<List_poly> &GG,    ///< Input, all computed polynomials
                list<Signature> *L              ///< Output, Signatures (t,f) from B* elements
                )
{
    Monomial::Index lt1, lt2;           //< leading monomials of the polys corresponding to i_f1 and i_f2
    list<Poly_poly>::const_iterator i_B;//< an iterator on Bstar
    Signature sig1, sig2;               //< signatures to be stored in L
    for(i_B = Bstar.begin(); i_B != Bstar.end(); i_B++) //< for each pair
    {
        //basic info of the signatures
        sig1.step = i_B->step1;
        sig2.step = i_B->step2;
        sig1.i_f = i_B->i_f1;
        sig2.i_f = i_B->i_f2;
        lt1 = GG[sig1.step].lead_monomial(sig1.i_f);
        lt2 = GG[sig2.step].lead_monomial(sig2.i_f);
        sig1.lt = sig2.lt = lcm_index(lt1, lt2);
        sig1.t = quo_index( sig1.lt , lt1 );
        sig2.t = quo_index( sig2.lt , lt2 );
        //insert if nor redundant
        if( find(L->begin(), L->end(), sig1) == L->end() )
            L->push_back(sig1);
        if( find(L->begin(), L->end(), sig2) == L->end() )
            L->push_back(sig2);
    }
}
```

```
/// Enlarges H with polys of the form t*f, t a monomial, f a poly in GG s.t LM(t*f) is a monomial in H.
void symbolic_preprocessing(const vector<List_poly> &GG,        ///< Input, all computed polynomials
                            const vector<vector<bool>> &ncsry,  ///< Input, all necessary elements of GG
                            const vector<list<Signature>> &HH,  ///< Input, previous signatures
                            List_poly *H,       ///< Output, new polynomials
                            list<Signature> *L  ///< Input/Output, signatures of elements of H
                            )
{
    //H = { mult(simplify(t,f,HH,GG)): (t,f)\in L }
    for(list<Signature>::iterator isig = L->begin(); isig != L->end(); isig++)
    {
        simplify(GG, HH, &(*isig));
        int i_h = H->new_poly();   //< Adds a new zero polynomial to H and returns its corresponding index
        multiply(isig->t, GG[isig->step], isig->i_f, H, i_h);
    }
    Signature sig;  //< stores a signature to search for.
    // leading monomials of H in a set that allows for log(n) time search
    set<Monomial::Index> H_lead_monomial(H->lead_monomial_begin(), H->lead_monomial_end());
    //for each monomial up to the largest degree of H
    for(sig.lt = Monomial::max_monomial(H->degree()); sig.lt >= 0; sig.lt--)
    {
        //lt \in M(H) and lt \notin LM(H)
        if( ! H->all_zero_coefs(sig.lt) &&
            find(H_lead_monomial.begin(), H_lead_monomial.end(), sig.lt) == H_lead_monomial.end() )
        {
            //search for a LM that divides t on previous steps
            bool found = false;
            for(sig.step = GG.size()-1; sig.step >= 0; sig.step--)//< for each step
            {
                const List_poly &G = GG[sig.step];   //< List of polys corresponding to that step
                const vector<bool> &ncsry_step = ncsry[sig.step];   //< corresponding necessary elts
                int m = G.size();   //< Number of polys in G
                for(sig.i_f = m-1; sig.i_f >= 0; sig.i_f--)     //< for each poly in G
                {
                    if(ncsry_step[sig.i_f]) //< if it is necessary
                    {
                        if(does_divide(G.lead_monomial(sig.i_f), sig.lt))    //< if LM(f) divides t
                        {
                            sig.t = quo_index(sig.lt, G.lead_monomial(sig.i_f));
                            simplify(GG, HH, &sig);
                            int i_h = H->new_poly();    //< index of a new zero polynomial in H
                            multiply(sig.t, GG[sig.step], sig.i_f, H, i_h);
                            L->push_back(sig);  //< insert corresponding signature in L
                            found = true; //< to break the for(sig.step loop
                            break;  //< breaks for(sig.i_f loop
                        }
                    }
                }
                if(found)
                    break;  //< breaks for(sig.step loop
            }
        }
    }
}
```

```
void simplify(const vector<List_poly> &GG,          ///< Input, all computed polynomials
              const vector<list<Signature>> &HH,    ///< Input, previous signatures
              Signature *sig    ///< Input/output, signature to be simplified
              )
{
    Monomial t( *(to_monomial(sig->t)) );    //< Monomial corresponding to sig.t
    Monomial uu;                            //< divisor of t
    short dd = -1;//< degree of divisor. Serves as flag, -1 tells the program that no u is known
    short d_t = degree(t);  //< degree of t
    while (true){
        //Next divisor
        if(dd == -1)    //< if new t in place
            first_divisor(t, &uu, &dd);
        else if( !next_divisor(t, &uu, &dd) )
                break; //break while(true) loop
        short d_f = degree(GG[sig->step].lead_monomial(sig->i_f)); //< degree of f
        //search for (u,f) in HH
        bool uf_found = false;  //< A flag that tells if uf has been found
        short uf_step;    //< index of step where looking for uf
        list<Signature>::const_iterator iter;   //< iterator on the signatures of HH
        for(uf_step = 0; uf_step < (short)HH.size(); uf_step++)
        {
            const list<Signature> &H = HH[uf_step];     //< A list of polynomials from HH.
            //search for (u,f in H)
            iter = H.begin();
            if(iter != H.end() && degree(iter->lt) == dd+d_f)
            {
                while(iter != H.end())
                {
                    if(uu == iter->t && sig->step == iter->step && sig->i_f == iter->i_f)
                    {
                        uf_found = true;
                        break;          //< breaks the while loop
                    }
                    iter++;
                }
                if(uf_found)
                    break;              //< breaks the i for loop
            }
        }
        if(uf_found)    //< if there exist s in HH s.t. uf in H
        {
            sig->step = uf_step;
            //search for p in GG[sig.step] s.t. LM(p)=LM(uf)
            int i_p = find_LM(GG[sig->step], iter->lt);    //< index of p in GG[sig.step]
            if (uu == t)
            {   //return (1,p)
                sig->t = 0;
                sig->i_f = i_p;
                return;
            }
            else
            {   //return (t/uu, p)
                t = quo(t,uu);
                d_t = degree(t);
                sig->i_f = i_p;
                dd = -1;         //serves as flag that tells the program that u is not initialized
            }
        }//< if there exist H in HH s.t. uf in H
    }//while true
    sig->t = t.to_index();
}
```

```
/// Implements a list of polynomials
class List_poly
{
public:
    //Construction & destructions
    List_poly(const short d = 2);   ///< Initializes The list with maximum degree d
    List_poly(const List_poly &F);  ///< Copy constructor
    void reset(const short d);      ///< Increases the degree allowed to be stored
    void clear();                   ///< Erases all elements and releases memory
    //Access Functions
    short degree() const;   ///< Maximum degree of the polynomials
    int size()const;        ///< Number of polynomials
    vector<Monomial::Index>::const_iterator lead_monomial_begin()const; ///< beginning of leading monomials.
    vector<Monomial::Index>::const_iterator lead_monomial_end()const;   ///< end of leading monomials.
    Monomial::Index lead_monomial(const int i_f)const;  ///< Leading monomial of the i-th poly
    //functions to modify the list
    int new_poly(const int n = 1);      ///< Inserts n zero polynomials. Returns the position of the first one.
    void remove_poly(const int i_rem);  ///< Removes i-th poly.
    //Predicates
    bool all_zero_coefs(const Monomial::Index t)const;  ///< True iff for all polys the coef of t is zero.
    int nnz_polys() const;    ///< Returns the number of non-zero polynomials
    int nnz_monomials() const;    ///< Returns the number of non-zero monomials
    int nnz_coefs() const;    ///< Returns the number of non-zero coefficients
    //other functions
    friend void update_lead_monomial(List_poly *H);  ///< Updates the leading monomials for the elements in H
    friend void reduction_row_echelon(List_poly *H); ///< Reduces H to row echelon form
    friend int find_LM(const List_poly &G, const Monomial::Index t); ///< Searches a poly in G with LM t
    /// Multiplies the poly in G[i_f] by the monomial t and stores the result in H[i_h]
    friend void multiply(const Monomial::Index t, const List_poly &G, int i_g, List_poly *H, const int i_h);
    friend void multiply(const Monomial &t, const List_poly &G, int i_g, List_poly *H, const int i_h);
    friend class col_transform; ///< Class function that transforms columns between 2 List_poly
private:
    //Data members
    short deg;  ///< maximum degree
    int m_poly; ///< Number of polynomials
    /** A matrix of size m*n where n = comb(num_var+d, d) = Monomial::max_monomial(deg)+1.
        For 0 <= i <= m, 0 <= j <= n,
        coef[i][j] gives the coefficient of the i-th polynomial in the col_to_monomial(n-j) monomial
    */
    Matrix coef;
    vector<Monomial::Index> lead_t;     ///< lead_t[i] gives the leading monomial of the i-th polynomial
    //Private functions for basic access
    void set_coef(  const int i_f,
                    const Monomial::Index t,
                    const Field v); ///< Sets the coefficient of the monomial t of the i-th poly to v.
    int max_monomial() const;       ///< maximum monomial up to this degree
    int monomial_to_col(const Monomial::Index t) const; ///< Converts an index monomial into column
    Monomial::Index col_to_monomial(const int j) const; ///< Converts a column into an index monomial
    //input-output
    friend ostream& operator<<(ostream& out, const List_poly &F);   ///< Outputs to a stream
};
```

```
/// A Monomial is a map from variables into exponents
/** Monomial t;
    t[2] = 3;
    t[7] = 1;
    creates the monomial x2^3 * x7
*/
class Monomial : public map<short,short>{
public:
    typedef int Index; ///< An Index represents a monomial indexed in indx_monomials
    //Conversion functions (monomial_static.cpp)
    Index to_index();                      ///< returns the corresponding index, or -1 if not indexed
    friend Monomial* to_monomial(const Index t);   ///< returns a pointer to the monomial indexed as t
    //Static functions (monomial_static.cpp)
    static void set_num_var(const short n);         ///< sets the number of variables to n
    static void increase_indexed(const short deg);  ///< indexes monomials up to degree deg
    static Index max_monomial(const short deg);     ///< Maximum indexed monomials of degree deg
    //divisor generation (monomial_static.cpp)
    friend void first_divisor(const Monomial &t, Monomial *u, short *d);
    friend bool next_divisor(const Monomial &t, Monomial *u, short *d);
    //basic operations (monomial.cpp)
    bool operator==(const Index t2) const;  ///< true if t1 == t2
    bool operator!=(const Index t2) const;  ///< true if t1 != t2
    friend short degree(const Index t);     ///< Degree of the monomial t
    friend short degree(const Monomial &t); ///< Degree of the monomial t
    friend Monomial mult(const Monomial &t1, const Monomial &t2);   ///< t1*t2 as a Monomial
    friend Monomial mult(const Monomial &t1, const Index t2);       ///< t1*t2 as a Monomial
    friend Monomial mult(const Index t1, const Index t2);           ///< t1*t2 as a Monomial
    friend Index mult_index(const Index t1, const Index t2);        ///< Index of t1*t2
    friend Index mult_index(const Monomial &t1, const Index t2);    ///< Index of t1*t2
    friend Monomial quo(const Monomial &t1, const Monomial &t2);///< t1/t2. Assumes t2 divides t1
    friend Monomial quo(const Monomial &t1, const Index t2);    ///< t1/t2. Assumes t2 divides t1
    friend Monomial quo(const Index t1, const Monomial &t2);    ///< t1/t2. Assumes t2 divides t1
    friend Monomial quo(const Index t1, const Index t2);        ///< t1/t2. Assumes t2 divides t1
    friend Index quo_index(const Monomial &t1, const Monomial &t2); ///< t1/t2 if t2 divides t1, -1 otherwise
    friend Index quo_index(const Index t1, const Monomial &t2);     ///< t1/t2 if t2 divides t1, -1 otherwise
    friend Index quo_index(const Index t1, const Index t2);         ///< t1/t2 if t2 divides t1, -1 otherwise
    friend bool does_divide(const Monomial &t1, const Monomial &t2);///< true if t1 divides t2
    friend bool does_divide(const Index t1, const Monomial &t2);    ///< true if t1 divides t2
    friend bool does_divide(const Index t1, const Index t2);        ///< true if t1 divides t2
    friend Monomial lcm(const Monomial &t1, const Monomial &t2);///< Least common multiple of t1 and t2
    friend Monomial lcm(const Index t1, const Index t2);        ///< Least common multiple of t1 and t2
    friend Index lcm_index(const Monomial &t1, const Index t2);///< Index of least common multiple of t1 & t2
    friend Index lcm_index(const Index t1, const Index t2);    ///< Index of least common multiple of t1 & t2
    friend bool are_disjoint(const Monomial &t1, const Monomial &t2);///< True if t1 and t2 are disjoint
    friend bool are_disjoint(const Index t1, const Index t2);        ///< True if t1 and t2 are disjoint
private:
    //Static members (defined in ter_static.cpp)
    static short num_var;   ///< Number of variables
    static short cur_deg;   ///< maximum degree indexed so far
    /// Indexed monomials.
    /**All monomials up to certain degree in ascending order according to the given monomial order*/
    static vector<Monomial> indx_monomials;
    /// multiplication table.
    /** For indexed monomials up to degree cur_deg-1 gives the index of the monomial
        that results from multiplying by a variables x_i, i.e.
        x_i * indx_monomials[j] = indx_monomials[mult_table[i + num_var*j]]*/
    static vector<Index> mult_table;
    /// Degree separations on indx_monomials. deg_sep[d] is the last index of a monomial of degree d.
    static vector<Index> deg_sep;
    //private functions
    friend short compare(const Monomial &t1, const Monomial &t2);   ///< Compares t1 and t2 in glex
};
```