New Algorithms for Computing Groebner Bases

Frank Volny

Clemson University, fvolny@g.clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Part of the Applied Mathematics Commons

Recommended Citation


https://tigerprints.clemson.edu/all_dissertations/725

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
NEW ALGORITHMS FOR COMPUTING GRÖBNER BASES

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematical Sciences

by
Frank Volny IV
May 2011

Accepted by:
Dr. Shuhong Gao, Committee Chair
Dr. Brian Dean
Dr. Elena Dimitrova
Dr. Peter Kiessler
Dr. Matthew Macauley
Abstract

In this thesis, we present new algorithms for computing Gröbner bases. The first algorithm, G2V, is incremental in the same fashion as F5 and F5C. At a typical step, one is given a Gröbner basis $G$ for an ideal $I$ and any polynomial $g$, and it is desired to compute a Gröbner basis for the new ideal $(I,g)$, obtained from $I$ by joining $g$. Let $(I:g)$ denote the colon ideal of $I$ divided by $g$. Our algorithm computes Gröbner bases for $(I,g)$ and $(I:g)$ simultaneously. In previous algorithms, S-polynomials that reduce to zero are useless, in fact, F5 tries to avoid such reductions as much as possible. In our algorithm, however, these “useless” S-polynomials give elements in $(I:g)$ and are useful in speeding up the subsequent computations. Computer experiments on some benchmark examples indicate that our algorithm is much more efficient (two to ten times faster) than F5 and F5C.

Next, we present a more general algorithm that matches Buchberger’s algorithm in simplicity and yet is more flexible than G2V. Given a list of polynomials, the new algorithm computes simultaneously a Gröbner basis for the ideal generated by the polynomials and a Gröbner basis for the leading terms of the syzygy module of the polynomials. For any term order for the ideal, one may vary the term order for the syzygy module. Under one term order for the syzygy module, the new algorithm specializes to the G2V algorithm, and under another term order for the syzygy module, the new algorithm may be several times faster than G2V, as indicated by computer experiments on benchmark examples.

Finally, we present a solid theoretical framework for G2V and GVW which makes the algorithm much more understandable. This theory also gives a major improvement of the GVW algorithm. A proof of termination is provided for all algorithms, and an argument is made that GVW computes the fewest number of generators for the signature based algorithms used by GVW and F5 (similarly for G2V and F5C).
# Table of Contents

Title Page ................................................................. i
Abstract ................................................................. ii
List of Tables .......................................................... v
List of Figures .......................................................... vi
List of Listings .......................................................... vii

1 An Introduction to Gr"obner Bases .................................. 1
   1.1 Gr"obner Bases and Buchberger’s Algorithm .................. 1
   1.2 Applications .................................................. 7
   1.3 Examples of Large Gr"obner Basis Calculations .......... 9
      1.3.1 Exponential Number of Generators .................. 10
      1.3.2 Exponential Degree of Signatures .................. 15
   1.4 Recent Algorithms ........................................ 16
   1.5 Organization and Contributions .......................... 18

2 The G2V Algorithm .................................................. 19
   2.1 Introduction ................................................. 19
   2.2 Theory .................................................... 21
   2.3 Algorithm ................................................ 26
   2.4 Comparisons and Conclusions ............................ 30

3 The GVW Algorithm ................................................ 33
   3.1 Introduction ............................................... 33
   3.2 Theory ................................................... 34
   3.3 Algorithm, Term Orderings and Time Comparison .......... 42
      3.3.1 Algorithm and Its Correctness .................... 42
      3.3.2 Finite Termination and Gr"obner Bases for the Syzygy Module 45
      3.3.3 Term Orders ...................................... 45
      3.3.4 Relationship to Other Algorithms ................. 46
      3.3.5 Comparison ..................................... 47
   3.4 Algorithm for Quotient Rings and Modules ............... 50
      3.4.1 Quotient Rings .................................. 50
      3.4.2 Modules ......................................... 51
   3.5 Conclusions .............................................. 52

4 An Improved GVW Algorithm ....................................... 56
   4.1 Introduction .............................................. 56


List of Tables

2.1 Runtimes in seconds and ratios of runtimes for various test cases in Singular 3.1.0.6 on an Intel Core 2 Quad 2.66 GHz. The #generators refers to reduced Gröbner basis ............................ 31

2.2 The number of generators in the Gröbner basis in the last iteration but before computing a reduced Gröbner basis. Of course, F5 never computes the reduced Gröbner basis ........................................ 32

2.3 The maximum amount of memory (in KiB) Singular 3.1.0.6 used from startup to the conclusion of the Gröbner basis computation. Memory amounts obtained with “memory(2);” ................................................................. 32

2.4 The count of J-pairs or S-polynomials processed by F5, F5C and G2V ................................. 32

3.1 Runtimes in seconds comparing F5, F5C and G2V (GVW under POT ordering) for various test cases in Singular 3110 on an Intel Core 2 Quad 2.66 GHz. This table is reproduced from [19] ......................................................... 47

3.2 Runtimes in seconds using Singular 3110 for different term orders ............................ 48

3.3 Runtimes in seconds using our C++ implementation for different term orders ............. 48

3.4 A comparison of the count of the J-pairs processed for the different term orders .... 49

3.5 Sizes of Gröbner bases before any interreduction for different term orders ......... 49

3.6 Maximal amount of memory used (MiB) by Singular for different term orders .... 49

4.1 Sizes of minimal strong Gröbner bases for different term orders ......................... 67

4.2 Counts of the J-pairs or S-polynomials processed by F5, F5C, G2V, and GVWHS under POT, TOP, g1 and g2 orders ........................................ 68

5.1 Runtimes and memory usages for policy P5. These values are also representative of policies P2, P4 and P7 ............................................................... 72

5.2 Runtimes and memory usages for policy P1. These values are also representative of policies P3, P6 and P8 ............................................................... 73

5.3 The count of reductions to zero with and without syzygies of the form (5.1) .... 76

5.4 Runtimes and memory usages for naive polynomial subtraction as in equation (5.2) . 80

5.5 Runtimes and memory usages for queue based polynomial subtraction ................ 81
List of Figures

1.1 Buchberger’s algorithm ................................................... 5
1.2 The general flow of information in modern cryptosystems ............... 9

2.1 The G2V algorithm ........................................................ 29

3.1 The GVW algorithm ....................................................... 54
3.2 the GVW algorithm applied to quotient rings .......................... 55

4.1 The GVWHS algorithm .................................................... 69

5.1 Counting bits in logarithmic time ...................................... 84
List of Listings

5.1 The implementation of policies in C++ code ........................................... 73
5.2 Counting one-bits in parallel ................................................................. 84
5.3 Calculating one-bits using the builtin command ..................................... 84
5.4 The conversion of multilinear monomials to multicubic monomials .............. 90
5.5 Determining whether a multicubic monomial is square-free ..................... 91
6 A sample makefile showing the compiler flags used for the code to follow .... 93
7 A simple function main that utilizes the code to follow ............................ 95
8 The GVWHS algorithm at a high level as presented in figure 4.1 ................. 97
9 A priority queue for the selection of the next J-pair using policies from section 5.1 104
10 Polynomial subtraction as described in section 5.3 .................................. 106
11 The J-pair class ..................................................................................... 109
12 The polynomial class ............................................................................. 111
13 The term class ....................................................................................... 115
14 The monomial class ............................................................................... 118
15 The field class ....................................................................................... 121
16 The term orders ..................................................................................... 123
17 The module term class .......................................................................... 125
18 The module term orders as described in section 3.3.3 ............................... 127
19 The polynomial formatter base class to read and write polynomials ............. 129
20 The polynomial formatter Singular class ................................................... 131
21 The GVW algorithm at a high level as described in section 5.4 .................... 135
22 The polynomial class as described in section 5.4.2 .................................. 142
23 The monomial class as described in section 5.4.1 .................................... 146
24 The term as described in section 5.4.1 .................................................. 150
25 The signature class as described in section 5.4.3 ...................................... 152
26 G2V in Singular as in figure 2.1 ............................................................... 157
27 GVW as in figure 3.1 ............................................................................. 162
28 Examples used in Singular ................................................................... 168
Chapter 1

An Introduction to Gröbner Bases

Polynomial systems are ubiquitous in mathematics, science and engineering, and the theory of Gröbner bases is one of the most powerful tools for solving polynomial systems. Buchberger introduced in 1965 the first algorithm for computing Gröbner bases, and it has been implemented in most computer algebra systems (e.g., Maple, Mathematica, Magma, Sage, Singular, Macaulay 2, CoCoA, etc). Computing Gröbner bases is a basic routine that is essential in many computational tasks in algebra and algebraic geometry. Due to the importance of Gröbner bases, Bruno Buchberger was awarded the 2007 ACM Paris Kanellakis Theory and Practice Award and the Golden Medal of Honor for his original development of Gröbner basis theory.

1.1 Gröbner Bases and Buchberger’s Algorithm

Given a polynomial ideal \( \langle f_1, \ldots, f_m \rangle \subset \mathbb{F}[x_1, \ldots, x_n] \), calculating a Gröbner basis finds another set of generators \( g_1, \ldots, g_k \), such that \( \langle g_1, \ldots, g_k \rangle = \langle f_1, \ldots, f_m \rangle \), with some very important computational properties. For simplicity of notation, we write monomials \( x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n} \) as \( x^\alpha \) where \( \alpha = (a_1, \ldots, a_n) \in \mathbb{Z}_{\geq 0}^n \). For our purposes, a term is a coefficient times a monomial. While polynomials are sums of terms, it is convenient for us to always write those terms in decreasing order. Term orders allow us to specify, for each polynomial, a leading monomial (lm) (leading term (lt) or leading coefficient (lc)). Thus, the leading monomial (term or coefficient) will be the largest nonzero term of a polynomial with respect to a given term order.

Definition. A **term ordering** \( \prec \) on the monomials (or terms) of \( \mathbb{F}[x_1, \ldots, x_n] \) satisfies the following
properties:

1. \( \prec \) is a total-ordering on the monomials of \( \mathbb{F}[x_1, \ldots, x_n] \),

2. \( \prec \) is a well-ordering of the monomials of \( \mathbb{F}[x_1, \ldots, x_n] \), and

3. For monomials \( x^\alpha, x^\beta \) and \( x^\gamma \), we require \( x^\alpha \prec x^\beta \) if and only if \( x^\alpha x^\gamma \prec x^\beta x^\gamma \).

**Example 1.1** (Lexicographic (Lex) order). Let \( \alpha, \beta \in \mathbb{Z}_{\geq 0}^n \). \( x^\alpha \prec x^\beta \) if and only if \( \alpha_i < \beta_i \) for some \( 1 \leq i \leq n \) and \( \alpha_j = \beta_j \) for all \( 1 \leq j < i \).

**Example 1.2** (Degree Reversed Lexicographic (DegRevLex) order). Let \( \alpha, \beta \in \mathbb{Z}_{\geq 0}^n \). \( x^\alpha \prec x^\beta \) if and only if \( \deg(x^\alpha) < \deg(x^\beta) \) or \( \deg(x^\alpha) = \deg(x^\beta) \) and \( \alpha_i > \beta_i \) for some \( 1 \leq i \leq n \) and \( \alpha_j = \beta_j \) for all \( i < j \leq n \).

**Lemma 1.3.** There is no infinitely decreasing sequence of monomials.

**Proof.** Suppose for the sake of contradiction that \( x^{\alpha_1} \succ x^{\alpha_2} \succ \cdots \) is an infinitely decreasing sequence of monomials. It is not possible that \( x^{\alpha_i} \mid x^{\alpha_j} \) for any \( i < j \) as a consequence of condition (3) of the definition of a term order. Thus, the sequence of ideals of the form

\[
I_n = \langle x^{\alpha_1}, x^{\alpha_2}, \ldots, x^{\alpha_n} \rangle \subset \mathbb{F}[x_1, \ldots, x_n]
\]

would form an infinitely ascending chain of ideals. But since \( \mathbb{F}[x_1, \ldots, x_n] \) is a Noetherian ring, this is impossible. Therefore, no such infinitely decreasing sequence of monomials exists. \( \square \)

For a given term order, we can perform polynomial division (or reduction). For polynomials \( f, g \in R = \mathbb{F}[x_1, \ldots, x_n] \), we say that \( f \) is **top-reducible** (or divisible) by \( g \) if \( \text{lm}(g) \) divides \( \text{lm}(f) \). The top-reduction is then

\[
f - \frac{\text{lt}(f)}{\text{lt}(g)} \cdot g.
\]

This top-reduction will have the effect of cancelling the leading monomial of \( f \) by the appropriate multiple of \( g \). Also, if \( F = \{g_1, \ldots, g_m\} \subset R \) is a set of polynomials, we say that \( f \) is **top-reducible** (or divisible) by \( F \) if for some \( 1 \leq i \leq m \), we have that \( g_i \) top-reduces \( f \). When \( f \) is fully top-reduced by (or “with respect to”) \( F \), it must be the case that the leading monomial of \( f \) is not divisible by any of \( \text{lm}(F) = \{\text{lm}(g_1), \ldots, \text{lm}(g_m)\} \). In this case, we say that \( f \) is top-irreducible by \( F \).
When \( f \) is top-irreducible by \( F \), we may proceed to reduce the lower order terms of \( f \), or its tail \( f - \text{lm}(f) \). If any of the terms of \( f \) are divisible by \( \text{lm}(F) \), we say that \( f \) is reducible by \( F \). We may continue until no term in \( f \) is reducible by \( F \). In general, this irreducible remainder of \( f \) depends on the order of reduction, so it is not unique.

**Example 1.4.** Let \( I = \langle xy - 1, x - y \rangle \subset R = \mathbb{F}[x_1, \ldots, x_n] \) be a polynomial ideal ordered by the lexicographic order such that \( x \succ y \). We would like to write \( (xy + 1) + I \in R/I \) uniquely. One might think that fully reducing \( xy + 1 \) by \( F = \{ xy - 1, x - y \} \) will produce the result, but the reduction is not unique. To see this, first top-reduce \( xy + 1 \) by \( xy - 1 \) to get \( 2 \) which is fully reduced by \( F \). Also, we may top-reduce \( xy + 1 \) instead by \( x - y \) to get \( y^2 + 1 \) which is also fully reduced by \( F \).

**Definition.** \( G = \{ g_1, \ldots, g_k \} \) is a \textbf{Gröbner basis} for \( I = \langle G \rangle \subset \mathbb{F}[x_1, \ldots, x_n] \) if for every nonzero \( f \in I \), \( f \) is top-reducible by \( G \), that is \( \langle \text{lm}(G) \rangle = \langle \text{lm}(I) \rangle \).

Thus a Gröbner basis for an ideal captures all the information about its leading monomials. Immediately from this very simple definition, we have the following results.

**Lemma 1.5.** Let \( G \subset I \subset \mathbb{F}[x_1, \ldots, x_n] \) be a Gröbner basis for an ideal \( I \). If \( f \in I \), then \( f \) always reduces to \( 0 \) after a finite number of top-reductions by elements of \( G \).

**Proof.** As long as \( f \) is nonzero, \( f \) has a top-reductor \( g \in G \). We perform the top-reduction and store the result back into \( f \) as follows.

\[
f := f - \frac{\text{lt}(f)}{\text{lt}(g)} \cdot g
\]

We repeat this process as long as needed. Only a finite number of top-reductions are required as with each top-reduction, the leading monomial of \( f \) decreases. Lemma 1.3 guarantees this forms a finite sequence of decreasing monomials.

Let \( f \) be a polynomial in \( \mathbb{F}[x_1, \ldots, x_n] \) and \( G \) a Gröbner basis for some ideal. We define the **normal form of \( f \) with respect to \( G \)** to be the remainder of \( f \) after fully reducing by \( G \). Thus, none of the terms of a normal form are in the monomial ideal \( \langle \text{lm}(I) \rangle \). A very useful property of Gröbner basis is that the normal form is always unique.

**Lemma 1.6.** The normal form of \( f \in \mathbb{F}[x_1, \ldots, x_n] \) with respect to Gröbner basis \( G \) is unique.

**Proof.** Let \( I = \langle G \rangle \subset R = \mathbb{F}[x_1, \ldots, x_n] \) be as above. Suppose for the sake of contradiction that we can find two different polynomials \( h \) and \( g \) that are both normal forms of \( f \) with respect to \( G \).
We note that as coset representatives, they are all equal, that is \( h + I = g + I = f + I \) in \( R/I \). Subtracting, we must have \( h - g \in I \). By lemma 1.5, the normal form of \( h - g \) is zero. Thus, either \( h = g \) or some term of either \( h \) or \( g \) is reducible by an element of \( G \). Either way, we have a contradiction proving the uniqueness of a normal form.

Already, we have seen some of the most important properties of Gröbner bases. For many more useful properties, see [10]. As section 1.2 begins to hint, having a Gröbner basis for an ideal solves many problems. Thus, computing a Gröbner basis is a major step in many algorithms and the focus of this dissertation.

Bruno Buchberger, the inventor of Gröbner basis theory, provided the first algorithm for their computation: Buchberger’s algorithm. His algorithm follows from a simple criterion which we describe shortly.

**Definition.** Given two polynomials \( f, g \in F[x_1, \ldots, x_n] \), their **S-polynomial**\(^1\), written \( S(f, g) \), is
\[
S(f, g) = \frac{lcm(lm(f), lm(g))}{lm(f)} \cdot f - \frac{lc(f)}{lc(g)} \cdot \frac{lcm(lm(g), lm(f))}{lm(g)} \cdot g.
\]

The monomial multipliers placed in front of the \( f \) and \( g \) are the smallest required to cancel the leading terms of \( f \) and \( g \). S-polynomials are used to check whether a set of generators is a Gröbner basis.

**Lemma 1.7** (Buchberger’s Criterion). A basis \( \{f_1, \ldots, f_m\} \subset F[x_1, \ldots, x_n] \) is a Gröbner basis if and only if every S-polynomial \( S(f_i, f_j) \) \( 1 \leq i < j \leq m \), **top-reduces** to zero by \( \{f_1, \ldots, f_m\} \).

Buchberger’s criterion actually gives us a little more than a way to detect the Gröbner property, it gives us an algorithm for producing Gröbner bases. If an S-polynomial \( S(f, g) \) does not reduce to zero by the basis, it will after we augment the basis with \( S(f, g) \) fully reduced by the current basis. Thus, Buchberger’s algorithm is presented in figure 1.1.

**Example 1.8.** Compute a Gröbner basis for \( \langle xy - 1, x - y \rangle \subset F[x_1, \ldots, x_n] \) using lex order with \( x \succ y \).

- \( S(xy - 1, x - y) = y^2 - 1 \) gets added to the generating list.

Now, our new generating list is \( F = \{xy - 1, x - y, y^2 - 1\} \).

\(^1\)S-polynomial is short for syzygy polynomial.
Buchberger’s Algorithm for computing Gröbner bases

| Input: | $f_1, \ldots, f_m \in R = \mathbb{F}[x_1, \ldots, x_n]$ and a term order for $R$, |
| Output: | A Gröbner basis for $I = \langle f_1, \ldots, f_m \rangle$, |

| Variables: | $F$ a list of polynomials representing the basis, |
|           | $SP$ a list of pairs $(i, j) \in \mathbb{N}^2$, where $S(f_i, f_j)$ is to be processed. |

**Step 0.** $F = [f_1, \ldots, f_m]$ and $SP = \{(i, j) : 1 \leq i < j \leq m\}$.

**Step 1.** Let $(i, j) \in SP$ and set $SP := SP \setminus \{(i, j)\}$.

**Step 2.** Form $S(f_i, f_j)$ and reduce it fully by $F$ to get remainder $f_{m+1}$.

**Step 3.** If $f_{m+1} \neq 0$, then

- $i)$ Set $F := F \cup \{f_{m+1}\}$.
- $ii)$ Set $SP := SP \cup \{(i, m+1) : 1 \leq i \leq m\}$
- $iii)$ Set $m := m + 1$.

**Step 4.** While $SP$ is not empty, go to step 1.

*Return:* $F$.

Figure 1.1: Buchberger’s algorithm

- $S(xy - 1, y^2 - 1) = x - y$ top-reduces to zero.
- $S(x - y, y^2 - 1) = x - y^3$ top-reduces to $-(y^3 - y)$ top-reduces to zero.

Thus $F = \{xy - 1, x - y, y^2 - 1\}$ is a Gröbner basis for $\langle F \rangle$.

While Buchberger’s algorithm is easy to state, it contains many inefficiencies. In step (3) of figure 1.1, when $f_{m+1} = 0$, no information is gained. The reductions to zero contribute no new information about the leading monomials of the ideal. Furthermore, these reductions to zero are quite expensive. The earlier we can detect that an S-polynomial is going to reduce to zero, the better. One example is Buchberger’s product criterion.

**Lemma 1.9.** The set of generators $F = \{f_1, \ldots, f_m\}$ is a Gröbner basis for $\langle F \rangle$ if and only if each $S$-polynomial $S(f_i, f_j), 1 \leq i < j \leq m$ can be written as follows.

$$S(f_i, f_j) = \sum_{k=1}^{m} h_k f_k \quad \text{such that} \quad \text{lm}(h_k f_k) \preceq S(f_i, f_j),$$

with equality for only one $1 \leq k \leq m$.

**Criterion 1.10** (The Product Criterion). For $f, g \in \mathbb{F}[x_1, \ldots, x_n]$, if $\text{lm}(f)$ and $\text{lm}(g)$ are relatively prime (that is $\text{lcm}(\text{lm}(f), \text{lm}(g)) = \text{lm}(f) \cdot \text{lm}(g)$), then $S(f, g)$ will top-reduce to zero by $\{f, g\}$. 

5
Proof. Let \( f, g \in \mathbb{F}[x_1, \ldots, x_n] \) such that \( \text{lm}(f) \) and \( \text{lm}(g) \) are relatively prime. We assume for simplicity that both \( f \) and \( g \) are monic. Then

\[
S(f, g) = \frac{\text{lcm}(\text{lm}(f), \text{lm}(g)) \cdot f - \frac{\text{lc}(f)}{\text{lc}(g)} \cdot \text{lcm}(\text{lm}(g), \text{lm}(f)) \cdot \text{lm}(f)}{\text{lcm}(\text{lm}(g), \text{lm}(g))} \cdot g
\]

\[
= \text{lm}(g) \cdot f - \text{lm}(f) \cdot g
\]

\[
= (g - g_1) \cdot f - (f - f_1) \cdot g
\]

\[
= g \cdot f_1 - f \cdot g_1.
\]

where \( f_1 = f - \text{lm}(f) \) and \( g_1 = g - \text{lm}(g) \) are the tails of \( f \) and \( g \). We invoke lemma 1.9 to see that \( \{f, g\} \) is a Gröbner basis for \( \langle f, g \rangle \). Thus, \( S(f, g) \) reduces to zero from polynomial reduction by \( \{f, g\} \).

\( \square \)

**Example 1.11.** Let \( p \) be a prime and \( R = \mathbb{F}_p[x_1, \ldots, x_n] \) be a polynomial ring. Then by the product criterion, the field equations \( F = \{x_i^p - x_i, 1 \leq i \leq n\} \) are a Gröbner basis (for \( \langle F \rangle \subset R \)).

Various other criteria designed to avoid the reduction of S-polynomials are described in [4, 5, 10]. The algorithms to follow in subsequent chapters generate information about the syzygy module of \( F = \{f_1, \ldots, f_m\} \subset R = \mathbb{F}[x_1, \ldots, x_n] \) (or the colon ideal) to recognize and prevent reductions to zero, thus providing us with additional criteria. We define the \( F \)-syzygy submodule of \( R^m \) to be

\[
\{(u_1, \ldots, u_m) \in R : f_1 u_1 + \cdots + f_m u_m = 0\} \subset R^m.
\]

Every S-polynomial that reduces to zero will provide another syzygy, and in turn, will prevent the unnecessary reduction to zero of other S-polynomials.

Finally, we comment on the non-uniqueness of Gröbner bases for a given ideal. Returning to example 1.8, we found \( \{xy - 1, x - y, y^2 - 1\} \) to be a Gröbner basis. But then \( \{xy - 1, 2x - 2y, 3y^2 - 3\} \) is also a Gröbner basis for the same ideal. Let \( G \) be a Gröbner basis. If the elements \( g \in G \) are all fully top-reduced with respect to \( G \setminus \{g\} \), then \( G \) is called a **minimal** Gröbner basis. A minimal Gröbner basis is not unique to an ideal, but its leading monomials (and therefore the number of generators) are unique. Also, if each generator \( g \in G \) is monic and fully reduced with respect to \( G \setminus \{g\} \), then \( G \) is called a **reduced** Gröbner basis. It is the reduced Gröbner basis that is unique to a given ideal. Furthermore, the reduced Gröbner bases for the trivial ideals are \( \{0\} \) and \( \{1\} \) depending on which trivial ideal is being represented: the zero ideal \( \langle 0 \rangle \) or the entire ring \( \langle 1 \rangle \).
1.2 Applications

Gröbner bases are quite pervasive throughout computational algebra. We have seen that reduction by a Gröbner basis is a well defined operation, and therefore solves many of the following problems. For details on any particular problem or many other applications, see [10].

Calculations in a quotient ring. Given an ideal \( I \subset R = \mathbb{F}[x_1, \ldots, x_n] \), perform calculations in \( R/I \). This is solved trivially by applying Gröbner bases. One only has to calculate a Gröbner basis for the ideal \( I \) and use it to find unique representatives for the elements \( R/I \). Thus, the results from calculations performed in \( R/I \) have canonical representatives. In fact, this also solves the ideal membership problem.

The ideal membership problem. Let \( f \in \mathbb{F}[x_1, \ldots, x_n] \) and \( I \subset \mathbb{F}[x_1, \ldots, x_n] \) an ideal. Is \( f \) in the ideal \( I \)? The solution to the quotient ring problem solves problem the ideal membership problem as well. If \( G \) is a Gröbner basis for \( I \), then the normal form of \( f \) with respect to \( G \) is zero if and only if \( f \in I \). Gröbner bases also tell us how to compare ideals.

Determining whether two ideals are equal. Given two sets of polynomials \( \{f_1, \ldots, f_m\} \) and \( \{h_1, \ldots, h_k\} \subset R = \mathbb{F}[x_1, \ldots, x_n] \), do they generate the same ideals over \( R \)? Is it the case that \( \langle f_1, \ldots, f_m \rangle = \langle h_1, \ldots, h_k \rangle \)? This problem is solved by calculating reduced Gröbner basis for each set of generators. As reduced Gröbner bases are unique for an ideal, the reduced Gröbner bases will match exactly if and only if the ideals they represent are equal. Thus Gröbner bases are quite important in computer algebra as they give us systematic ways of calculating sums of ideals, products of ideals, and intersections of ideals. The interested reader is encouraged to visit [10] for more details.

Some of the most important applications of Gröbner bases comes from solving systems of polynomial equations. In much the same way that Gauss elimination is used to triangularize a system of linear equations, Gröbner basis will triangularize a system of polynomial equations.

Deciding whether a system of polynomial equations has a solution. Given a system of polynomial equations \( f_1 = 0, f_2 = 0, \ldots, f_m = 0 \) with each \( f_i \in \mathbb{F}[x_1, \ldots, x_n], 1 \leq i \leq m \), does the system have a common solution? The solution to this problem is equivalent to calculating a reduced Gröbner basis for \( \langle f_1, \ldots, f_m \rangle \). If \( G = \{g_1, \ldots, g_k\} \) is a reduced Gröbner basis for \( \langle f_1, \ldots, f_m \rangle \), then \( g_1 = 0, \ldots, g_k = 0 \) is an equivalent system of polynomial equations. As \( G \) is unique, then \( f_1 = 0, \ldots, f_m = 0 \) has a common solution in \( \mathbb{F} \) if and only if \( G \neq \{1\} \). Also, all of \( \mathbb{F}^n \) satisfies the system if and only if \( G = \{0\} \).
Solving a system of polynomial equations. Solve the system of polynomial equations $f_1 = 0, f_2 = 0, \ldots, f_m = 0$ with each $f_i \in \mathbb{F}[x_1, \ldots, x_n]$, $1 \leq i \leq m$. In fact, Gröbner basis can be used for a lot more than simply determining whether a system has solutions. It can be used to triangularize a system in preparation for back-substitution. Suppose we have $f_1, \ldots, f_m \in \mathbb{F}[x_1, \ldots, x_n]$ providing a system of polynomial equations to be solved. We are going to use the lexicographic term ordering with $x_1 \prec x_2 \prec \cdots \prec x_n$. Let $\{g_1, \ldots, g_k\}$ be a Gröbner basis for $\langle f_1, \ldots, f_m \rangle$ ordered such that $\text{lm}(g_i) \prec \text{lm}(g_j)$ if and only if $i < j$. Thus, $g_1$ has the smallest leading monomial of the basis and involves the fewest number of the variables. Solving $g_1$ will give a set of partial solutions that can be substituted into $g_2$. Repeating in this fashion will provide all solutions to the system of equations.

The elimination theorem [10] tell us that this is the best that can be done in the following sense. Using the setup above, a Gröbner basis will contain a polynomial in only $x_1, \ldots, x_k$ if and only if any there exists any polynomial in $I = \langle f_1, \ldots, f_m \rangle \cap \mathbb{F}[x_1, \ldots, x_k]$ having fewer than all of $\mathbb{F}^k$ as solutions. In general, suppose $G$ is a Gröbner basis for $I$ under an elimination order such that any non-constant monomial in $x_{k+1}, \ldots, x_n$ is larger than any monomial in $x_1, \ldots, x_n$. Then

$$\langle G \cap \mathbb{F}[x_1, \ldots, x_k] \rangle = I \cap \mathbb{F}[x_1, \ldots, x_k].$$

In particular, $G \cap \mathbb{F}[x_1, \ldots, x_k]$ is a Gröbner basis for $I \cap \mathbb{F}[x_1, \ldots, x_k]$. Finally, it may be the case that certain variable orderings, $x_{\sigma(1)} \prec x_{\sigma(2)} \prec \cdots \prec x_{\sigma(n)}$ for some $\sigma \in S_n$, work better than others. Hopefully, the situation providing the system of equations will help shed some light on which order to use. One might choose the most “independent” of the variables to be the smallest. But in general, it might be a good idea to try a few randomly chosen orders, $\sigma \in S_n$, and use the one that provides the most triangularized system. This, however, is easier said than done. The calculation of a Gröbner basis is usually the most computationally difficult step in solving a system of equations. For example, the basis calculated in section 1.3 has an exponential (in the number of variables) number of generators in a reduced Gröbner basis.

Finally, modern cryptosystems are a fantastic source of polynomial systems that are too large for the calculation of a Gröbner basis. Known plaintext attacks of modern cryptosystems. Given a large enough plaintext/ciphertext pair from AES, what key was used to generate it? Given the plaintext and key, each ciphertext bit can be described using a polynomial over $\mathbb{F}_2$ in the plaintext.
bits and key bits. See figure 1.2 for the general flow of information. Thus, one could write a system of
equations linking the plaintext bits $x_1, \ldots, x_{128}$, ciphertext bits $y_1, \ldots, y_{128}$, and key bits $k_1, \ldots, k_{128}$ (or 192 or 256). Substituting the known plaintext and ciphertext gives a system of equations for
the key bits, and one has “only” to solve the system of polynomial equations to recover the key
bits. Such Gröbner based known-plaintext attacks on cryptosystems are as methodical as they are
impractical.

1.3 Examples of Large Gröbner Basis Calculations

Gröbner bases can be very large, even for ideals with few generators. Sections 1.3 provides
an example of a large Gröbner basis. Various attempts have been made to bound the number of
generators, the degrees of the generators, and the degrees of coefficients [10, 24]. However, the size
of a Gröbner basis does not always indicate the extent of the computations required to arrive at
such a basis. Section 1.3.2 describes a situation where calculating a Gröbner basis for a trivial ideal
involves polynomials of very large degree.
1.3.1 Exponential Number of Generators

For an example of a Gröbner basis with an exponential number of generators (in the number of variables), we explicitly determine the reduced Gröbner basis of the following ideal

\[ \langle f, (x^b_i)^2 + x^b_i, 1 \leq i \leq n, b_i \in \{0,1\} \rangle \subset \mathbb{F}_2[x_0^1, x_1^1, x_2^1, \ldots, x_n^1], \]

where \( f = x_0^0 x_1^1 + \cdots + x_n^0 x_n^1 \) for any \( n \in \mathbb{N} \). Such a result could be useful for computing Gröbner bases for large systems of quadratic polynomials over \( \mathbb{F}_2 \), arising from cryptosystems and other applications. First, define

\[ F = \left\{ f, (x^b_i)^2 + x^b_i, 1 \leq i \leq n, b_i \in \{0,1\} \right\} \quad \text{and for } 0 \leq m < n, \]

\[ G_m = F \bigcup \limits_{k=1}^m \left\{ (x^b_1 + 1) \cdots (x^b_k + 1)(x^0_{k+1} x^1_{k+1} + \cdots + x^0_n x^1_n), b_i \in \{0,1\} \right\}. \]

Claim. \( G_{n-1} \) is a basis for \( \langle F \rangle \) under lex order with \( x_0^0 > x_1^0 > x_2^0 > \cdots > x_n^0 > x_n^1 \).

Proof. Clearly, \( \langle F \rangle \subseteq \langle G_{n-1} \rangle \). To get the reverse inclusion, we show the chain of containments \( \langle F \rangle \supseteq \langle G_0 \rangle \supseteq \langle G_1 \rangle \supseteq \cdots \supseteq \langle G_{n-1} \rangle \). The first containment is trivial as \( F = G_0 \) by definition. We show that whenever \( 1 < k < n \), it must be the case that \( f_k \in \langle G_{k-1} \rangle \) for any

\[ f_k = (x^b_1 + 1) \cdots (x^b_k + 1)(x^0_{k+1} x^1_{k+1} + \cdots + x^0_n x^1_n) \in G_k \setminus G_{k-1}. \]

Set \( f_{k-1} = (x^b_1 + 1) \cdots (x^b_{k-1} + 1)(x^0_k x^1_k + \cdots + x^0_n x^1_n) \in G_{k-1} \). We define \( g \) to be the S-polynomial of \( f_{k-1} \) and \( h(x^b_k) = (x^b_k)^2 + x^b_k \). We relax the notion of an S-polynomial slightly, by the nature of lemma 1.9, in that the coefficients of \( f_{k-1} \) and \( h(x^b_k) \) are polynomials rather than monomials, but the leading terms of those coefficients are what the coefficients of the traditional S-polynomial
would be.

\[ g = S\text{-poly} \left( f_{k-1}, h \left( x_k^{b_k} \right) \right) \]
\[ = (x_1^{b_1} + 1) \cdots (x_k^{b_k} + 1) \left[ x_k^{b_k} (x_k^{0})^1 + \cdots + x_n^{0,1} \right] + x_k^{1-b_k} \left( x_k^{b_k} \right)^2 + x_k^{b_k} \]
\[ = (x_1^{b_1} + 1) \cdots (x_k^{b_k} + 1) \left[ x_k^{b_k} (x_k^{0})^1 + \cdots + x_n^{0,1} \right] \]
\[ + \left( x_k^{b_k} \right)^2 + x_k^{0,1} \]
\[ = (x_1^{b_1} + 1) \cdots (x_k^{b_k} + 1) \left[ x_k^{b_k} (x_k^{0})^1 + \cdots + x_n^{0,1} \right] + x_k^{0,1}. \]

As \( f_{k-1}, g \in \langle G_{k-1} \rangle \), we add them (this is exactly one top-reduction) to get

\[ f_{k-1} + g = (x_1^{b_1} + 1) \cdots (x_k^{b_k} + 1) \left[ x_k^{b_k} (x_k^{0})^1 + \cdots + x_n^{0,1} \right] + x_k^{0,1} \]
\[ + x_k^{0,1} \]
\[ = (x_1^{b_1} + 1) \cdots (x_k^{b_k} + 1) (x_k^{0})^1 + \cdots + x_n^{0,1} = f_k \in \langle G_{k-1} \rangle. \]

Thus \( \langle G_k \rangle \subseteq \langle G_{k-1} \rangle \) and \( G_{n-1} \) is a basis for \( \langle F \rangle \).

**Claim.** \( G_{n-1} \) is a reduced Gröbner basis for \( \langle F \rangle \).

**Proof.** We use Buchberger’s criterion. First we consider generalized S-polynomials of the form:

\[ S\text{-poly} \left( (x_1^{b_1} + 1) \cdots (x_m^{b_m} + 1) (x_{m+1}^{0})^1 + \cdots + x_n^{0,1} , \left( x_k^{b_k} \right)^2 + x_k^{b_k} \right). \]

We know that these reduce to zero when \( k = m + 1 < n \) from the proof above. When \( k = m + 1 = n \) and \( c \in \{ 0, 1 \} \), we have

\[ S\text{-poly} \left( (x_1^{b_1} + 1) \cdots (x_n^{b_n} + 1) (x_n^{c})^1 + x_n^{c} \right) \]
\[ = (x_1^{b_1} + 1) \cdots (x_n^{b_n} + 1) \left[ (x_n^{c})^2 x_n^{1-c} + (x_n^{c})^2 x_n^{1-c} + x_n^{0,1} \right]. \]

It is important to note at this point that these generalized S-polynomials have the same leading term (after cancellation) as the standard S-polynomials. Therefore, by lemma 1.9, there is no difficulty in using generalized S-polynomials in Buchberger’s algorithm as long as we continue to show that
the leading terms are the same as those in traditional S-polynomials. When \( m + 1 < k \leq n \), the product criterion 1.10 says it reduces to zero. Consider the case when \( k < m + 1 < n \).

\[
\text{S-poly} \left( (x_1^{b_1} + 1) \cdots (x_m^{b_m} + 1)(x_{m+1}^0x_{m+1}^1 + \cdots + x_n^0x_n^1), \left( x_k^{b_k} \right)^2 + x_k^{b_k} \right) = \left( x_k^{b_k} \right)^2 + x_k^{b_k}.
\]

which is a multiple of a field equation, \( x_k^{b_k} + x_k^{b_k} \). If \( k < m + 1 = n \), the above still works as all the extra stuff being added is still smaller than the leading monomial of a standard S-polynomial. In this case, everything cancels inside the square brackets. For any \( k < m + 1 \), we can flip \( b_k \) to \( 1 - b_k \), and in considering

\[
\text{S-poly} \left( (x_1^{b_1} + 1) \cdots (x_m^{b_m} + 1)(x_{m+1}^0x_{m+1}^1 + \cdots + x_n^0x_n^1), \left( x_k^{1-b_k} \right)^2 + x_k^{1-b_k} \right),
\]

we can again use the product criterion 1.10 as the leading monomials are relatively prime.

Next, we consider S-polynomials of the form \( \text{S-poly}(f, g) \) where

\[
f = (x_1^{b_1} + 1) \cdots (x_m^{b_m} + 1)(x_{m+1}^0x_{m+1}^1 + \cdots + x_n^0x_n^1) \quad \text{and} \quad g = (x_1^{c_1} + 1) \cdots (x_k^{c_k} + 1)(x_{k+1}^0x_{k+1}^1 + \cdots + x_n^0x_n^1).
\]

We must now begin to be careful with generalized S-polynomials. If \( k = m \), \( f \neq g \) and \( 1 \leq j \leq k \) is the largest index for which \( c_j \neq b_j \), set

\[
h = \text{lcm} \left\{ \frac{(x_1^{b_1} + 1) \cdots (x_k^{b_k} + 1)}{(x_j^{b_j} + 1)}, \frac{(x_1^{c_1} + 1) \cdots (x_k^{c_k} + 1)}{(x_j^{c_j} + 1)} \right\}.
\]
Then

\[
S\text{-poly}(f, g) = h \left[ x_j^0(x_j^0 + 1) + x_j^1(x_j^0 + 1) \right] \left( x_{k+1}^0x_{k+1}^1 + \cdots + x_n^0x_n^1 \right) \quad \text{(before cancel)}
\]

\[
= h \left[ x_j^0 + x_j^1 \right] \left( x_{k+1}^0x_{k+1}^1 + \cdots + x_n^0x_n^1 \right) \quad \text{(after cancellation)}
\]

\[
= h \left[ (x_j^0 + 1) + (x_j^1 + 1) \right] \left( x_{k+1}^0x_{k+1}^1 + \cdots + x_n^0x_n^1 \right)
\]

is a generalized S-polynomial with the same leading term as a standard S-polynomial, and it is clearly reducible to zero modulo \( G_{n-1} \). If \( k < m \), we set

\[
h = \text{lcm} \left\{ \frac{(x_1^b + 1) \cdots (x_m^b + 1)}{(x_{k+1}^0 + 1)}, (x_1^c + 1) \cdots (x_k^c + 1) \right\}.
\]

Before proceeding, it is helpful to see the following reduction. If we think of \( hx_{m+i}^0x_{m+i}^1(x_{k+1}^0x_{k+1}^1 + \cdots + x_n^0x_n^1) \) as multiples of basis elements for \( 2 \leq i \leq n \), then \( h \cdot x_{k+1}^0x_{k+1}^1(x_{m+2}^0x_{m+2}^1 + \cdots + x_n^0x_n^1) \) when top-reduced by \( h(x_{k+1}^0x_{k+1}^1 + \cdots + x_n^0x_n^1)(x_{m+2}^0x_{m+2}^1 + \cdots + x_n^0x_n^1) \) yields \( h \cdot (x_{k+2}^0x_{k+2}^1 + \cdots + x_n^0x_n^1) \). It can be shown (straightforward but messy) that in all cases, the following generalized S-polynomial has the same leading term as the standard S-polynomial. All
equivaleces are reductions modulo $G_{n-1}$.

$$ S\text{-poly}(f, g) = h \left[ (x_{k+1}^0 + 1)(x_{k+1}^0 + 1)(x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) 
\quad + x_{m+1}^0 x_{m+1}^1 (x_{k+1}^0 x_{k+1}^1 + \cdots + x_n^0 x_n^1) \right] $$

$$ = h \left[ x_{k+1}^0 x_{k+1}^0 x_{m+1}^0 x_{m+1}^1 + x_{k+1}^0 x_{k+1}^1 (x_{m+2}^0 x_{m+2}^1 + \cdots + x_n^0 x_n^1) 
\quad + (x_{k+1}^0 + 1 + x_{k+1}^1) (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) 
\quad + (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) 
\quad + x_{m+1}^0 x_{m+1} x_{k+1}^1 + x_{m+1}^0 x_{m+1} (x_{k+2}^0 x_{k+2}^1 + \cdots + x_n^0 x_n^1) \right] $$

$$ \equiv h \left[ (x_{k+2}^1 x_{k+2}^1 + \cdots + x_n^0 x_n^1) (x_{m+2}^0 x_{m+2}^1 + \cdots + x_n^0 x_n^1) 
\quad + (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) + x_{m+1}^0 x_{m+1} (x_{k+2}^0 x_{k+2}^1 + \cdots + x_n^0 x_n^1) \right] $$

$$ \equiv h \left[ x_{k+2}^1 x_{k+2}^1 + \cdots + x_n^0 x_n^1 (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) 
\quad + (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) \right] $$

$$ \equiv h \left[ (x_{k+2}^1 x_{k+2}^1 + \cdots + x_n^0 x_n^1) (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) 
\quad + (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) \right] $$

$$ \equiv h \left[ (x_{k+2}^1 x_{k+2}^1 + \cdots + x_n^0 x_n^1) (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) 
\quad + (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) \right] $$

$$ \equiv h \left[ (x_{k+2}^1 x_{k+2}^1 + \cdots + x_n^0 x_n^1) (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) 
\quad + (x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1) \right] $$

Thus Buchberger’s criterion (lemma 1.9) is satisfied. Finally, $G_{n-1}$ must be a reduced Gröbner basis since all leading terms are of the form

$$ x_{k+1}^0 x_{k+1}^1 \prod_{j=1}^{k} x_j^{b_j}, \quad (1.1) $$

14
and all terms of any generator \((x_1^{b_1} + 1) \cdots (x_m^{b_m} + 1)(x_{m+1}^0 x_{m+1}^1 + \cdots + x_n^0 x_n^1)\) are of the following form for any \(S \subseteq \{1, \ldots, m\}\) and \(m + 1 \leq i \leq n\).

\[
x_i^0 x_i^1 \prod_{j \in S} x_j^{c_j}. \tag{1.2}
\]

In order for (1.1) to divide (1.2), we must have that \(i = k + 1\), \(S = \{1, \ldots, k\}\), and \(c_i = b_i\) for all \(1 \leq i \leq k\). But this happens exactly when (1.2) is the leading monomial of \((x_1^{b_1} + 1) \cdots (x_k^{b_k} + 1)(x_{k+1}^0 x_{k+1}^1 + \cdots + x_n^0 x_n^1)\).

It is interesting to note that while \(x_1^0 x_1^1 + \cdots + x_n^0 x_n^1\) is interpreted in the most simple setting one could imagine: the binary field in the presence of field equations, this single generator (or \(2n + 1\) generators if you count the field equations) has a reduced Gröbner basis of \(O(2^n)\) generators. Thus, calculating a reduced Gröbner basis can involve processing an exponential (in the number of variables and equations) number of S-polynomials.

### 1.3.2 Exponential Degree of Signatures

For an example (due to Masser, Philippon, Lazard and Mora) of a Gröbner basis requiring an exponential degree of the coefficients (very larger signatures as they are called in later chapters), we consider the system over \(R = \mathbb{F}[x_1, \ldots, x_n]\) below.

\[
\begin{align*}
f_1 &= x_1^d \\
f_2 &= x_1 - x_2^d \\
f_3 &= x_2 - x_3^d \\
& \vdots \\
f_{n-1} &= x_{n-2} - x_{n-1}^d \\
f_n &= 1 - x_n - x_n^{d-1}
\end{align*} \tag{1.3}
\]

The above system has no common zeros. To see this, we set the first function in system (1.3) equal to zero to find that \(x_1 = 0\). The second function similarly gives \(x_2 = 0\). Iterating, we find that \(x_1 = x_2 = \cdots = x_{n-1} = 0\). Thus, it is impossible for the final function in (1.3) to have a common zero. Without any common zeros, the ideal \(\langle f_1, \ldots, f_n \rangle\) equals \(\langle 1 \rangle = R\). Therefore, there must exist
polynomials \( g_1, \ldots, g_n \in R \) such that

\[ g_1 \cdot f_1 + \cdots + g_n \cdot f_n = 1. \tag{1.4} \]

To show that the degree of \( g_1 \) must be very large, we momentarily view \( f_1, \ldots, f_n \) as elements in a larger ring \( \mathbb{F}(t)[x_1, \ldots, x_n] \) for some variable \( t \). After restricting \( x_1, \ldots, x_n \) in terms of \( t \neq 0 \) as in the following

\[ x(t) = (x_1, \ldots, x_n) = \left( t^{d^n-2(d-1)}, t^{d^n-3(d-1)}, \ldots, t^{d^2(d-1)}, t^{d^1(d-1)}, t^{d^0(d-1)}, \frac{1}{t} \right), \]

we are in a place to obtain a lower bound on the degree of \( g_1 \). First, notice that \( f_i(x(t)) = 0 \) for every \( 2 \leq i \leq n \). Thus the \( R \)-linear combination in (1.4) collapses to

\[ g_1(x(t))f_1(x(t)) = g_1(x(t)) \left( t^{d^n-2(d-1)} \right)^d = 1. \]

Finally, we obtain a lower bound on the degree of \( g_1 \) as

\[ \deg_{x_n} g_1 \geq d^{n-1}(d-1) \]

which is exponential in the number of variables and the number of generators. Such an example will pose problems for Gröbner basis algorithms when the degrees of the coefficients (the \( g_i \)) increase slowly throughout the course of the algorithm. Unfortunately, even when the resulting reduced Gröbner basis is a very simple list of generators, the intermediate computations may be intractable.

### 1.4 Recent Algorithms

There has been extensive effort in developing more efficient algorithms for calculating Gröbner bases. In Buchberger’s original algorithm (1965, [3]), one has to reduce many useless S-polynomials (i.e., those that reduce to 0 via long division), and each reduction is time consuming. It is natural to avoid useless reductions as much as possible. Buchberger [4, 5] discovered two simple criteria for detecting useless S-polynomials (see lemma 1.10 for one). Note that a reduction of an S-polynomial to 0 corresponds to a syzygy (for the initial list of polynomials). Möller, Mora and
Traverso (1992, [26]) go a step further to present an algorithm using the full module of syzygies, however, their algorithm is not very efficient.

Faugère (2002, [17]) introduced the idea of signatures and rewriting rules that can detect many useless S-polynomials hence saving a significant amount of time that would be used in reducing them. In fact, for a regular sequence of polynomials, his algorithm F5 detects all useless reductions. By computer experiments, Faugère showed that his algorithm F5 is many times faster than previous algorithms. In fact, Faugère and Joux (2003, [18]) solved the first Hidden Field Equation (HFE) Cryptosystem Challenge which involves a system of 80 polynomial equations with 80 variables over the binary field (1996, [27]). Since F5 seems difficult to understand, there have been several papers trying to simplify and improve F5: see Eder and Perry (2009, [14]), Sun and Wang (2009, [28]), and Hashemi and Ars (2010, [21]). F5 has other limitations. For example, it is an incremental algorithm and requires degree compatible orderings. Furthermore, F5 is designed to only work on homogeneous polynomials. Eder and Perry in [14] created F5C, a variant of F5, that began to take advantage of F5’s incremental nature. By calculating reduced Gröbner bases at each iteration, F5C has a significant advantage over F5 in terms of runtimes and memory usage.

Algorithm G2V, see chapter 2 or [19], can be thought of as another improvement on F5. Also an incremental algorithm, it calculates a reduced Gröbner bases at each iteration. Also, in the process of calculating a Gröbner basis incrementally, the colon ideal is computed which is used to prevent subsequent reductions to zero. Although G2V is not a direct generalization of F5, its algorithmic framework is extremely similar to that of F5 and Arri’s algorithm [1, 13] with a few “policy” differences. Finally, G2V does not need to be run (strictly) incrementally (see chapter 3), does not require any particular term order, and does not require homogeneous generators.

In another direction of research, one tries to speed up the reduction step. Lazard (1983, [24]) pointed out the connection between Gröbner bases and linear algebra, that is, a Gröbner basis can be computed by Gauss elimination of a Sylvester matrix. The XL algorithm of Courtois et al. (2000, [8]) is an implementation of this Sylvester matrix, which is recently improved by Ding et al. (2008, [12]). A more clever approach is the F4 algorithm of Faugère (1999, [16]). F4 is an efficient method for reducing several S-polynomials simultaneously where the basic idea is to apply fast linear algebra methods to the submatrix of the Sylvester matrix consisting of only those rows that are needed for the reductions of a given list of S-polynomials. This method benefits from the efficiency of fast linear algebra routines. The main problem with this approach, however, is that the memory usage grows
too quickly, even for medium systems of polynomials. Finally, there are numerous heuristics and special cases to speed the computation of a Gröbner basis. Several examples are Sugar [20], slimGB [6], and PolyBoRi [2].

1.5 Organization and Contributions

In this thesis, we present several new algorithms, namely G2V, GVW, and its improvement. These algorithms were developed at different stages of this thesis, and each subsequent algorithm improves upon and supersedes previous ones. The final algorithm, an improved GVW or GVWHS, is the culmination of this thesis. It provides a theoretical foundation for computing Gröbner bases and minimal Gröbner bases for syzygy modules.

Chapters 2 and 3 describe the G2V and GVW algorithms respectively. G2V uses information about the colon ideal to prevent reductions to zero. Similarly, GVW uses information about the syzygy module to prevent reductions to zero. Just as the syzygy module is a generalization of the colon ideal, GVW is a generalization of G2V in that when using the POT ordering, the GVW algorithm proceeds almost identically to G2V. Next, chapter 4 presents a much nicer framework explaining why GVW (and hence G2V) terminates in finite time and gives the correct answer. Moreover, GVWHS, the topic of chapter 4, also provides several improvements over GVW and G2V which translate into a very efficient algorithm. Finally, chapter 5 discusses implementation details. As the correctness of GVWHS is firmly in place, many implementation details are left to the imagination. This final chapter helps answer some of the remaining questions, and gives the details we used in our implementations for the sake of clarification and reproduction of results.

Various (pseudo) code listings are included throughout to firm the exposition. Actual C++ code snippets are included where relevant in chapter 5, but for the sake of completeness, entire programs are included in the appendices. First, as C++ code timings and statistics were listed in chapters 3 and 4, C++ source code is included in appendix A for algorithm GVWHS in figure 4.1. Also, the relevant parts of the binary field equation version of GVW, described in section 5.4, are included in appendix B. Finally, as we used implementations of G2V and GVW in Singular CAS, their source code listings are included in the appendix C.
Chapter 2

The G2V Algorithm

The following is joint work with Shuhong Gao and Yinhua Guan. It was published as [19] and presented at ISSAC2010 in Munich Germany.

2.1 Introduction

Our main purpose of the current chapter is to present a new algorithm that is both simpler and more efficient than F5 and F5C. Our algorithm is incremental just like F5 and F5C. Let \(\mathbb{F}\) be any field and \(\mathcal{R} = \mathbb{F}[x_1, \cdots, x_n]\). Fix an arbitrary monomial order on \(\mathcal{R}\). At a typical iterative step, a Gröbner basis \(\mathcal{G}\) for an ideal \(\mathcal{I}\) in \(\mathcal{R}\) is already computed, and it is desired to compute a Gröbner basis for the new ideal \(\langle \mathcal{I}, g \rangle\) for a given polynomial \(g \in \mathcal{R}\). In F5, the basis \(\mathcal{G}\) may not be reduced, thus containing many redundant polynomials. F5C is the same as F5 except that \(\mathcal{G}\) is replaced by a reduced Gröbner basis in the next iterative step. Our algorithm will use a reduced Gröbner basis \(\mathcal{G}\) as in F5C, but the crucial difference is that we introduce a so-called “super top-reduction” to detect “useless” polynomials. Furthermore, if there happens to be a polynomial that reduces to 0, it will be used to detect more useless polynomials. Hence reduction to 0 in our algorithm is not useless at all. In fact, it gives us a polynomial in the colon ideal

\[
\langle \mathcal{I}, g \rangle = \{ u \in \mathcal{R} : ug \in \mathcal{I} \}. \tag{2.1}
\]
It is of independent interest to have an efficient algorithm for computing Gröbner bases for colon ideals of the form \((I : g)\), as it is a routine repeatedly used in primary decomposition, especially in separating components of different dimensions.

In Section 2.2, we shall present a relation between the Gröbner bases of \(\langle I, g \rangle\) and \((I : g)\). This is based on the exact sequence of \(R\)-modules:

\[
0 \to R/(I : g) \to R/I \to R/\langle I, g \rangle \to 0
\]

where the second morphism is defined by multiplication by \(g\), which is injective by the definition in (2.1), and the third is the canonical morphism. The exactness of the sequence implies that

\[
\dim_F(R/I) = \dim_F(R/\langle I, g \rangle) + \dim_F(R/(I : g)). \tag{2.2}
\]

For an arbitrary ideal \(I\), we show in Section 2.2 how to compute \(F\)-linear bases for all of these vector spaces from a given Gröbner basis for \(I\). In particular, we have the following result.

**Theorem 2.1.** Suppose \(I\) is a zero-dimensional ideal in \(R = F[x_1, \ldots, x_n]\). Let \(N = \dim_F(R/I)\) (which is equal to the number of common solutions of \(I\) over the algebraic closure of \(F\), counting multiplicities). Then, given a Gröbner basis for \(I\) (under any monomial order) and a polynomial \(g \in R\), Gröbner bases for \(\langle I, g \rangle\) and \((I : g)\) can be computed deterministically using \(O((nN)^3)\) operations in \(F\).

The time complexity claimed by the theorem is of interest only when \(N\) is small compared to \(n\) (say \(N = n^{O(1)}\)). For when \(N\) is large or \(\infty\), we introduce an enhanced algorithm in Section 2.3. We shall define regular top-reductions and super top-reductions, as well as J-polynomials and J-signatures for any pair of polynomials. A J-polynomial means the joint of two polynomials, which is different from an S-polynomial but plays a similar role. Our algorithm is very similar to Buchberger’s algorithm, where we replace S-polynomials by J-polynomials and “reduction” by “regular top-reduction”. There are, however, two new features: (a) a super top-reduction is introduced to detect a useless J-polynomial, and (b) each reduction to zero gives a polynomial in \((I : g)\) and is subsequently used in detecting future useless J-polynomials. We have implemented the resulting algorithm in Singular. In Section 2.4, we present some comparisons with F5 and F5C. Our computer experiments on several benchmark examples show that the new algorithm is more efficient, often
two to ten times faster than F5 and F5C.

2.2 Theory

We give a computational proof for the correspondence of linear bases for the equation (2.1) and the theorem mentioned in the previous section. The proof itself is more important than the theorem for our algorithm presented in the next section.

Let $I$ be an arbitrary ideal in $R = F[x_1, \ldots, x_n]$ and $g$ any polynomial in $R$. Suppose we know a Gröbner basis $G$ for $I$ with respect to some monomial order $\prec$. Then we can find the standard monomial basis for $R/I$:

$$B(I) = \{x^{\alpha_1} = 1, x^{\alpha_2}, \ldots, x^{\alpha_N}\},$$

that is, $B(I)$ consists of all the monomials that are not reducible by $\text{lm}(I)$.\(^1\) Then $B(I)$ is a linear basis for $R/I$ over $F$. We assume the monomials in $B(I)$ are ordered in increasing order, that is, $x^{\alpha_i} \prec x^{\alpha_j}$ whenever $i < j$. Please note that when $I$ is not 0-dimensional, $N$ is $\infty$ and it is possible that there are infinitely many monomials between some two monomials in $B(I)$ (especially for lex order). The following proof is for an arbitrary ideal $I$.

Suppose

$$\begin{pmatrix}
  x^{\alpha_1} \\
  x^{\alpha_2} \\
  \vdots \\
  x^{\alpha_N}
\end{pmatrix} \cdot g \equiv \begin{pmatrix}
  h_1(x) \\
  h_2(x) \\
  \vdots \\
  h_N(x)
\end{pmatrix} \pmod{G} \tag{2.3}
$$

$$= A(x^{\alpha_1}, x^{\alpha_2}, \ldots, x^{\alpha_N})^T, \tag{2.4}
$$

where $h_i \in \text{span}_F(x^{\alpha_1}, \ldots, x^{\alpha_N})$, $1 \leq i \leq N$, that is, each $h_i$ is the normal form of $x^{\alpha_i} \cdot g \mod G$, and $A \in \mathbb{F}^{N \times N}$ is a matrix with the $i^{th}$ row representing the coefficients of $h_i$, $1 \leq i \leq N$.

Note the matrix $A$ in (2.4) has an important property that is useful for finding points (or solutions) of the algebraic variety defined by the ideal $I$. In fact, when $I$ is zero-dimensional, the eigenvalues of $A$ correspond to the values of the polynomial $g$ when evaluated at the points in the variety of $I$ (and the corresponding eigenvectors are determined by the points alone, independent of

---

\(^1\)We say that a polynomial $f$ is reducible by a set of polynomials $G$ if $\text{lm}(f)$ is divisible by $\text{lm}(g)$ for some $g \in G$.\)
Now apply the following row operations to both sides of (2.3) (equivalently (2.4)):

(R1) for $1 \leq i < j \leq N$ and $a \in \mathbb{F}$, subtract from the $j^{th}$ row by the $i^{th}$ row multiplied by $a$ (i.e. $A_j := A_j - aA_i$),

(R2) for $a \in \mathbb{F}$ with $a \neq 0$, multiply the $i^{th}$ row by $a$.

This means that we only apply row operations downward as one would perform Gauss elimination (to equation (2.4)) to get a triangular matrix. For example, suppose $x^\beta$ is the leading monomial of $h_1(x)$. We can use $h_1(x)$ to eliminate the term $x^\beta$ in all $h_j(x)$, $2 \leq j \leq N$. In fact, we only need to eliminate it if it is the leading term. Then continue with the leading monomial of the resulting $h_2(x)$ and so on. Since a monomial order is a well ordering, there is no infinite decreasing sequence of monomials, hence each $h_i(x)$ needs only be reduced by finitely many rows above it (even if there are infinitely many rows about the row of $h_i(x)$). Therefore, using downward row operations, the right hand side of (2.3) can be transformed into a quasi-triangular form, say

$$
\begin{pmatrix}
  u_1(x) \\
  u_2(x) \\
  \vdots \\
  u_N(x)
\end{pmatrix}
\cdot g 
\equiv
\begin{pmatrix}
  v_1(x) \\
  v_2(x) \\
  \vdots \\
  v_N(x)
\end{pmatrix}
\pmod G,
$$

(2.5)

where $u_i(x)$ and $v_i(x)$ are in $\text{span}_{\mathbb{F}}(x^{\alpha_1}, \ldots, x^{\alpha_N})$, and for each $1 \leq i, j \leq N$ with $v_i(x) \neq 0$ and $v_j(x) \neq 0$, we have $\text{lm}(v_i(x)) \neq \text{lm}(v_j(x))$, i.e. the nonzero rows of the right hand side have distinct leading monomials.

Since row operations are downward only, and the $B(I)$ are written in increasing order, we have that each $u_i(x)$ is monic and

$$\text{lm}(u_i(x)) = x^{\alpha_i}, \ 1 \leq i \leq N.$$

Let

$$G_0 = G \cup \{u_i(x) : 1 \leq i \leq N \text{ with } v_i(x) = 0\}, \text{ and}$$

$$G_1 = G \cup \{v_i(x) : 1 \leq i \leq N\}.$$
Certainly, \( G_1 \subseteq (I, g) \) and \( G_0 \subseteq (I : g) \) (as \( u_i(x) \cdot g \in I \) whenever \( v_i(x) = 0 \)). We prove the following:

(a) \( G_0 \) is a Gröbner basis for \((I : g)\), and

(b) \( G_1 \) is a Gröbner basis for \((I, g)\).

Since (2.5) is obtained from (2.3) by downward row operations, there is an upper triangular non-
singular matrix \( M \in \mathbb{F}^{N \times N} \) (with each row containing only finitely many nonzero entries) such that

\[
(u_1(x), \ldots, u_N(x))^T = M (x^{\alpha_1}, \ldots, x^{\alpha_N})^T,
\]

and

\[
(v_1(x), \ldots, v_N(x))^T = M (h_1(x), \ldots, h_N(x))^T.
\]

Even though \( N \) could be infinite, \( M \) does have an inverse \( M^{-1} \) with each row containing only finitely many nonzero entries. For any \( w(x) \in R/I \), we can write it as

\[
w(x) = \sum_{i=1}^{N} w_i x^{\alpha_i}, \ w_i \in \mathbb{F}, \tag{2.6}
\]

where there are only finitely many nonzero \( w_i \)'s. Let

\[
(c_1, \ldots, c_N) = (w_1, \ldots, w_N)M^{-1} \in \mathbb{F}^N.
\]

Note that the vector \((c_1, \ldots, c_N)\) contains only finitely many nonzero entries, as it is a linear com-
bination of finitely many rows of \( M^{-1} \). Then we have

\[
w(x) = (w_1, \ldots, w_N)M^{-1}M(x^{\alpha_1}, \ldots, x^{\alpha_N})^T
\]

\[
= (c_1, \ldots, c_N)(u_1(x), \ldots, u_N(x))^T,
\]

i.e.

\[
w(x) = \sum_{i=1}^{N} c_i u_i(x), \tag{2.7}
\]

23
and
\[
  w(x) \cdot g = (w_1, \ldots, w_N)(x^{\alpha_1}, \ldots, x^{\alpha_N})^T \cdot g
  \equiv (w_1, \ldots, w_N)M^{-1}M(h_1(x), \ldots, h_N(x))^T
  = (c_1, \ldots, c_N)(v_1(x), \ldots, v_N(x))^T,
\]
i.e.
\[
  w(x) \cdot g \equiv \sum_{i=1}^N c_i v_i(x) \pmod{G}. \tag{2.8}
\]

For (a), to prove that $G_0$ is a Gröbner basis for $(I : g)$, it suffices to show that each $f \in (I : g)$ can be reduced to zero by $G_0$ via long division. Indeed, for any $f \in (I : g)$, since $G$ is a Gröbner basis, $f$ can be reduced by $G$ to some $w(x)$ as in (2.6). Then, by (2.7) and (2.8), we have
\[
f \equiv w(x) \equiv \sum_{i=1}^N c_i u_i(x) \pmod{G},
\]
and
\[
f \cdot g \equiv w \cdot g \equiv \sum_{i=1}^N c_i v_i(x) \pmod{G}.
\]
As $f \in (I : g)$, we have $f \cdot g \in I$, so $f \cdot g \equiv 0 \pmod{G}$. This implies that $\sum_{i=1}^N c_i v_i(x) = 0$, hence $c_i = 0$ whenever $v_i(x) \neq 0$, as the nonzero $v_i(x)$’s have distinct leading monomials. Thus
\[
f \equiv w(x) \equiv \sum_{u_i \in G_0} c_i u_i(x) \pmod{G}. \tag{2.9}
\]

This implies that $f$ can be reduced to 0 by $G_0$ via long division. Therefore, $G_0$ is a Gröbner basis for $(I : g)$.

For (b), for any $f \in (I, g)$, there exists $w(x)$ of the form (2.6) such that
\[
f \equiv w(x) \cdot g \pmod{G}.
\]
By (2.8),
\[
f \equiv w(x) \cdot g \equiv \sum_{i=1}^N c_i v_i(x) = \sum_{v_i(x) \neq 0} c_i v_i(x) \pmod{G}. \tag{2.10}
\]
Hence $f$ can be reduced to 0 by $G \cup \{v_i(x) : 1 \leq i \leq N\}$ via long division. This shows that $G_1$ is a
Gröbner basis for \((I, g)\).

Now we explicitly describe \(B(I : g)\) and \(B(\langle I, g \rangle)\), the standard monomial bases for \(R/(I : g)\) and \(R/(I, g)\), respectively. We first show that

\[
B(I : g) = \{x^{\alpha_j} : 1 \leq j \leq N \text{ and } v_j(x) \neq 0\}. \tag{2.11}
\]

Since \(I \subseteq (I : g)\), we have

\[
B(I : g) \subseteq B(I) = \{x^{\alpha_1}, \ldots, x^{\alpha_N}\}.
\]

Recall that \(\text{lm}(u_j(x)) = x^{\alpha_j}, 1 \leq j \leq N\). For each \(1 \leq j \leq N\), if \(v_j(x) = 0\), then \(u_j(x) \in G_0\), so \(x^{\alpha_j} \notin B(I : g)\). If \(v_j(x) \neq 0\), we claim that there is no \(f \in (I : g)\) such that \(\text{LM}(f) = x^{\alpha_j}\). Suppose otherwise. Then \(f \equiv w(x) \pmod{G}\) for some \(w(x)\) as in (2.6) and \(\text{LM}(w(x)) = \text{LM}(f) = x^{\alpha_j}\). By (2.9), \(x^{\alpha_j}\) must be equal to the leading monomial of some \(u_i(x) \in G_0\), hence \(u_j(x) \in G_0\). This contradicts the assumption that \(v_j(x) \neq 0\). Hence (2.11) holds.

Next we claim that

\[
B(\langle I, g \rangle) = B(I) \setminus \{\text{LM}(v_i(x)) : 1 \leq i \leq N\}. \tag{2.12}
\]

This holds, as the equation (2.10) implies that the leading monomial of any \(f \in \langle I, g \rangle\) is either divisible by \(\text{lm}(G)\) or equal to some \(\text{lm}(v_i(x))\), where \(v_i(x) \neq 0, 1 \leq i \leq N\).

Now back to the proof of the theorem. The equation (2.2) follows from the equations (2.11) and (2.12), as the leading monomials of the nonzero \(v_i(x)\) are distinct and are contained in \(B(I)\). When \(I\) is zero-dimensional, the normal forms \(h_i(x)\) in (2.3) can be computed in time cubic in \(nN\), say by using the border basis technique [23], and Gauss elimination also needs cubic time. Hence the claimed time complexity follows.

Finally, we make a few observations concerning the above proof. They will be the basis for our algorithm below.

- \(\text{lm}(u_i(x)) = x^{\alpha_i}\), so \(u_i\) is not divisible by \(\text{lm}(G)\), for all \(1 \leq i \leq N\). The monomial \(x^{\alpha_i}\) is an index for the corresponding row in (2.3), which will be called a signature.

- For any \(i\) with \(v_i(x) \neq 0\), \(\text{lm}(u_i(x))\) is not divisible by \(\text{lm}(G_0)\). This follows from (2.11).

- In the process of computing the Gröbner bases, whenever we get some \(u \cdot g \equiv 0 \pmod{G}\), we
add $u$ to $G_0$. So we never need to consider any $u'$ such that $\text{lt}(u')$ is divisible by $\text{lt}(u)$.

- Both $G_0$ and $G_1$ have many redundant polynomials. We do not want to store most of them.

We need to decide which rows to store and how to perform row operations while many rows are missing. In the next section, we shall introduce regular top-reductions to emulate the row operations above and super top-reductions to detect rows that need not be stored.

### 2.3 Algorithm

Our algorithm computes a Gröbner basis for $(\mathbf{I}: g)$ in the process of computing a Gröbner basis for $(\mathbf{I}, g)$. The Gröbner basis for $(\mathbf{I}: g)$ is stored in the list $H$ in the algorithm described in figure 2.1. If one does not need a Gröbner basis for $(\mathbf{I}: g)$, one is free to retain only the leading monomials of $H$. This improves efficiency when only the Gröbner basis for $(\mathbf{I}, g)$ is required. We provide Singular code for this version at http://www.math.clemson.edu/~sgao/code/g2v.sing.

Let $R = \mathbb{F}[x_1, \ldots, x_n]$ with any fixed monomial order $\prec$ as above. Let $G = \{f_1, f_2, \ldots, f_m\}$ be any given Gröbner basis for $\mathbf{I}$ and let $g \in R$. Consider all pairs $(u, v) \in R^2$ satisfying

\[ ug \equiv v \pmod{G}. \]

Certainly, $G \subset (\mathbf{I}, g)$ and $G \subset (\mathbf{I}: g)$. That is, we have the trivial solutions $(f_1, 0), (f_2, 0), \ldots, (f_m, 0)$ and

\[ (0, f_1), (0, f_2), \ldots, (0, f_m). \]

The first nontrivial solution for (2.13) is $(1, g)$.

We need to introduce a few concepts before proceeding. For any pair $(u, v) \in R^2$, $\text{lm}(u)$ is called the signature of $(u, v)$. We make the convention that $\text{lm}(0) = 0$. Our definition of signature is similar in purpose to that of Faugère [17]. To simulate the row operation (R1), we introduce the concept of regular top-reduction. Our regular top-reduction is similar to the top-reduction used by Faugère [17], but our use of super top-reduction below seems to be new. We say that $(u_1, v_1)$ is top-reducible by $(u_2, v_2)$ if

(i) $\text{lm}(v_2) \mid \text{lm}(v_1)$, and

(ii) $\text{lm}(tu_2) \preceq \text{lm}(u_1)$ where $t = \frac{\text{lm}(v_2)}{\text{lm}(v_1)}$. 

26
The corresponding **top-reduction** is then

\[(u_1, v_1) - ct(u_2, v_2) \equiv (u_1 - ctu_2, v_1 - ctv_2) \pmod{G},\]

where \(c = \text{lc}(v_1)/\text{lc}(v_2)\). The effect of a top-reduction is that the leading monomial in the \(v\)-part is canceled. A top-reduction is called **super**, if

\[\text{lm}(u_1 - ctu_2) \prec \text{lm}(u_1),\]

that is, the leading monomial in the \(u\)-part is also canceled. A super top-reduction happens when

\[\text{lm}(tu_2) = \text{lm}(u_1) \quad \text{and} \quad \frac{\text{lc}(u_1)}{\text{lc}(u_2)} = \frac{\text{lc}(v_1)}{\text{lc}(v_2)}.\]

A top-reduction is called **regular** if it is not super. The signature is preserved by regular top-reductions, but not by super top-reductions.

In our algorithm, we only perform regular top-reductions. We also keep all the \(u\) monic (or 0 for trivial solutions). Hence, for each regular top-reduction of \((u_1, v_1)\) by \((u_2, v_2)\) where \(u_1\) and \(u_2\) are monic, we perform the following steps:

- \(u := u_1 - ctu_2\), and \(v := v_1 - ctv_2\) where \(t = \frac{\text{lm}(v_2)}{\text{lm}(v_2)}\) and \(c = \frac{\text{lc}(v_1)}{\text{lc}(v_2)}\);
- if \(\text{lm}(u_1) = t\text{lm}(u_2)\) then \(u := u/(1 - c)\) and \(v := v/(1 - c)\);
- \(u := \text{Normal}(u, G)\) and \(v := \text{Normal}(v, G)\), the normal forms of \(u\) and \(v\) modulo \(G\).

Note that, if \(\text{lm}(u_1) = t\text{lm}(u_2)\) and \(c = 1\), then \((u_1, v_1)\) is super top-reducible by \((u_2, v_2)\). We never perform super top-reductions in our algorithm. In the case that \((u_1, v_1)\) is not regular top-reducible by other pairs known but is super top-reducible, we discard the pair \((u_1, v_1)\), which corresponds to a row in the equation \((2.5)\) that needs not be stored (in this case \(v_1\) is redundant in \(G_1\)).

Now we introduce a new concept of so-called \(J\)-pair for any two pairs of polynomials. Initially, we have the trivial solution pairs in \((2.14)\) and the pair

\[(1, v), \quad \text{where} \ v = \text{Normal}(g, G), \text{assuming} \ v \neq 0.\]

We find new solution pairs that are not top-reducible by the known pairs, hence must be stored. For
any monomial \( t \), consider the pair \( t(1, v) \). If \( t(1, v) \) is not top-reducible by any \((0, f)\) where \( f \in G \), then \( t(1, v) \mod G \) is super top-reducible by \((1, v)\), hence we don’t need to store this pair. However, if \( t(1, v) \) top-reducible by some \((0, f)\) where \( f \in G \), then the new pair after reduction by \((0, f)\) may not be top-reducible by \((1, v)\) any more, hence it must be stored. This means we find the smallest monomial \( t \) so that the pair \( t(1, v) \) is top-reducible by some \((0, f)\). This can happen only if \( t \text{lm}(v) \) is divisible by \( \text{lm}(f) \) for some \( f \in G \). Hence \( t \) should be such that \( t \text{lm}(v) = \text{lcm}(\text{lm}(v), \text{lm}(f)) \). We consider all these \( t \) given by \( f \in G \).

More generally, suppose we have computed a list of solution pairs

\[
(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k),
\]

including the pairs in (2.14). We consider all pairs \( t(u_i, v_i) \), \( 1 \leq i \leq k \), that may be top-reducible by some pair in (2.15). The \( t \) must come from \( \text{lcm}(\text{lm}(v_i), \text{lm}(v_j)) \) for some \( j \neq i \). This leads us to the concept of a joint pair from any two pairs as defined below.

Let \( (u_1, v_1) \) and \( (u_2, v_2) \) be two pairs of polynomials with \( v_1 \) and \( v_2 \) both nonzero. Let

\[
\text{lcm}(\text{lm}(v_1), \text{lm}(v_2)) = t, \quad t_1 = \frac{t}{\text{lm}(v_1)}, \quad t_2 = \frac{t}{\text{lm}(v_2)}.
\]

Find \( \max(t_1 \text{lm}(u_1), t_2 \text{lm}(u_2)) \), say equal to \( t_1 \text{lm}(u_i) \). Then

- \( t_1 \text{lm}(u_i) \) is called the **J-signature** of the two pairs;
- \( t_1 v_i \) is called the **J-polynomial** of the two pairs;
- \( t_i (u_i, v_i) = (t_i u_i, t_i v_i) \) is called the **J-pair** of the two pairs;

where \( J \) means “joint”. In comparison, the \( S \)-polynomial of \( v_1 \) and \( v_2 \) is \( t_1 v_1 - (c_1/c_2)t_2 v_2 \) where \( c_i = \text{lc}(v_i) \). Hence our J-polynomials are related to \( S \)-polynomials. Notice that the J-signature of \( (u_1, v_1) \) and \( (u_2, v_2) \) is the same as the signature of the J-pair of \( (u_1, v_1) \) and \( (u_2, v_2) \).

The basic idea of our algorithm is as follows. Initially, we have the pair \((1, g) \mod G \) and the trivial pairs in (2.14). From these pairs, we form all J-pairs and store them in a list \( \text{JP} \). Then take the smallest J-pair from \( \text{JP} \) and repeatedly perform regular top-reductions until it is no longer regular top-reducible. If the \( v \) part of the resulting pair is zero, then the \( u \) part is a polynomial in \((I : g)\), and we store this polynomial. If the \( v \) part is nonzero, then we check if the resulting J-pair is super top-reducible. If so, then we discard this J-pair; otherwise, we add this pair to the current
Gröbner basis and form new J-pairs and add them to JP. Repeat this process for each pair in JP.

The algorithm is described more precisely in Figure 2.1 below. In the algorithm, we include two options: in first option we only keep the leading monomials of $u$’s and there is no need to update $u$’s in each regular top-reduction, so we compute a Gröbner basis for $\text{lm}(I : g)$; in the second option, we actually update $u$ in each regular top-reduction as specified above, so we compute a Gröbner basis for $(I : g)$.

**Input:**
- $G = [f_1, f_2, \ldots, f_m]$, a Gröbner basis for an ideal $I$, and
- $g$ a polynomial.

**Output:**
- A Gröbner basis for $\langle I, g \rangle$, and a Gröbner basis for $\text{lm}(I : g)$ or for $(I : g)$.

**Variables:**
- $U$ a list of monomials for $\text{lm}(u)$ or of polynomials for $u$;
- $V$ a list of polynomials for $v$;
- $H$ a list of monomials for $\text{lm}(u)$ or of polynomials for $u$ so that $u \in (I : g)$ found so far;
- $JP$ a list of pairs $(t, i)$, where $t$ is a monomial so that $t(u_i, v_i)$ is the J-pair of $(u_i, v_i)$ and $(u_j, v_j)$ for some $j \neq i$.

We shall refer $(t, i)$ as the J-pair of $(u_i, v_i)$ and $(u_j, v_j)$.

**Step 0.**
- $U = [0, \ldots, 0]$ with length $m$, and $V = [f_1, \ldots, f_m]$ (so that $(u_i, v_i) = (0, f_i)$, $1 \leq i \leq m$);
- $H = [\text{lm}(f_1), \text{lm}(f_2), \ldots, \text{lm}(f_m)]$ or $H = [f_1, f_2, \ldots, f_m]$;
- Compute $v = \text{Normal}(g, G)$;
- If $v = 0$, then append 1 to $H$ and return $V$ and $H$ (stop the algorithm);
- else append 1 to $U$ and $v$ to $V$;
- $JP = [\ ]$, an empty list;
- For each $1 \leq i \leq m$,
  - compute the J-pair of the two pairs $(u_{m+1}, v_{m+1}) = (1, v)$ and $(u_i, v_i) = (0, f_i)$, such a J-pair must be of the form $(t_i, m + 1)$,
  - insert $(t_i, m + 1)$ into $JP$ whenever $t_i$ is not reducible by $H$.
  (store only one J-pair for each distinct J-signature).

**Step 1.**
- Take a minimal (in signature) pair $(t, i)$ from $JP$, and delete it from $JP$.

**Step 2.**
- Reduce the pair $t(u_i, v_i)$ repeatedly by the pairs in $(U, V)$, using regular top-reductions, say to get $(u, v)$, which is not regular top-reducible.

**Step 3a.**
- If $v = 0$, then append $\text{lm}(u)$ or $u$ to $H$ and delete every J-pair $(t, \ell)$ in $JP$ whose signature $t\text{lm}(u_\ell)$ is divisible by $\text{lm}(u)$.

**Step 3b.**
- If $v \neq 0$ and $(u, v)$ is super top-reducible by some pair $(u_j, v_j)$ in $(U, V)$, then discard the pair $(t, i)$.

**Step 3c.**
- Otherwise,
  - append $u$ to $U$ and $v$ to $V$,
  - form new J-pairs of $(u, v)$ and $(u_j, v_j)$, $1 \leq j \leq \#U - 1$, and
  - insert into $JP$ all such J-pairs whose signature are not reducible by $H$.
  (store only one J-pair for each distinct J-signature).

**Step 4.**
- While $JP$ is not empty, go to step 1.

**Return:**
- $V$ and $H$.

Figure 2.1: The G2V algorithm

29
It can be proved that, when $JP$ is empty, $\text{Im}(H)$ is a Gröbner basis for $\text{Im}(I : g)$ and $V$ is a Gröbner basis for $(I, g)$, which may not be minimal. Also, for each solution $(u, v)$ to (2.13), we have either $\text{Im}(u)$ is reducible by $H$, or $(u, v)$ can be top-reduced to $(0, 0)$ by $(U, V)$ (using both regular and super top-reductions). The proof of the algorithm will be included elsewhere for a more general version of this algorithm that needs not be incremental.

It should be remarked that in our algorithm we always pick the J-pair with minimal signature to reduce. This is to emulate the downward row operations of the matrix. The algorithm may not work if one uses another strategy, say picking J-pairs with minimal total degree in the $v$ part.

### 2.4 Comparisons and Conclusions

In order to determine how our algorithm compared to, say F5 and F5C, we computed Gröbner basis for various benchmark examples as provided in [14]. We used the examples and algorithm implementation for F5 and F5C provided by the URL in [14] which was all implemented in the Singular computer algebra system. Our implementation was meant to mirror the F5C implementation in terms of code structure and Singular kernel calls. For example, both implementations use the procedure “reduce” to compute normal form of a polynomial modulo a Gröbner basis. Reasonable differences were unavoidable though. For example, F5C uses Quicksort while $G^2V$ performs one step of a Mergesort in the function “insertPairs”.

All examples considered were over the field of 7583 elements with the graded reverse lexicographic ordering. In addition to the usual wall clock times, several other measures of performance were considered, namely

1. Wall clock time (from a single run),
2. Extraneous generators,
3. Memory usage,
4. Count of J-pairs or S-pairs reduced, and
5. Count of normal forms computed.

The runtimes and ratios of runtimes are presented in Table 2.1. One can see that, for these examples, our algorithm is two to ten times faster than F5 and F5C.
Table 2.1: Runtimes in seconds and ratios of runtimes for various test cases in Singular 3.1.0.6 on an Intel Core 2 Quad 2.66 GHz. The #generators refers to a reduced Gröbner basis

<table>
<thead>
<tr>
<th>Test Case (#generators)</th>
<th>F5</th>
<th>F5C</th>
<th>G²V</th>
<th>F5/G²V</th>
<th>F5C/G²V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>1.48</td>
<td>0.93</td>
<td>0.36</td>
<td>4.11</td>
<td>2.58</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>2.79</td>
<td>2.34</td>
<td>0.37</td>
<td>7.54</td>
<td>6.32</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>30.27</td>
<td>22.76</td>
<td>4.64</td>
<td>6.52</td>
<td>4.91</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>290.97</td>
<td>177.74</td>
<td>29.88</td>
<td>9.74</td>
<td>5.95</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>1180.08</td>
<td>299.65</td>
<td>21.34</td>
<td>55.30</td>
<td>14.04</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>30.93</td>
<td>29.87</td>
<td>2.06</td>
<td>15.01</td>
<td>14.50</td>
</tr>
<tr>
<td>Cyclic6 (99)</td>
<td>28.44</td>
<td>22.06</td>
<td>5.65</td>
<td>5.03</td>
<td>3.90</td>
</tr>
<tr>
<td>Cyclic7 (443)</td>
<td>4591.20</td>
<td>2284.05</td>
<td>732.33</td>
<td>6.27</td>
<td>3.12</td>
</tr>
</tbody>
</table>

F5, F5C and our algorithm G²V are all incremental. That is, given a list of polynomials \( g_1, \ldots, g_m \), a Gröbner basis is computed for \( \langle g_1, g_2, \ldots, g_i \rangle \) for \( i = 1, 2, \ldots, m \). Hence, in each iteration, all three algorithms are given a polynomial \( g \in R \) and a Gröbner basis \( G \) for some ideal \( I \), and they compute a Gröbner basis for \( \langle I, g \rangle \). The computed Gröbner basis is not necessarily reduced, and any redundant polynomials in the basis will result in extra S-polynomials or J-polynomials to be reduced. Fewer generators at any given time means that fewer S-polynomials or J-polynomials need to be considered. F5 uses \( G \) as it was computed, so may not be reduced, however, F5C and our algorithm always replace \( G \) by a reduced Gröbner basis. Table 2.2 lists the number of polynomials in the Gröbner bases that were output by each algorithm on the last iteration of each example. Computation time is not the only limiting factor in a Gröbner basis computation. Storage requirements also limit computation. Table 2.3 lists the maximum amount of memory each algorithm needed in the processing of examples. Again, we cannot make generalizations from the memory results because this is only one possible implementation of each algorithm in one possible CAS. The last two criteria were also measured, but the results were less convincing of a clear winner. The counts of J-pairs/S-polynomials processed are included in table 2.4.

In conclusion, we presented a precise relationship among the degrees of the ideals \( I, \langle I, g \rangle \) and \( \langle I : g \rangle \), and a connection between the Gröbner bases of \( \langle I, g \rangle \) and \( \langle I : g \rangle \). This allowed us to design a new algorithm, which is conceptually simpler and yet more efficient than F5 and F5C.
Table 2.2: The number of generators in the Gröbner basis in the last iteration but before computing a reduced Gröbner basis. Of course, F5 never computes the reduced Gröbner basis.

<table>
<thead>
<tr>
<th>Test Case (#generators)</th>
<th>F5</th>
<th>F5C</th>
<th>G²V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>61</td>
<td>44</td>
<td>63</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>74</td>
<td>65</td>
<td>52</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>185</td>
<td>163</td>
<td>170</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>423</td>
<td>367</td>
<td>335</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>643</td>
<td>399</td>
<td>189</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>237</td>
<td>217</td>
<td>115</td>
</tr>
<tr>
<td>Cyclic6 (99)</td>
<td>202</td>
<td>183</td>
<td>146</td>
</tr>
<tr>
<td>Cyclic7 (443)</td>
<td>1227</td>
<td>1006</td>
<td>658</td>
</tr>
</tbody>
</table>

Table 2.3: The maximum amount of memory (in KiB) Singular 3.1.0.6 used from startup to the conclusion of the Gröbner basis computation. Memory amounts obtained with “memory(2);”

<table>
<thead>
<tr>
<th>Test Case (#generators)</th>
<th>F5</th>
<th>F5C</th>
<th>G²V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>1359</td>
<td>828</td>
<td>1255</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>1955</td>
<td>1409</td>
<td>1254</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>8280</td>
<td>4600</td>
<td>5369</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>40578</td>
<td>20232</td>
<td>20252</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>130318</td>
<td>50566</td>
<td>32517</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>3144</td>
<td>2720</td>
<td>2824</td>
</tr>
<tr>
<td>Cyclic6 (99)</td>
<td>2749</td>
<td>2280</td>
<td>1789</td>
</tr>
<tr>
<td>Cyclic7 (443)</td>
<td>48208</td>
<td>23292</td>
<td>24596</td>
</tr>
</tbody>
</table>

Table 2.4: The count of J-pairs or S-polynomials processed by F5, F5C and G²V
Chapter 3

The GVW Algorithm

The following is joint work with Shuhong Gao and Mingsheng Wang. It was presented at ISSAC 2010 in Munich, Germany, and, as of this writing, it has been submitted to SIAM Journal on Computing.

3.1 Introduction

In our previous work [19], we presented an algorithm (G2V) that is incremental in the same fashion as F5 and F5C but is much simpler and faster (in the range of 2 to 10 times faster on the benchmark problems of [14]). Incremental algorithms, however, are at a disadvantage as the order of the input generators can have a profound effect on the complexity of the intermediate bases. Our contribution in this chapter is to leave the incremental structure behind and develop a new algorithm (GVW) that matches Buchberger’s algorithm in simplicity yet is much faster than modern algorithms. This new algorithm uses signatures in the same fashion as F5 and G2V style algorithms, but allows arbitrary orderings of the signatures. In this way, a Gröbner basis for \langle g_1, \ldots, g_m \rangle can be computed in one-shot, incrementally, or as a hybrid of the two. Moreover, just as reductions to zero in G2V provide information about the colon ideal (and thus saving further computations), the present algorithm uses reductions to zero to discover more information about the \langle g_1, \ldots, g_m \rangle-syzygy module in order to prevent later reductions to zero.

Upon termination, the present algorithm computes a (leading terms of) minimal Gröbner basis for the \langle g_1, \ldots, g_m \rangle-syzygy module. Unless one needs to compute the syzygy module under a
certain term order, one is free to choose the term order of the syzygy module to maximize performance. The POT order corresponds to an incremental style algorithm while non-elimination orders correspond to one-shot algorithms. In fact, in the incremental mode, our new algorithm specializes to the G2V algorithm, but we find that the one-shot mode can be much faster (another 2 to 10 times faster than incremental mode and therefore something like 4 to 20 times faster than F5 and F5C).

The chapter is organized as follows. In Section 3.2, we introduce the basic concepts and theory for our algorithm. In particular, we define signatures, regular top-reductions, super top-reductions, and the concept of eventually super top-reducible. We also introduce J-pairs that are in some sense similar to S-polynomials. Then we characterize Gröbner bases in terms of J-pairs in a similar fashion as Buchberger’s characterization in terms of S-polynomials. Our characterization goes a step further, that is, it also tells us when we have a Gröbner basis for the corresponding syzygy module. In Section 3.3, we present our algorithm and prove its correctness. The problem of finite termination of our algorithm is still open. We present computer experiments of our algorithm that shows how the algorithm performs under different term orders for the syzygy module. Finally, in Section 3.4, we show how our algorithm can be adapted to compute Gröbner bases for modules and for polynomials over quotient rings, which would allow one to design more flexible incremental algorithms.

3.2 Theory

Let \( R = \mathbb{F}[x_1, \ldots, x_n] \) be a polynomial ring over a field \( \mathbb{F} \) with \( n \) variables. Given polynomials \( g_1, \ldots, g_m \in R \), we wish to compute a Gröbner basis for the ideal

\[
I = \langle g_1, \ldots, g_m \rangle = \{ u_1 g_1 + \cdots + u_m g_m : u_1, \ldots, u_m \in R \} \subseteq R
\]  

(3.1)

with respect to some term order on \( R \). Define

\[
H = \{(u_1, \ldots, u_m) \in R^m : u_1 g_1 + \cdots + u_m g_m = 0 \},
\]  

(3.2)

called the syzygy module of \( g = (g_1, \ldots, g_m) \). We would like to develop an algorithm that computes Gröbner bases for both \( I \) and \( H \). Note that elements of \( R^m \) are viewed as row vectors.
and are denoted by bold letters say $g, u$ etc. We consider the following $R$-submodule of $R^m \times R$:

$$M = \{(u, v) \in R^m \times R : ug^t = v \}.$$  \hfill (3.3)

We define $E_i \in R^m$ to be the $i^{th}$ standard unit vector. Note that a monomial (or a term) in $R$ is of the form $x^\alpha = \prod_{i=1}^n x_i^{a_i}$ where $\alpha = (a_1, \ldots, a_n) \in \mathbb{N}^n$ is any vector of non-negative integers, and a term in $R^m$ is of the form $x^\alpha E_i$

where $1 \leq i \leq m$ and $\alpha \in \mathbb{N}^n$. The $R$-module $M$ is generated by

$$(E_1, g_1), (E_2, g_2), \ldots, (E_m, g_m).$$ \hfill (3.4)

Fix any term order $\prec_1$ on $R$ and any term order $\prec_2$ on $R^m$. We emphasize that the order $\prec_2$ may or may not be related to $\prec_1$ in the theory below, though $\prec_2$ is usually an extension of $\prec_1$ to $R^m$ in implementation. For the sake of convenience, we shall use the following convention for leading terms:

$$\text{lm}(u) = \text{lm}_{\prec_1}(u), \quad \text{lm}(v) = \text{lm}_{\prec_2}(v)$$

for any $v \in R$ and $u \in R^m$. Note that, for $v \in R$, $\text{lm}(v)$ is a monomial $x^\alpha$, while, for $u \in R^m$, $\text{lm}(u)$ is a term $x^\alpha E_i$ for some $\alpha \in \mathbb{N}^n$ and $1 \leq i \leq m$. We make the convention that if $v = 0$ then $\text{lm}(v) = 0$; similarly for $\text{lm}(u)$. This should not cause any confusion, but the reader should keep the two different orders in mind.

For any $(u, v) \in R^m \times R$, we call $\text{lm}(u)$ the signature of $(u, v)$. This is similar to the signature used in F5. Suppose $(u_1, v_1), (u_2, v_2) \in R^m \times R$ are two pairs with $v_1$ and $v_2$ both nonzero. Let

$$t = \text{lcm}(\text{lm}(v_1), \text{lm}(v_2)), \quad t_1 = \frac{t}{\text{lm}(v_1)}, \quad t_2 = \frac{t}{\text{lm}(v_2)}.$$  \hfill (3.5)

Suppose $\max(t_1 \text{lm}(u_1), t_2 \text{lm}(u_2)) = t_i \text{lm}(u_i)$ where $i = 1$ or 2. Then

- $t_i(u_i, v_i) = (t_i u_i, t_i v_i)$ is called a J-pair of $(u_1, v_1)$ and $(u_2, v_2)$;
• $t_i \text{lm}(u_i)$ is called the J-signature of $(u_1, v_1)$ and $(u_2, v_2)$;

where J means “joint” of the two pairs. When $t_1 \text{lm}(u_1) = t_2 \text{lm}(u_2)$, we can pick either $t_1(u_1, v_1)$ or $t_2(u_2, v_2)$ as a J-pair. However, the J-signature of $(u_1, v_1)$ and $(u_2, v_2)$ is unique.

We should mention that the S-polynomial of $v_1$ and $v_2$ is $t_1 v_1 - c t_2 v_2$ where $c = \text{lc}(v_1) / \text{lc}(v_2)$. Hence the monomials $t_1$ and $t_2$ used in our J-pair is the same as those used in the S-polynomial. In the case of S-polynomials, the goal is to cancel the leading terms of $v$’s. In our J-pairs, the leading terms of $v$’s are not cancelled, but will be cancelled in later top-reductions (for most cases, it is cancelled by the other pair $t_3 - i(u_3 - i, v_3 - i)$). In our algorithm below, we may produce many J-pairs that have the same J-signature, but we only keep one per distinct signature. Note that we never calculate the J-pair of $(u_1, v_1)$ and $(u_2, v_2)$ when either $v_1$ or $v_2$ is zero.

Next we define top-reductions in $R^m \times R$. Let $(u_1, v_1), (u_2, v_2) \in R^m \times R$ be any two pairs. When $v_2$ is nonzero, we say $(u_1, v_1)$ is top-reducible by $(u_2, v_2)$ if the following two conditions are satisfied:

(i) $v_1$ is nonzero and $\text{lm}(v_2)$ divides $\text{lm}(v_1)$; and

(ii) $\text{lm}(tu_2) \preceq \text{lm}(u_1)$ where $t = \text{lm}(v_1) / \text{lm}(v_2)$.

The corresponding top-reduction is then

$$(u_1, v_1) - c t(u_2, v_2) = (u_1 - c tu_2, v_1 - ct v_2), \quad (3.5)$$

where $c = \text{lc}(v_1) / \text{lc}(v_2)$. The effect of a top-reduction is that the leading monomial in the $v$-part is canceled without increasing the signature of $(u_1, v_1)$. Such a top-reduction is called regular, if

$$\text{lm}(u_1 - c tu_2) = \text{lm}(u_1),$$

and super otherwise. So the signature of $(u_1, v_1)$ remains the same under a regular top-reduction but becomes smaller under a super top-reduction. A super top-reduction happens if

$$\text{lm}(tu_2) = \text{lm}(u_1) \text{ and } \frac{\text{lc}(u_1)}{\text{lc}(u_2)} = \frac{\text{lc}(v_1)}{\text{lc}(v_2)}.$$ 

When $v_2 = 0$, we say that $(u_1, v_1)$ is top-reducible by $(u_2, 0)$ if $u_1$ and $u_2$ are both nonzero and $\text{lm}(u_2)$ divides $\text{lm}(u_1)$. In this case, we could use $(u_2, 0)$ to top-reduce $(u_1, v_1)$ by setting
\[ t = \text{lcm}(u_1)/\text{lc}(u_2) \] and \( c = \text{lc}(u_1)/\text{lc}(u_2) \) in equation (3.5). Such a top-reduction will decrease a the signature of \((u_1, v_1)\) without increasing the leading term of \(v_1\) (even if \(v_1 = 0\)) and is therefore always called super. We note that a pair \((u_1, 0)\) is never top-reducible by \((u_2, v_2)\) for \(v_2 \neq 0\).

In our algorithm below, we only detect super top-reductions of the two kinds defined here, but never actually perform super top-reductions. We should mention that the top-reductions used in F5 correspond to regular top-reductions in our sense, but some of our regular top-reductions are not allowed in \(F_5\) (e.g. when \(\text{lcm}(u_1) = t\text{lc}(u_2)\)).

**Lemma 3.1.** Let \(t\) be a monomial in \(R\). If a pair \((u_1, v_1)\) is (regular) top-reducible by \((u_2, v_2)\), where both \(v_1\) and \(v_2\) are nonzero, then \(t_1(u_1, v_1)\) is a J-pair of \((u_1, v_1)\) and \((u_2, v_2)\), where

\[
t_1 = \frac{\text{lcm}(\text{lcm}(v_1), \text{lcm}(v_2))}{\text{lc}(v_1)} = \frac{\text{lcm}(v_2)}{\gcd(\text{lc}(v_1), \text{lc}(v_2))}
\]

and \(t_1\) is a divisor of \(t\). Furthermore, \(t_1(u_1, v_1)\) is (regular) top-reducible by \((u_2, v_2)\).

**Proof.** Since \(t(u_1, v_1)\) is top-reducible by \((u_2, v_2)\) and both \(v_1\) and \(v_2\) are nonzero, there is a monomial \(s\) such that

\[
t \text{lc}(v_1) = s \text{lc}(v_2), \quad t \text{lc}(u_1) \succeq s \text{lc}(u_2).
\]

(3.6)

Let

\[
t_2 = \frac{\text{lcm}(\text{lc}(v_1), \text{lc}(v_2))}{\text{lc}(v_2)} = \frac{\text{lc}(v_1)}{\gcd(\text{lc}(v_1), \text{lc}(v_2))}.
\]

Then (3.6) implies that, for some monomial \(w\),

\[
t = \frac{\text{lc}(v_2)}{\gcd(\text{lc}(v_1), \text{lc}(v_2))} w = t_1 w, \text{ and }
\]

\[
s = \frac{\text{lc}(v_1)}{\gcd(\text{lc}(v_1), \text{lc}(v_2))} w = t_2 w.
\]

Hence (3.6) implies that \(t_2 \text{lc}(u_2) \leq t_1 \text{lc}(u_1)\). So we know \(\max(t_2 \text{lc}(u_2), t_1 \text{lc}(u_1)) = t_1 \text{lc}(u_1)\), thus \(t_1(u_1, v_1)\) is a J-pair of \((u_1, v_1)\) and \((u_2, v_2)\). Note that by (3.6), we have that \(t_1(u_1, v_1)\) is regular top-reducible by \((u_2, v_2)\) whenever \(t(u_1, v_1)\) is regular top-reducible by \((u_2, v_2)\). \(\square\)

Now let

\[(u_1, v_1), \ldots, (u_k, v_k)\] (3.7)

be a list of pairs in \(M\) as defined in (3.3). The list (3.7) is called a **strong Gröbner basis for** \(M\).
if every pair \((u, v) \in M\) is top-reducible by some pair in \((3.7)\).

**Proposition 3.2.** Suppose that the list of pairs in \((3.7)\) is a strong Gröbner basis for \(M\). Then

1. \(G_0 = \{u_i : v_i = 0, 1 \leq i \leq k\}\) is a Gröbner basis for the syzygy module of \(g = (g_1, \ldots, g_m)\), and
2. \(G_1 = \{v_i : 1 \leq i \leq k\}\) is a Gröbner basis for \(I = \langle g_1, \ldots, g_m \rangle\).

**Proof.** For any \(u = (u_1, \ldots, u_m)\) in the syzygy module of \(g\), we have \((u, 0) \in M\). By our assumption, \((u, 0)\) is top-reducible by some pair \((u_i, v_i)\) in \((3.7)\). Then we must have \(v_i = 0\), thus \(u_i \in G_0\) and \(\text{lm}(u)\) is reducible by \(\text{lm}(u_i)\). This proves that \(G_0\) is a Gröbner basis for the syzygy module of \(g\).

Now suppose \(v \in I\) and is nonzero. Then there exists \(u = (u_1, \ldots, u_m) \in R^m\) so that \(ug^t = v\), hence \((u, v) \in M\). Among all such \(u\), we pick one so that \(\text{lm}(u)\) is minimum. Since \((u, v) \in M\), it is top-reducible by some \((u_i, v_i)\) where \(1 \leq i \leq k\). If \(v_i = 0\), then we could use \((u_i, 0)\) to reduce \((u, v)\) to get a \(u'\) so that \(u'g^t = v\) and \(\text{lm}(u')\) is smaller than \(\text{lm}(u)\), contradicting to the minimality of \(\text{lm}(u)\). So \(v_i \neq 0\) and \(\text{lm}(v_i)\) divides \(\text{lm}(v)\). Hence \(G_1\) is a Gröbner basis for \(I\).

**Remark.** Note that \(M \subset R^m \times R\) has a Gröbner basis in the usual sense as a submodule of \(R^{m+1}\) where the leading term of \((u, v)\) is \(\text{lm}(v)E_{m+1}\) if \(v \neq 0\) and \(\text{lm}(u)\) if \(v = 0\). The above proposition implies that a strong Gröbner basis for \(M\) is a Gröbner basis for \(M\) as a submodule of \(R^{m+1}\), but the converse may not be true for an arbitrary submodule \(M\) of \(R^{m+1}\). This is why we call our basis a strong Gröbner basis.

Let \(S\) be any set of pairs in \(R^m \times R\). We say that a pair \((u, v) \in R^m \times R\) is regular top-reducible by \(S\) if it is regular top-reducible by at least one pair in \(S\). We call \((u, v)\) eventually super top-reducible by \(S\) if there is a sequence of regular top-reductions of \((u, v)\) by pairs in \(S\) that reduce \((u, v)\) to a pair \((u', v')\) that is no longer regular top-reducible by \(S\) but is super top-reducible by at least one pair in \(S\).

**Theorem 3.3.** Suppose the list \((3.7)\) satisfies the following: for any term \(T \in R^m\), there is a pair \((u_i, v_i)\), \(1 \leq i \leq k\), and a monomial \(t\) such that \(T = t \text{lm}(u_i)\). Then \((3.7)\) is a strong Gröbner basis for \(M\) if and only if every \(J\)-pair of the pairs from \((3.7)\) is eventually super top-reducible by \((3.7)\).

**Proof.** The forward implication is immediate from the definition of a strong Gröbner basis. To show the reverse, we assume that every \(J\)-pair of the pairs in \((3.7)\) is eventually super top-reducible by
(3.7). Assume that there is a pair \((u, v) \in M\) that is not top-reducible by any pair in (3.7). We want to get a contradiction. Among all such pairs \((u, v)\) we pick one with minimal signature \(T = \text{lm}(u)\). Note that \(T \neq 0\). Next, we select a pair \((u_i, v_i)\) from (3.7) such that

(a) \(T = t \text{lm}(u_i)\) for some monomial \(t\), and

(b) \(t \text{lm}(v_i)\) is minimal among all \(1 \leq i \leq k\) satisfying (a).

We claim that \(t(u_i, v_i)\) is not regular top-reducible by (3.7). To prove this claim, we suppose that \(t(u_i, v_i)\) is regular top-reducible by some \((u_j, v_j)\), so both \(v_i\) and \(v_j\) are nonzero. We want to derive a contradiction to the condition (b). By Lemma 3.1, the J-pair of \((u_i, v_i)\) and \((u_j, v_j)\) is \(t_1(u_i, v_i)\) and that \(t_1(u_i, v_i)\) is still regular top-reducible by \((u_j, v_j)\), where

\[
t_1 = \frac{\text{lcm}(\text{lm}(v_i), \text{lm}(v_j))}{\text{lm}(v_i)}, \quad \text{and} \quad t = t_1 w
\]

for some monomial \(w\). As \(t_1(u_i, v_i)\) is a J-pair of two pairs from (3.7), \(t_1(u_i, v_i)\) is eventually super top-reducible by (3.7), say

\[
t_1(u_i, v_i) = \sum_{r=1}^{d} m_r(u_{i_r}, v_{i_r}) + (u', v'), \quad (3.8)
\]

where the first part of the sum represents a sequence of regular top-reductions of \(t_1(u_i, v_i)\) by (3.7), and \((u', v')\) is not regular top-reducible by any pair in (3.7) but is super top-reducible by some pair in (3.7). Note that \(d \geq 1\) as \(t_1(u_i, v_i)\) is regular top-reducible by \((u_j, v_j)\). Also, each regular top-reduction strictly reduces the leading monomial of \(v_i\), but the leading monomial of \(u_i\) remains unchanged. Thus, we have \(\text{lm}(u') = \text{lm}(t_1 u_i)\), but \(\text{lm}(v') < t_1 \text{lm}(v_i)\). Let \(1 \leq \ell \leq k\) be such that \((u', v')\) is super top-reducible by \((u_{\ell}, v_{\ell})\). If \(v_{\ell} = 0\), then \(\text{lm}(u_{\ell})\) divides \(\text{lm}(u') = \text{lm}(t_1 u_i)\), thus divides \(\text{lm}(u)\). Hence \((u, v)\) is top-reducible by \((u_{\ell}, 0)\), contradicting our assumption that \((u, v)\) is not top-reducible by (3.7). We may thus assume that \(v_{\ell} \neq 0\), hence \(v' \neq 0\). Then

\[
(\text{lm}(u'), \text{lm}(v')) = t_3(\text{lm}(u_{\ell}), \text{lm}(v_{\ell})),
\]

where \(t_3 = \text{lm}(v')/\text{lm}(v_{\ell})\). Let \(\overline{t} = t_3 w\). Then

\[
\overline{t} \text{lm}(u_{\ell}) = w \text{lm}(u') = t \text{lm}(u_i) = T
\]

39
and
\[ \ell \text{lm}(v) = w \text{lm}(v') \prec wt_1 \text{lm}(v) = t \text{lm}(v). \]

Thus \((u_i, v_i)\) satisfies (a) but violates (b). Hence \(t(u_i, v_i)\) is not regular top-reducible by (3.7) as claimed.

Returning to the main proof, we perform the cancellation
\[ (\overline{u}, \overline{v}) = (u, v) - ct(u_i, v_i), \quad (3.9) \]
where \(c = lc(u)/lc(u_i)\) so that \(\text{lm}(\overline{u}) \prec \text{lm}(u) = T\). Note that \(\text{lm}(v) \neq t \text{lm}(v_i)\), since otherwise \((u, v)\) would be top-reducible by \((u_i, v_i)\) which contradicts the fact that \((u, v)\) is not top-reducible by any pair in 3.7. Hence \(\overline{v} \neq 0\). Also, as \((\overline{u}, \overline{v}) \in M \) and \(\text{lm}(\overline{u}) \prec T\), we have that \((\overline{u}, \overline{v})\) is top-reducible by (3.7). If \((\overline{u}, \overline{v})\) is top-reducible by some pair \((u_\ell, v_\ell)\) from (3.7) with \(v_\ell = 0\), then we can reduce \((\overline{u}, \overline{v})\) repeatedly by such pairs to get a new pair \((\tilde{u}, \tilde{v})\) that is not top-reducible by any pair in (3.7) with \(v\)-part being zero. Note that \((\tilde{u}, \tilde{v})\) is still in \(M\) and \(\text{lm}(\tilde{u}) \prec T\). Hence \((\tilde{u}, \tilde{v})\) is top-reducible by some pair \((u_\ell, v_\ell)\) from (3.7) with \(v_\ell \neq 0\). As \(\text{lm}(v) \neq t \text{lm}(v_i)\), we consider two cases:

(i) \(\text{lm}(v) \prec t \text{lm}(v_i)\). Then \(\text{lm}(\overline{v}) = t \text{lm}(v_i)\), hence \(t(u_i, v_i)\) is regular top-reducible by \((u_\ell, v_\ell)\) (as \(\text{lm}(\overline{u}) \prec t \text{lm}(u_i)\)). Since \(t(u_i, v_i)\) is not regular top-reducible by any pair in (3.7), this case is impossible.

(ii) \(\text{lm}(v) \succ t \text{lm}(v_i)\). Then \(\text{lm}(\overline{v}) = \text{lm}(v)\), and \((u, v)\) is regular top-reducible by \((u_\ell, v_\ell)\), contradicting the fact that \((u, v)\) is not top-reducible by any pair in (3.7).

Therefore such a pair \((u, v)\) does not exist in \(M\), thus every pair in \(M\) is top-reducible by (3.7). \(\square\)

**Theorem 3.4.** In Theorem 3.3, the condition “every J-pair of pairs from (3.7) is eventually super top-reducible by (3.7)” can be replaced by “for every distinct J-signature from (3.7) there is at least one J-pair from (3.7) with the same J-signature that is eventually super top-reducible by (3.7).”

**Proof.** The proof is the same as that of Theorem 3.3, except that in the equation (3.8) we can still assume that the pair \((u', v')\) is not regular top-reducible by (3.7), but we need to prove that it is super top-reducible by (3.7), which is used in the subsequent proof. By our assumption, however, there is a J-pair, say \(t_2(u_\ell, v_\ell)\), that has the same signature as that of \(t_1(u_i, v_i)\) and is eventually
super top-reducible by (3.7). Suppose that

\[ t_2(u_\ell, v_\ell) = \sum_{r=1}^{s} n_r(u_\ell, v_\ell) + (u'', v''), \]

where the first part of the sum represents a sequence of regular top-reductions of \( t_2(u_\ell, v_\ell) \) by (3.7), and \((u'', v'')\) is not regular top-reducible by any pair in (3.7), but is super top-reducible by some pair \((u_\epsilon, v_\epsilon)\) in (3.7). Note that

\[ \text{lm}(u'') = t_2 \text{lm}(u_\ell) = t_1 \text{lm}(u_\ell) = \text{lm}(u') = T, \]

and \( \text{lm}(v'') \preceq t_2 \text{lm}(v_\ell) \). We may assume that both \( u' \) and \( u'' \) are monic. We claim that \( \text{lm}(v') = \text{lm}(v'') \) and their coefficients are also equal. This implies the desired property that \((u', v')\) is super top-reducible by \((u_\epsilon, v_\epsilon)\), since \( \text{lm}(v') = \text{lm}(v'') \), \( \text{lm}(u') = \text{lm}(u'') \) and \((u'', v'')\) is super top-reducible by \((u_\epsilon, v_\epsilon)\).

To prove the claim, suppose it is not true, that is, either \( \text{lm}(v') \neq \text{lm}(v'') \) or \( \text{lm}(v') = \text{lm}(v'') \) but their coefficients are not equal. Then \( \text{lm}(v' - v'') = \text{lm}(v') \) or \( \text{lm}(v'') \). Note that the signature of \((u' - u'', v' - v'')\) is strictly smaller than \( \text{lm}(u') = \text{lm}(t_1 u_1) \preceq T \). By the hypothesis on the minimality of \( T \), the pair \((u' - u'', v' - v'')\) is top-reducible by (3.7). It follows that either \((u', v')\) is regular top-reducible by (3.7) if \( \text{lm}(v' - v'') = \text{lm}(v') \) or \((u'', v'')\) is regular top-reducible by (3.7) if \( \text{lm}(v' - v'') = \text{lm}(v'') \). Both are contradicting to our assumption that they are not regular top-reducible. Hence the claim, and thus the theorem is proved. \( \square \)

**Remark.** Suppose a final strong Gröbner basis for \( M \) is

\[(u_1, v_1), \ldots, (u_k, v_k). \quad (3.10)\]

At any intermediate step of computation, we only know

\[(u_1, v_1), \ldots, (u_p, v_p) \quad (3.11)\]

for some \( p < k \). In general, a pair \((u, v)\) may be eventually super top-reducible by (3.11) but not by (3.10). How can one decide whether \((u, v)\) is eventually super top-reducible by (3.10) when only (3.11) is known? Our strategy is to always pick the J-pair with minimal signature to reduce. Then
a pair that is eventually super top-reducible by an intermediate basis is always eventually super top-reducible by the final basis. A more detailed argument will be given in the next section.

### 3.3 Algorithm, Term Orderings and Time Comparison

#### 3.3.1 Algorithm and Its Correctness

Our algorithm is based on Theorems 3.3 and 3.4. The basic idea of our algorithm is as follows. Initially, we have the pairs in (3.4) in our Gröbner basis. So the first condition of the theorem is satisfied. From these pairs, we form all J-pairs, keeping only one J-pair for each J-signature. We then take the smallest J-pair among them and repeatedly perform regular top-reductions until it is no longer regular top-reducible, say to get \((u, v)\). If the \(v\) part of the resulting pair is zero, then the \(u\) part is a syzygy in \(H\), and we store this vector. If the \(v\) part is nonzero, then we check if \((u, v)\) is super top-reducible. If so, then we discard this J-pair; otherwise, we add this \((u, v)\) pair to the current Gröbner basis, and form new J-pairs. Repeat this process until all J-pairs are eventually super top-reducible.

We make two improvements on this basic algorithm. First, storing and updating syzygies \(u \in H\) are expensive. In our computation, we shall make all pairs \((u, v)\) monic, namely, the leading coefficient of \(u\) is 1. Now suppose \((u_1, v_1)\) and \((u_2, v_2)\) are any two monic pairs. Then a top-reduction (regular or super) is determined only by \(\text{lm}(u_1), \text{lm}(u_2), v_1, v_2\). The other terms of \(u_1\) and \(u_2\) are not used at all. Let \(T_1 = \text{lm}(u_1)\) and \(T_2 = \text{lm}(u_2)\), the signatures of \((u_1, v_1)\) and \((u_2, v_2)\), respectively. Suppose we store only \((T_1, v_1)\) and \((T_2, v_2)\). Then \((T_1, v_1)\) is regular top-reducible by \((T_2, v_2)\) when \(v_2 \neq 0\), \(\text{lm}(v_1)\) is divisible by \(\text{lm}(v_2)\), \(tT_2 \preceq T_1\), and \(\text{lc}(v_1) \neq \text{lc}(v_2)\) if \(tT_2 = T_1\). The corresponding top-reduction is

\[
v := v_1 - ctv_2
\]

where \(t = \text{lm}(v_1)/\text{lm}(v_2)\) and \(c = \text{lc}(v_1)/\text{lc}(v_2)\), and furthermore, if \(tT_2 = T_1\) then we update \(v\) as

\[
v := v/(1 - c),
\]

to keep the \(u\)-part monic of \((u, v)\) where \(T_1 = \text{lm}(u)\). Then \((T_1, v)\) is the resulting pair of the reduction, and it replaces \((T_1, v_1)\). Our algorithm below will perform regular top-reductions in this
fashion.

Another improvement is to use trivial syzygies. We will store the leading terms of known syzygies in a list called $H$. Let $(T_1, v_1)$ and $(T_2, v_2)$ be any two pairs from the Gröbner basis computed so far, where $v_1$ and $v_2$ are both nonzero. Then, for $1 \leq i \leq 2$, there are $u_i \in \mathbb{R}^m$ such that $\text{lm}(u_i) = T_i$ and $(u_i, v_i) \in M$. Then we have

$$v_2(u_1, v_1) - v_1(u_2, v_2) = (v_2u_1 - v_1u_2, 0) \in M.$$ 

Hence $v_2u_1 - v_1u_2$ is a syzygy of $(g_1, \ldots, g_m)$. Its leading term is

$$T = \max(T_1\text{lm}(v_2), T_2\text{lm}(v_1)),$$

provided that $T_1\text{lm}(v_2) \neq T_2\text{lm}(v_1)$, or $T_1\text{lm}(v_2) = T_2\text{lm}(v_1)$ but $\text{lc}(v_1) \neq \text{lc}(v_2)$. When $T_1\text{lm}(v_2) = T_2\text{lm}(v_1)$ and $\text{lc}(v_1) = \text{lc}(v_2)$, the leading terms in $v_2(u_1, v_1)$ and $v_1(u_2, v_2)$ cancel each other. In that case, we don’t know the leading term of the syzygy, so we just ignore such a syzygy. In all other cases, our algorithm will add $T$ to the list $H$. The benefit of $H$ is in detecting useless reductions. That is, whenever a J-pair has a signature that is divisible by a term in $H$, it is always eventually super top-reducible and hence discarded, thus saving time.

The algorithm is described more precisely in Figure 3.1 below. As mentioned above, we use $H$ to record leading terms of syzygies. In addition to $H$, our algorithm uses two more lists to store the pairs $(T_1, v_1), (T_2, v_2), \ldots, (T_k, v_k)$ with $v_i \neq 0$ for $1 \leq i \leq k$. This list will be stored as

$$U = [T_1, T_2, \ldots, T_k], \quad V = [v_1, v_2, \ldots, v_k].$$

Then $[U, V]$ represents the whole list $(T_1, v_1), (T_2, v_2), \ldots, (T_k, v_k)$.

**Theorem 3.5.** If the algorithm in Figure 3.1 terminates, then $V$ is a Gröbner basis for $I = (g_1, g_2, \ldots, g_m)$ and $H$ is a Gröbner basis for the leading terms of the syzygy module of $(g_1, g_2, \ldots, g_m)$.

**Proof.** To prove the correctness of the algorithm, we need to show the following:

(i) One can delete J-pairs in Steps 0, 3a, and 3b whose signatures are divisible by $\text{lm}(u)$, where $u \in H$.

(ii) A pair that is eventually super top-reducible by an intermediate basis is always eventually
super top-reducible by the final basis.

(iii) One just needs to keep one J-pair for each signature, which follows directly from Theorem 3.4.

Our current basis consists of pairs in $(U, V)$ and $(H, 0)$. For (i), let $(u, v)$ be any pair whose signature $\text{lm}(u)$ is divisible by $\text{lm}(u')$ for some $u' \in H$. Then $(u, v)$ is top-reducible by $(u', 0)$. Any regular top-reduction of $(u, v)$ won’t change $\text{lm}(u)$, so the pair obtained from $(u, v)$ by any sequence of regular top-reductions will be super top-reducible by $(u', 0)$. Hence $(u, v)$ is eventual super top-reducible by the current basis. This means that we don’t need to reduce $(u, v)$, and so we simply discard it.

To see (ii), suppose the final Gröbner basis computed for $M$ is

$$(u_1, v_1), \ldots, (u_k, v_k),$$

while at any intermediate step, we only know

$$(u_1, v_1), \ldots, (u_p, v_p)$$

for some $p < k$. Suppose that the smallest $J$-pair from $\text{JP}$ is $(t, i)$ (i.e., $t(u_i, v_i)$). If $t(u_i, v_i)$ is eventually super top-reducible by (3.13), then $t(u_i, v_i)$ remains eventually super top-reducible by (3.12), as all $(u_j, v_j)$, $j > p$, have strictly larger signature than $t(u_i, v_i)$. If $t(u_i, v_i)$ is not eventually super top-reducible by (3.13), then the basis (3.13) is augmented by a new pair $(u_{p+1}, v_{p+1})$, which is obtained from $t(u_i, v_i)$ via regular top-reductions by (3.13). Hence the $J$-pair $t(u_i, v_i)$ is eventually super top-reducible by the new basis

$$(u_1, v_1), \ldots, (u_p, v_p), (u_{p+1}, v_{p+1}).$$

Note that $(u_{p+1}, v_{p+1})$ has the same signature as the $J$-pair $t(u_i, v_i)$. All new $J$-pairs formed using $(u_{p+1}, v_{p+1})$ will have strictly greater signature than that of $(u_{p+1}, v_{p+1})$ (we never keep any future $J$-pair that has the same signature as $(u_{p+1}, v_{p+1})$). Hence $(u_{p+1}, v_{p+1})$ can not be top-reducible by any pair $(u_j, v_j)$, $j > p + 1$, so the $J$-pair $t(u_i, v_i)$ remains eventually super top-reducible by (3.12). Therefore, any pair that is eventually super top-reducible by our current basis remains so by the final basis.
3.3.2 Finite Termination and Gröbner Bases for the Syzygy Module

As of the submission of this paper, we were unable to show that the algorithm terminates in finitely many steps for general term orders $\prec_1$ and $\prec_2$. Huang [22] has a correct proof of finite termination that requires compatibility between term orders $\prec_1$ and $\prec_2$. Specifically, the compatibility required is that $x^\alpha \prec_1 x^\beta$ if and only if $x^\alpha E_i \prec_2 x^\beta E_i$ for all $1 \leq i \leq m$. Our similar proof of termination is presented in chapter 4.

The algorithm as presented in figure 3.1 only calculates that leading terms of the syzygy module. While one has the option of modifying the algorithm to compute syzygies instead of leading terms of syzygies, there is a more efficient way. Suppose that the algorithm terminates with lists $U, V$, and $H$, then we can compute a minimal Gröbner basis for the syzygy module as follows. Simply make note of the signatures in $U$ and $H$. These signatures are the only $J$-signatures that need to be processed as all others are discarded (in step 3b, they are eventually super top-reducible by $(U, V)$). Now, rerun the algorithm in figure 3.1 calculating the entire $u$-part of each pair for each of the above noted signatures. Upon termination, $H$ will contain a minimal Gröbner basis for the $(g_1, \ldots, g_m)$-syzygy module with respect to ordering $\prec_2$.

3.3.3 Term Orders

Now we discuss choices of term orders. We use $\prec_1$ to represent a term ordering on $R$ and $\prec_2$ to represent a term ordering on $R^m$. While computing Gröbner bases for both $(g_1, \ldots, g_m)$ and $H$, one should set $\prec_1$ and $\prec_2$ to the appropriate term orderings for the Gröbner bases desired. Often, however, the Gröbner basis for $H$ is not needed. Then we only need the leading terms of $H$ to speed up the computation of $(g_1, \ldots, g_m)$. In this case, we have tremendous freedom in the choice of $\prec_2$.

There are many ways that we can construct a term ordering on $R^m$. We consider four extreme cases below. Let $\prec$ be some term order on $R$. We extend $\prec$ to $R^m$ as follows.

(POT) The first is called position over term ordering (POT). We say that $x^\alpha E_i \prec x^\beta E_j$ if $i < j$ or $i = j$ and $x^\alpha \prec x^\beta$.

(TOP) The second is the term over position ordering (TOP). We say that $x^\alpha E_i \prec x^\beta E_j$ if $x^\alpha \prec x^\beta$ or $x^\alpha = x^\beta$ and $i < j$.

(g1) Next is the $g$-weighted degree followed by TOP. We say that $x^\alpha E_i \prec x^\beta E_j$ if $\deg(x^\alpha g_i) <
deg(x^β g_j) or deg(x^α g_i) = deg(x^β g_j) and x^α E_i \prec_{top} x^β E_j, where deg is for total degree.

Finally, we have g-weighted \prec followed by POT. We say that x^α E_i \prec x^β E_j if \text{lm}(x^α g_i) \prec \text{lm}(x^β g_j) or \text{lm}(x^α g_i) = \text{lm}(x^β g_j) and x^α E_i \prec_{pot} x^β E_j.

We remark that, under the POT order, our new algorithm closely corresponds to the G2V algorithm presented in [19]. The reason is that this new algorithm always first picks J-pairs with signatures containing E_1, then those with E_2, etc. This means that it computes Gröbner bases for \langle g_1 \rangle, \langle g_1, g_2 \rangle, \ldots, \langle g_1, g_2, \ldots, g_m \rangle, just like G2V and F5. The only difference is that the intermediate bases may not be reduced and non-leading terms are not reduced as in the computing of normal forms. Because of this fact and other implementation choices, the running times under POT reported here are much slower than those in [19].

Another remark is that our algorithm under the g_1 order roughly corresponds to the behavior of the F4 and XL algorithms [8]. In the XL algorithm, one performs row reductions on a matrix whose rows correspond to all polynomials x^α g_i, 1 \leq i \leq m, with total degree of x^α g_i smaller than some bound. Our algorithm basically works with only some of those rows that correspond to J-signatures. So our algorithm needs much less storage.

3.3.4 Relationship to Other Algorithms

When equipped with the POT order, the algorithm described here processes signatures in the exact same order as G2V. In fact, the difference between this new algorithm with the POT order and G2V is very analogous to the difference between F5 and F5C [14]. Both G2V and F5C calculate a reduced Gröbner basis before the next increment. We will soon see that sacrificing a reduced Gröbner basis in favor of other orderings (g_2 in particular) saves many computations.

While moving away from the incremental structure of G2V is a very natural thing to do, a new proof becomes necessary. The proof in [19] worked for G2V with zero dimensional ideals, but it does not work for non-POT orderings \prec_2. Thus, the proof provided herein is necessary for figure 3.1 and it also proves correctness for G2V with positive dimensional ideals. Also, another benefit of this more general structure is that our new algorithm looks much more like Buchberger’s algorithm than G2V.

Other than allowing the signatures in \( R^m \) to be processed according to a general order \prec_2, the differences between our new algorithm and F5 are identical to the differences between G2V
<table>
<thead>
<tr>
<th>Test Case (#generators)</th>
<th>F5</th>
<th>F5C</th>
<th>G2V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>1.48</td>
<td>0.93</td>
<td>0.36</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>2.79</td>
<td>2.34</td>
<td>0.37</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>30.27</td>
<td>22.76</td>
<td>4.64</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>290.97</td>
<td>177.74</td>
<td>29.88</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>1180.08</td>
<td>299.65</td>
<td>21.34</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>30.93</td>
<td>29.87</td>
<td>2.06</td>
</tr>
<tr>
<td>Cyclic6 (99)</td>
<td>28.44</td>
<td>22.06</td>
<td>5.65</td>
</tr>
<tr>
<td>Cyclic7 (443)</td>
<td>4591.20</td>
<td>2284.05</td>
<td>732.33</td>
</tr>
</tbody>
</table>

Table 3.1: Runtimes in seconds comparing F5, F5C and G2V (GVW under POT ordering) for various test cases in Singular 3110 on an Intel Core 2 Quad 2.66 GHz. This table is reproduced from [19].

and F5/C. Indeed, our algorithms do not require a degree compatible ordering or homogeneous polynomials. For other differences, the interested reader is referred to [13].

3.3.5 Comparison

For ease of exposition, we refer to our algorithm as GVW. We implemented GVW in Singular CAS and C++ so that $(g_1, \ldots, g_m)$ is computed in one-shot, that is, non-incrementally. The Singular implementation is very similar to the that provided in [19] except that it no longer uses Singular’s “reduce” function. Without this use of Singular’s kernel, GVW and G2V are not very comparable in terms of runtimes. For the exact same reason, we did not compare GVW to F5 or F5C as we did in [19] (see Table 3.1, reproduced here\(^1\) for comparison purposes). But as mentioned earlier, GVW under POT is the G2V algorithm.

Just as in [19], various benchmark examples (from [14]) were run for comparison. We collected data from each example under each term ordering for comparison. Tables 3.2 and 3.3 list the runtimes in seconds of GVW for each of the four term orderings. One might notice that the Singular runtimes are surprisingly large (especially compared to G2V in [19]), but that is most likely the result of relying on Singular’s kernel routines less. In examining the timings, we find that $g_2$ seems to be a clear winner among the four term orders.

A more computer independent measure would be a count of J-pairs processed and the number of extraneous generators produced. Table 3.4 lists the total number of J-pairs processed for each term ordering. It is analogous to counting the number of S-pairs processed in F5 or Buchberger’s algorithm. As with the timings, $g_2$ seems to be the most efficient. We remark that in [19], it was

\(^1\)with permission from ACM.
observed that G2V and F5 performed very similarly in terms of J-pairs/S-pairs processed. Therefore, G(VW) under the \( g_2 \) order tends to process fewer J-pairs/S-pairs. Finally, Table 3.5 lists the size of the Gröbner bases produced by G(VW) with each term ordering. These are the Gröbner bases produced by the algorithm before any interreduction occurs to produce a reduced Gröbner basis. We believe this measure to be significant since fewer extraneous generators means quicker reductions. Again, we see that \( g_2 \) produces less redundancy than the other orderings.

One might make the observation that in [19] (or Table 3.1), G2V outperformed F5 and F5C by runtimes of 2 to 10 times\(^2\), while with the present algorithm, G(VW) under the \( g_2 \) ordering outperforms G2V (G(VW) under the POT ordering) by another factor of 2 to 10 times. This comparison shows that if G(VW) under \( g_2 \) were implemented comparably to F5 or F5C, it would compute Gröbner bases around 4 to 20 times faster.

---

\(^2\)We mention that while F5 and F5C require homogeneous input polynomials, G2V and G(VW) do not. In all the tables presented throughout, including table 3.1, G2V and G(VW) were also given the same homogeneous input polynomials as F5 and F5C. In retrospect, it may have been more fair to G2V and G(VW) to remove the homogenizing variable from the input polynomials.

---

### Table 3.2: Runtimes in seconds using Singular 3110 for different term orders

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>( g_1 )</th>
<th>( g_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>1.12</td>
<td>1.01</td>
<td>1.17</td>
<td>0.70</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>1.60</td>
<td>3.25</td>
<td>3.76</td>
<td>1.92</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>24.03</td>
<td>18.00</td>
<td>19.94</td>
<td>9.22</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>167.40</td>
<td>107.97</td>
<td>115.89</td>
<td>52.45</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>80.08</td>
<td>62.19</td>
<td>66.34</td>
<td>66.26</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>10.57</td>
<td>41.90</td>
<td>38.43</td>
<td>11.13</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>27.09</td>
<td>1043.36</td>
<td>1129.20</td>
<td>20.63</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>4194.24</td>
<td>-</td>
<td>-</td>
<td>1835.63</td>
</tr>
</tbody>
</table>

### Table 3.3: Runtimes in seconds using our C++ implementation for different term orders

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>( g_1 )</th>
<th>( g_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>0.34</td>
<td>0.37</td>
<td>0.36</td>
<td>0.37</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>3.26</td>
<td>2.92</td>
<td>2.97</td>
<td>3.16</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>1.78</td>
<td>3.65</td>
<td>3.64</td>
<td>3.81</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>0.08</td>
<td>0.44</td>
<td>0.36</td>
<td>0.09</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>0.34</td>
<td>3.30</td>
<td>3.24</td>
<td>0.15</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>139.56</td>
<td>21417.40</td>
<td>20800.60</td>
<td>35.75</td>
</tr>
<tr>
<td>Cyclic 8 (1182)</td>
<td>107684.35</td>
<td>-</td>
<td>-</td>
<td>5737.41</td>
</tr>
</tbody>
</table>
### Table 3.4: A comparison of the count of the J-pairs processed for the different term orders

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>64</td>
<td>61</td>
<td>61</td>
<td>36</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>72</td>
<td>90</td>
<td>90</td>
<td>50</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>216</td>
<td>181</td>
<td>181</td>
<td>93</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>439</td>
<td>359</td>
<td>359</td>
<td>182</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>475</td>
<td>204</td>
<td>204</td>
<td>214</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>313</td>
<td>378</td>
<td>346</td>
<td>276</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>441</td>
<td>4388</td>
<td>4388</td>
<td>368</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>3562</td>
<td>69502</td>
<td>69502</td>
<td>2375</td>
</tr>
<tr>
<td>Cyclic 8 (1182)</td>
<td>37757</td>
<td>-</td>
<td>-</td>
<td>12245</td>
</tr>
</tbody>
</table>

### Table 3.5: Sizes of Gröbner bases before any interreduction for different term orders

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>1.26</td>
<td>1.26</td>
<td>1.26</td>
<td>0.75</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>1.26</td>
<td>1.76</td>
<td>1.76</td>
<td>1.26</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>6.28</td>
<td>5.77</td>
<td>5.77</td>
<td>2.76</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>25.83</td>
<td>22.81</td>
<td>22.84</td>
<td>8.78</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>39.82</td>
<td>6.28</td>
<td>6.28</td>
<td>6.28</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>2.28</td>
<td>4.28</td>
<td>3.28</td>
<td>1.78</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>1.28</td>
<td>13.56</td>
<td>13.56</td>
<td>1.79</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>22.24</td>
<td>-</td>
<td>-</td>
<td>26.62</td>
</tr>
</tbody>
</table>

### Table 3.6: Maximal amount of memory used (MiB) by Singular for different term orders

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>1.26</td>
<td>1.26</td>
<td>1.26</td>
<td>0.75</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>1.26</td>
<td>1.76</td>
<td>1.76</td>
<td>1.26</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>6.28</td>
<td>5.77</td>
<td>5.77</td>
<td>2.76</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>25.83</td>
<td>22.81</td>
<td>22.84</td>
<td>8.78</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>39.82</td>
<td>6.28</td>
<td>6.28</td>
<td>6.28</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>2.28</td>
<td>4.28</td>
<td>3.28</td>
<td>1.78</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>1.28</td>
<td>13.56</td>
<td>13.56</td>
<td>1.79</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>22.24</td>
<td>-</td>
<td>-</td>
<td>26.62</td>
</tr>
</tbody>
</table>
3.4 Algorithm for Quotient Rings and Modules

3.4.1 Quotient Rings

Let $\mathbb{F}$ be any field and $R = \mathbb{F}[x_1, \ldots, x_n]$ be a polynomial ring. Let $J$ be an ideal of $R$ with Gröbner basis $G = \{f_1, \ldots, f_k\}$. Suppose $I$ is an ideal of $R/J$ generated by $\{g_1, \ldots, g_m\}$ where each $g_i$ is already in normal form with respect to $G$. We wish to compute a Gröbner basis for $I = \langle g_1, \ldots, g_m \rangle$, and the $(g_1, \ldots, g_m)$-syzygy module.

We represent polynomials in $R/J$ in normal form modulo $G$. This means that, for any $g \in R$, we have

$$g \equiv \sum_{i=1}^{\ell} c_i x^{\alpha_i} \pmod{G}$$

(3.15)

where no term $x^{\alpha_i}$ is divisible by any leading term of $G$. This expression can be obtained from $g$ by long division via $G$. When $g \in R$ is viewed as a polynomial in $R/J$, the leading term of $g$ is the maximal $x^{\alpha_i}$ that appears in the normal form (3.15) of $g$. So the leading term of $g$ is never divisible by any leading term of $G$.

We begin by defining a Gröbner basis for an ideal $I \subset R/J$. We say that a generating set $\{g_1, \ldots, g_m\} \subset R/J$ is a Gröbner basis for $I = \langle g_1, \ldots, g_m \rangle$ if for any $h \in I$, the leading monomial of $h$ is divisible by the leading monomial of one of the generators, that is

$$\text{lm}(g_i) \mid \text{lm}(h) \quad \text{for some } 1 \leq i \leq m.$$ 

In other words, if $\{g_1, \ldots, g_m\}$ is a Gröbner basis for $I \subset R/J$ and $\{f_1, \ldots, f_k\}$ is a Gröbner basis for $J \subset R$, then $\{g_1, \ldots, g_m, f_1, \ldots, f_k\}$ is a Gröbner basis for $\langle g_1, \ldots, g_m, f_1, \ldots, f_k \rangle \subset R$.

The syzygy module for $g = (g_1, \ldots, g_m) \in (R/J)^m$ is defined as

$$H = \{(u_1, \ldots, u_m) \in (R/J)^m : u_1 g_1 + \cdots + u_m g_m = 0 \text{ in } R/J\}.$$ 

If viewed in the original ring $R$, every $(g_1, \ldots, g_m)$-syzygy in $(R/J)^m$ can be extended to an $(g_1, \ldots, g_m, f_1, \ldots, f_k)$-syzygy in $R^{m+k}$, which may vary depending on how $u_1 g_1 + \cdots + u_m g_m$ is reduced to 0 by $G$. In our computation, we only need to store the leading term of $(u_1, \ldots, u_m) \in H$ where no terms in the $u_i$’s are divisible by the leading terms of $G$.

Figure 3.2 describes a slight modification to the GVW algorithm that produces a Gröbner ba-
sis for \((g_1, \ldots, g_m) \subset R/J\) and a Gröbner basis for the leading terms of the syzygy module \(H\), which can be used to calculate an actual Gröbner basis for \(H\). Figure 3.2 needs little explanation beyond Figure 3.1. By saying a J-signature \(x^\alpha e_i\) “is not reducible by \(G\) or \(H\), we mean the following. Not being reducible by \(G\) means that for any \(x^\beta \in \text{lm}(G)\), we require that \(x^\beta \nmid x^\alpha\), while not being reducible by \(H\) means for any \(x^\beta e_j \in H\), then either \(i \neq j\) or \(x^\beta \nmid x^\alpha\).

This version of GVW can be used to compute Gröbner bases incrementally, each time adding \(m\) polynomials. For example, to compute a Gröbner basis for an ideal \(I = \langle g_1, \ldots, g_t \rangle \subset R\), one can first compute a Gröbner basis \(G\) for \(J = \langle g_1, \ldots, g_k \rangle \subset R\) where \(k < t\). Then compute a Gröbner basis \(G_1\) for \((g_{k+1}, \ldots, g_t)\) in the quotient ring \(R/J\). Then \(G \cup G_1\) is a Gröbner basis for \(I\). And in the process, \(G\) is used in the reduction of many polynomials (e.g., the \(v\) part of every J-pair).

By interpreting any polynomial in \(R/J\) as having already been reduced to normal form modulo \(G\), we keep the number of terms in each polynomial to a minimum, thus reducing computational and storage requirements. Also, as the choice for \(k\) and \(m\) are arbitrary, one can design an algorithm that can compute Gröbner bases in one-shot, incrementally, or some hybrid of the two. This provides a flexible strategy for computing Gröbner bases for large systems of polynomials.

Calculating reduced Gröbner bases at intermediate steps is another advantage to running the algorithm in this mode. The biggest performance difference between G2V and GVW under POT is that G2V is able to calculate a reduced Gröbner basis between each iteration. In fact, this is the advantage provided by F5C [14]. Our quotient ring version of GVW is able to do the same. For this reason, whenever an elimination order is used on \(R^m\), this quotient ring version should improve performance.

### 3.4.2 Modules

Let \(F\) be a field and \(R = F[x_1, \ldots, x_n]\) be a polynomial ring. Let \(g_1, \ldots, g_m\) be elements in \(R^s\). We define an \(R\)-linear operator \(T : R^m \to R^s\), uniquely determined by \(g_1, \ldots, g_m\), given by

\[
(f_1, \ldots, f_m) \mapsto \left[ \begin{array}{c} g_1 \\ g_2 \\ \vdots \\ g_m \end{array} \right] .
\]
We wish to determine the image space and kernel of $T$. Note that the image is the $R$-submodule $I$ generated by \{${g}_1,\ldots, {g}_m$\} in $R^s$ while the kernel of $T$ corresponds to the $({g}_1,\ldots, {g}_m)$-syzygy module $H$ in $R^m$.

We fix term orders $\prec_1$ on $R^s$ and $\prec_2$ on $R^m$, and let $u = (f_1,\ldots, f_m) \in R^m$ and $v = T(u) \in R^s$. We redefine $M$ as an $R$-submodule of $R^m \times R^s$ so that

$$M = \{(u,v) \in R^m \times R^s : T(u) = v\}.$$ 

We continue to use $E_i$, $1 \leq i \leq m$ as the $i^{th}$ unit vector in $R^m$, but to avoid confusion we use $F_j$, $1 \leq j \leq s$ as the $j^{th}$ unit vector in $R^s$. And now, the $R$-module $M$ is generated by

$$(E_1, {g}_1),(E_2, {g}_2),\ldots,(E_m, {g}_m).$$

By now it should be clear that the GVW algorithm is a special case of this situation where $s = 1$ and is immediately applicable. The only differences that arise in this general case are in dealing with the leading monomials of the $v$ part. Suppose $(u_1, v_1)$ and $(u_2, v_2)$ are two pairs in $R^m \times R^s$, with $x^\alpha F_j = \text{lm}(v_1)$ and $x^\beta F_k = \text{lm}(v_2)$. We consider $(u_2, v_2)$ as a candidate to top-reduce $(u_1, v_1)$ only if $j = k$. Also, we only calculate the J-pair between $(u_1, v_1)$ and $(u_2, v_2)$ if $j = k$. In this case, assuming $v_1, v_2 \neq 0$, we have

$$t = \text{lcm}(x^\alpha, x^\beta), \quad t_1 = \frac{t}{x^\alpha}, \quad t_2 = \frac{t}{x^\beta},$$

and if $t_1 u_i = \max\{t_1 u_1, t_2 u_2\}$, then $t_i(u_i, v_i)$ is a J-pair. Everything else proceeds as before.

### 3.5 Conclusions

We have presented a simple and fast algorithm for computing Gröbner bases for ideals and modules (including syzygy modules). Our algorithm is more flexible than F5 and our previous algorithm G2V [19] in that we allow a Gröbner basis to be computed incrementally, in one-shot, or a hybrid of the two. It is in this flexibility that we achieve an efficiency boost over G2V as some monomial orderings perform better than others. Indeed, the $g2$ ordering performs better than others and is suggested for general use.
In terms of simplicity, GVW is as simple as Buchberger’s algorithm making implementation an easy matter. In terms of speed, we have shown that GVW derives its efficiency from the use of the syzygy module in preventing future reductions to zero and allowing GVW to outperform other known algorithms by at least a factor of 4 to 20 times. We believe that F4 style fast reductions are possible within the context of our algorithm, but the question remains as to how to implement it efficiently.
Algorithm for computing Gröbner bases

<table>
<thead>
<tr>
<th>Step 0.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variables:</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Step 1.</strong></td>
</tr>
<tr>
<td><strong>Step 2.</strong></td>
</tr>
<tr>
<td><strong>Step 3a.</strong></td>
</tr>
<tr>
<td><strong>Step 3b.</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Step 4.</strong></td>
</tr>
<tr>
<td><strong>Return:</strong></td>
</tr>
</tbody>
</table>

Figure 3.1: The GVW algorithm
Algorithm for computing Gröbner bases in quotient rings

<table>
<thead>
<tr>
<th>Input:</th>
<th>$G = {f_1, \ldots, f_k}$ a Gröbner basis for an ideal $J \subset R$, $g_1, \ldots, g_m$ polynomials in $R$ in normal form modulo $G$, and term orders for $R$ and $R^m$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>A Gröbner basis for $(g_1, \ldots, g_m) \in R/J$ and a Gröbner basis for $\text{lm}(H)$, the leading terms of the syzygy module.</td>
</tr>
<tr>
<td>Variables:</td>
<td>$U$ a list of terms $T_i$, representing signatures of $(u_i, v_i) \in M$. $V$ a list of polynomials $v_i$ for $(u_i, v_i) \in M$; $H$ a list of $\text{lm}(u)$ where $u \in R^m$ is a syzygy found so far, $JP$ a list of pairs $(t, i)$, where $t$ is a monomial so that $t(u_i, v_i)$ is a J-pair of $(u_i, v_i)$ and $(u_j, v_j)$ for some $j \neq i$. We shall refer $(t, i)$ as the J-pair of $(u_i, v_i)$ and $(u_j, v_j)$.</td>
</tr>
<tr>
<td>Step 0.</td>
<td>$U = [0, \ldots, 0]$ with length $k$, and $V = [f_1, \ldots, f_k]$ (so that $(u_i, v_i) = (0, f_i)$, $1 \leq i \leq k$); $JP = []$ and $H = []$, empty lists; Add $E_i$ to $U$ and $g_i$ to $V$ for $1 \leq i \leq m$ (so that $(u_{k+i}, v_{k+i}) = (E_i, g_i)$); Find the leading terms of the principle syzygies $g_j E_i - g_i E_j$ for $1 \leq i &lt; j \leq m$, and add them to $H$; For each $1 \leq i \leq m$, and $1 \leq j &lt; k + i$ compute the J-pair of the two pairs $(u_{k+i}, v_{k+i}) = (E_i, g_i)$ and $(u_j, v_j)$, inserting it into $JP$ whenever the J-signature is not reducible by $G$ or $H$ (storing only one J-pair for each distinct J-signature).</td>
</tr>
<tr>
<td>Step 1.</td>
<td>Take a minimal (in signature) pair $(t, i)$ from $JP$, and delete it from $JP$.</td>
</tr>
<tr>
<td>Step 2.</td>
<td>Reduce the pair $t(T_i, v_i)$ repeatedly by the pairs in $(U, V)$, using regular top-reductions, until it is not regular top-reducible, say to get $(T, v)$</td>
</tr>
<tr>
<td>Step 3a.</td>
<td>If $v = 0$, then append $T$ to $H$ and delete every J-pair $(t, j)$ in $JP$ whose signature $tT_j$ is divisible by $T$.</td>
</tr>
<tr>
<td>Step 3b.</td>
<td>If $v \neq 0$ and $(T, v)$ is not super top-reducible by $(U, V)$, then</td>
</tr>
<tr>
<td></td>
<td>i) append $T$ to $U$ and $v$ to $V$,</td>
</tr>
<tr>
<td></td>
<td>ii) form new J-pairs of $(T, v)$ and $(T_j, v_j)$, $1 \leq j \leq</td>
</tr>
<tr>
<td></td>
<td>iii) insert into $JP$ all such J-pairs whose signatures are not reducible by $G$ or $H$ (storing only the J-pair with minimal $\text{lm}(v)$ for each distinct J-signature $T$).</td>
</tr>
<tr>
<td></td>
<td>iv) Add the leading terms of the principle syzygies $vT_j - v_j T$ for $1 \leq j \leq</td>
</tr>
<tr>
<td>Step 4.</td>
<td>While $JP$ is not empty, go to step 1.</td>
</tr>
<tr>
<td>Return:</td>
<td>$V$ and $H$.</td>
</tr>
</tbody>
</table>

Figure 3.2: the GVW algorithm applied to quotient rings
Chapter 4

An Improved GVW Algorithm

The following is joint work with Shuhong Gao, Lei Huang, Jeffrey Stroomer, and Mingsheng Wang.

4.1 Introduction

Algorithm G2V described in chapter 2 is a special case of algorithm GVW of chapter 3. In this chapter, we present a theoretical foundation for an improved GVW algorithm. This theory tells us how to improve the GVW algorithm so that a minimal Gröbner basis is computed without any extra reductions to zero.

Let \( R = \mathbb{F}[x_1, \ldots, x_n] \) be a polynomial ring over a field \( \mathbb{F} \) with \( n \) variables. Given polynomials \( g_1, \ldots, g_\ell \in R \), we wish to compute a Gröbner basis for the ideal

\[
I = \langle g_1, \ldots, g_\ell \rangle = \{u_1g_1 + \cdots + u_\ell g_\ell : u_1, \ldots, u_\ell \in R\} \subseteq R
\]  

(4.1)

with respect to an arbitrary term order on \( R \). Define

\[
H = \{(u_1, \ldots, u_\ell) \in R^\ell : u_1g_1 + \cdots + u_\ell g_\ell = 0\},
\]  

(4.2)

called the syzygy module of \( g = (g_1, \ldots, g_\ell) \).

Fix any compatible term orders \( \prec_1 \) on \( R \) and \( \prec_2 \) on \( R^\ell \). The specific compatibility that we require is that \( x^\alpha \prec_1 x^\beta \) if and only if \( x^\alpha \mathbf{E}_i \prec_2 x^\beta \mathbf{E}_i, \ 1 \leq i \leq \ell \). For any \( (u, v) \in R^\ell \times R \), we call
lm(u) the signature of (u, v), and we call lm(v) the leading monomial of (u, v). To capture both pieces of information, we call \( \text{lm}(u, v) = (\text{lm}(u), \text{lm}(v)) \), the leading monomial pair of (u, v). We call the pair (u, v) non-syzygy when the leading monomial of (u, v) is nonzero. Any non-syzygy pair is assumed to be a non-trivial pair, i.e. not equal to (0, 0).

Next we define top-reductions in \( R^\ell \times R \) a little differently than chapters 2 and 3. Let \((u, v), (u_1, v_1) \in R^\ell \times R \) be any two pairs. When \( v_1 \) is nonzero, we say (u, v) is top-reducible by (u_1, v_1) if the following two conditions are satisfied:

(i) \( v \) is nonzero and \( \text{lm}(v_1) \) divides \( \text{lm}(v) \); and

(ii) \( t \text{lm}(u_1) \preceq \text{lm}(u) \) where \( t = \text{lm}(v)/\text{lm}(v_1) \).

The corresponding top-reduction is then

\[
(u, v) - ct(u_1, v_1) = (u - ctu_1, v - ctv_1),
\]

where \( c = \text{lc}(v)/\text{lc}(v_1) \). The effect of a top-reduction is that the leading monomial in the v-part is canceled without increasing the signature of (u, v). Such a top-reduction is called regular, if

\[
t \text{lm}(u_1) \prec \text{lm}(u),
\]

and super otherwise. So the signature of (u, v) remains the same under a regular top-reduction but may become smaller under a super top-reduction. When \( v_1 = 0 \), we say that (u, v) is top-reducible by (u_1, 0) if u and u_1 are both nonzero and \( \text{lm}(u_1) \) divides \( \text{lm}(u) \). In this case, we could use (u_1, 0) to reduce the signature of (u, v) without increasing the leading term of v (even if \( v = 0 \)); such a top-reduction is always called super. We note that a pair (u, 0) is never top-reducible by (u_1, v_1) for \( v_1 \neq 0 \). In our algorithm below, we only detect super top-reductions of the two kinds defined here, but never actually perform super top-reductions.

The difference between this definition and that of previous chapters is that this definition does not use a coefficient to discriminate between a regular and super top-reduction. Removing this special case puts this definition of regular top-reduction more in line with that of F5 style algorithms. While the change has no practical effect on the algorithm, the removal of this one special case makes the theory much easier to prove.
One other change from previous chapters is in our definition of a J-pair. Suppose \((u_1, v_1), (u_2, v_2) \in R^\ell \times R\) are two pairs with \(v_1\) and \(v_2\) both nonzero. Let

\[
t = \text{lcm}(\text{lm}(v_1), \text{lm}(v_2)), \quad t_1 = \frac{t}{\text{lm}(v_1)}, \quad t_2 = \frac{t}{\text{lm}(v_2)}.
\]

Suppose \(t_1 \text{lm}(u_1) \neq t_2 \text{lm}(u_2)\) and \(\max(t_1 \text{lm}(u_1), t_2 \text{lm}(u_2)) = t_i \text{lm}(u_i)\) where \(i = 1\) or \(2\). Then

- \(t_i(u_i, v_i) = (t_iu_i, t_iv_i)\) is called a J-pair of \((u_1, v_1)\) and \((u_2, v_2)\);
- \(t_i \text{lm}(u_i)\) is called the J-signature of \((u_1, v_1)\) and \((u_2, v_2)\);

where \(J\) means “joint” of the two pairs. When \(t_1 \text{lm}(u_1) = t_2 \text{lm}(u_2)\), we do not define the J-pair – this is the only difference between this and previous definitions. The proof of theorem 4.11 explains why “J-pairs” of this form needn’t be defined. Also, we mention that lemma 3.1 still applies under these altered definitions.

We note the following transitivity result to simplify what is to come.

**Lemma 4.1.** Suppose \((u, v), (u_1, v_1), (u_2, v_2) \in M\) are either all syzygies or none of them are syzygies. If \((u_2, v_2)\) top-reduces \((u_1, v_1)\) and \((u_1, v_1)\) top-reduces \((u, v)\), then \((u_2, v_2)\) top-reduces \((u, v)\).

**Proof.** If \(v, v_1, v_2 = 0\) then the result is immediate. We suppose that none of \(v, v_1, v_2\) are zero. Let \(t_2 = \text{lm}(v_1)/\text{lm}(v_2)\) and note that \(t_2 \text{ lm}(u_2) \preceq \text{lm}(u_1)\). Also, let \(t_1 = \text{lm}(v)/\text{lm}(v_1)\) and note that \(t_1 \text{ lm}(u_1) \preceq \text{lm}(u)\). Then \(t_1t_2 = \text{lm}(v)/\text{lm}(v_2)\) and \(t_1t_2 \text{ lm}(u_2) \preceq t_1 \text{ lm}(u_1) \preceq \text{lm}(u)\). Thus \((u_2, v_2)\) top-reduces \((u, v)\). \(\square\)

### 4.2 Theoretical Foundation

We call two pairs \((u_1, v_1), (u_2, v_2) \in M\) equivalent if their leading monomial pairs are equal, that is

\[(\text{lm}(u_1), \text{lm}(v_1)) = (\text{lm}(u_2), \text{lm}(v_2)).\]

Thus equivalent pairs are interchangeable when we are only concerned with leading monomials. Given some \((u, v) \in M\), we say that \((u, v)\) is semi-irreducible if it is not regular top-reducible by any pair in \(M\), and we say that \((u, v)\) is irreducible if it is not top-reducible by any pair in \(M\).
with different signature. Stated differently, a \((u, v)\)-pair is irreducible if it is top-reducible only by pairs in \(M\) that are equivalent to itself. For clarity in what is to follow, we provide the following summary of relationships among these terms.

- When an arbitrary pair \(\mathbf{(u, v)}\) is regular top-reduced as much as possible by the irreducible pairs in \(M\), it is then semi-irreducible.
- Every semi-irreducible pair in \(M\) is equivalent to a (possibly trivial) monomial multiple of some irreducible pair in \(M\).

**Lemma 4.2.** Let \(\mathbf{(u_1, v_1), (u_2, v_2)} \in M\) so that \(\text{lm}(u_1) = \text{lm}(u_2)\) and \(\text{lm}(v_2) \prec \text{lm}(v_1)\). Then \(\mathbf{(u_1, v_1)}\) is regular top-reducible by some pair in \(M\).

**Proof.** Let \(\mathbf{(u, v)} = (u_1, v_1) - (u_2, v_2)\), then \(\text{lm}(u) \prec \text{lm}(u_1)\) while \(\text{lm}(v) = \text{lm}(v_1)\). Thus \(\mathbf{(u, v)} \in M\) will regular top-reduce \((u_1, v_1)\).

The above lemma implies that any two irreducible (or semi-irreducible) pairs in \(M\) with the same signature must have the same leading monomial. Thus, we call a signature \(s\) irreducible if there is an irreducible pair in \(M\) whose signature is \(s\). It also explains why the algorithm only keeps one J-pair per J-signature.

**Lemma 4.3.** Every nonzero syzygy \(\mathbf{(u, 0)} \in M\) is top-reducible by an irreducible syzygy. Every non-syzygy pair \(\mathbf{(u, v)} \in M\) is top-reducible by a non-syzygy irreducible pair. Thus, every nonzero \(\mathbf{(u, v)} \in M\) is top-reducible by an irreducible pair.

**Proof.** Let \(\mathbf{(u, 0)} \in M\) be any nontrivial syzygy. Of all the syzygies in \(M\) that will top-reduce \(\mathbf{(u, 0)}\), the one with smallest signature is necessarily irreducible.

Suppose \(\mathbf{(u, v)} \in M\) is a non-syzygy minimal signature counterexample to the second statement. As \(\mathbf{(u, v)}\) is not an irreducible pair, it must be top-reducible by another pair \(\mathbf{(u_1, v_1)} \in M\) of strictly smaller signature, \(\text{lm}(u_1) \prec \text{lm}(u)\). We first consider the case where \(v_1 \neq 0\). Then as \(\mathbf{(u_1, v_1)}\) is a non-syzygy pair with strictly smaller signature, it is top-reducible by a non-syzygy irreducible pair \(\mathbf{(u_2, v_2)}\). But as non-syzygy top-reductions are transitive (lemma 4.1), \(\mathbf{(u_2, v_2)}\) must also top-reduce \((u, v)\), and it must be the case that \(\mathbf{(u_1, v_1)} = \mathbf{(u_1, 0)}\) is a syzygy.

Now there exists some monomial \(t > 1\) such that \(\text{lm}(u) = t \text{lm}(u_1)\). We use this to produce a new pair \(\mathbf{(u_2, v)} = (u, v) - t(u_1, 0) \in M\) with signature strictly smaller (but nonzero) than that
of \((u, v)\). Thus \((u_2, v)\) is top-reducible by some irreducible non-syzygy pair \((u_3, v_3) \in M\). Again, as \((u_3, v_3)\) will also top-reduce \((u, v)\), this case is impossible. Therefore no such counterexample exists.

**Corollary 4.4.** Whenever a non-syzygy pair \((u, v) \in M\) is regular top-reducible, it is also regular top-reducible by an irreducible pair.

**Lemma 4.5.** No semi-irreducible non-syzygy pair of \(M\) is super top-reducible by any syzygy in \(M\). Consequently, every nonzero semi-irreducible pair is equivalent to a (possibly trivial) monomial multiple of some irreducible pair.

**Proof.** Suppose that \((u, v) \in M\) is super top-reducible by \((u_1, 0) \in M\) so that \(\text{lm}(u) = t \text{lm}(u_1)\). We set \((u_2, v) = (u, v) - t(u_1, 0) \in M\) and note that either \(v = 0\) or \((u, v)\) is regular top-reducible by \((u_2, v)\). Therefore, semi-irreducible non-syzygy pairs are not super top-reducible by syzygies. The second statement follows trivially for syzygies. For non-syzygies, there are exactly three cases for top-reductions, and semi-irreducibility rules out two of them.

Now let
\[
(u_1, v_1), \ldots, (u_k, v_k)
\]
be a list of pairs in \(M\). The list (4.3) is called a strong Gröbner basis for \(M\) if every nonzero pair \((u, v) \in M\) is top-reducible by some pair in (4.3).

**Proposition 4.6.** Suppose that the list of pairs in (4.3) is a strong Gröbner basis for \(M\). Then

1. \(G_0 = \{u_i : v_i = 0, 1 \leq i \leq k\}\) is a Gröbner basis for the syzygy module of \(g = (g_1, \ldots, g_\ell)\), and
2. \(G_1 = \{v_i : 1 \leq i \leq k\}\) is a Gröbner basis for \(I = \langle g_1, \ldots, g_\ell \rangle\).

**Proof.** For any \(u = (u_1, \ldots, u_\ell)\) in the syzygy module of \(g\), we have \((u, 0) \in M\). By our assumption, \((u, 0)\) is top-reducible by some pair \((u_i, v_i)\) in (4.3). Then we must have \(v_i = 0\), thus \(u_i \in G_0\) and \(\text{lm}(u)\) is reducible by \(\text{lm}(u_i)\). This proves that \(G_0\) is a Gröbner basis for the syzygy module of \(g\).

Now suppose \(v \in I\) and is nonzero. Then there exists \(u = (u_1, \ldots, u_\ell) \in \mathbb{R}^\ell\) so that \(ug^t = v\), hence \((u, v) \in M\). Among all such \(u\), we pick one so that \(\text{lm}(u)\) is minimum. Since \((u, v) \in M\), it is top-reducible by some \((u_i, v_i)\) where \(1 \leq i \leq k\). If \(v_i = 0\), then we could use \((u_i, 0)\) to reduce \((u, v)\) to get a \(u'\) so that \(u'g^t = v\) and \(\text{lm}(u')\) is smaller than \(\text{lm}(u)\), contradicting to the minimality of \(\text{lm}(u)\). So \(v_i \neq 0\) and \(\text{lm}(v_i)\) divides \(\text{lm}(v)\). Hence \(G_1\) is a Gröbner basis for \(I\).
Lemma 4.7. Let $G \subseteq M$. The following are equivalent.

1. $G$ is a strong Gröbner basis for $M$.

2. $\text{Imp}(G)$ contains all irreducible leading monomial pairs in $M$.

Proof. Suppose that $\text{Imp}(G)$ contains all irreducible leading monomial pairs in $M$. Given any nonzero $(u, v) \in M$, by lemma 4.3 it is top-reducible by some irreducible pair in $M$. Thus it is top-reducible by some pair in $G$.

Now, suppose that $G$ is a strong Gröbner basis for $M$. Given any irreducible pair $(u, v) \in M$, it is only top-reducible by an equivalent pair. Therefore $\text{Imp}(u, v) \in \text{Imp}(G)$.

Thus, a complete list of irreducible pairs is a strong Gröbner basis for $M$. Moreover, every irreducible pair (up to equivalence) must be contained within a strong Gröbner basis. Therefore, we call the irreducible pairs in $M$ (up to equivalence) a minimal strong Gröbner bases for $M$.

Next, we show that a minimal strong Gröbner basis is always finite. To this end, we begin thinking of leading monomial pairs $(x^\alpha E_i, \text{lm}(v))$, $1 \leq i \leq \ell$ as monomials in $n(\ell + 1)$ variables, and we say $\text{Imp}(u_1, v_1)$ divides $\text{Imp}(u, v)$ to mean that $\text{lm}(u_1)$ divides $\text{lm}(u)$ and $\text{lm}(v_1)$ divides $\text{lm}(v)$.

For the following, we recall the restriction on our term orderings. We require that $\prec_1$ and $\prec_2$ are compatible term orders, that is, $m_1 \prec_1 m_2$ if and only if $m_1 E_i \prec_2 m_2 E_i$ for all $1 \leq i \leq \ell$.

Lemma 4.8. Suppose $\prec_1$ and $\prec_2$ are compatible term orders, and $(u, v), (u_1, v_1) \in M$ are non-equivalent pairs such that $(u, v)$ is semi-irreducible and $\text{Imp}(u_1, v_1)$ divides $\text{Imp}(u, v)$. Then $\text{lm}(u)$ is not an irreducible signature.

Proof. Suppose that for $(u, v), (u_1, v_1) \in M$, $\text{Imp}(u_1, v_1)$ divides $\text{Imp}(u, v)$. We let monomials $s$ and $m$ be such that $\text{lm}(u) = s \text{lm}(u_1)$ and $m = \text{lm}(v) / \text{lm}(v_1)$. We note that $s$ and $m$ may be equal but not both constant. If $s \geq_1 m$, then $(u_1, v_1)$ will top-reduce $(u, v)$ as $m \text{lm}(v_1) = \text{lm}(v)$ but $m \text{lm}(u_1) \leq_2 s \text{lm}(u_1) = \text{lm}(u)$. On the other hand, if $s \prec_1 m$, then $(u_2, v_2) = (u, v) - s(u_1, v_1)$ will regular top-reduce $(u, v)$ as $\text{lm}(v_2) = \text{lm}(v)$ and $\text{lm}(u_2) \prec_2 \text{lm}(u)$. Finally, as $(u, v)$ is semi-irreducible but super top-reducible, $(u, v)$ is not an irreducible pair and $\text{lm}(u)$ is not an irreducible signature.

Theorem 4.9. For any compatible term orders on $R^\ell$ and $R$, the set of irreducible signatures of $M$ is finite.
Proof. We view leading monomial pairs as monomials in the \( n(\ell + 1) \) variables: \( x_{01}, \ldots, x_{0n}, x_{11}, \ldots, x_{1n}, \ldots, x_{\ell 1}, \ldots, x_{\ell n} \). Let

\[ I = \langle \text{lmp}(M) \rangle \subseteq F[x_{01}, \ldots, x_{0n}, x_{11}, \ldots, x_{1n}, \ldots, x_{\ell 1}, \ldots, x_{\ell n}] \]

be the monomial ideal of leading monomial pairs of \( M \). Dickson’s lemma asserts that \( I \) is finitely generated, that is \( I = \langle \text{lmp}(u_i, v_i) \rangle \) for \( (u_i, v_i) \in M, 1 \leq i \leq k \). Given any irreducible pair \( (u, v) \in M, \text{lmp}(u, v) \in I \) must equal \( \text{lmp}(u_i, v_i) \) for some \( 1 \leq i \leq k \), since otherwise lemma 4.8 prevents \( (u, v) \) from being irreducible. Hence, there are at most \( k \) distinct irreducible signatures.

We remark that it is lemma 4.8 that requires the compatible term orders. Without such a restriction on \( \prec_1 \) and \( \prec_2 \), the divisibility described in lemma 4.8 fails, and one can construct relatively simple examples with an infinite number of irreducible signatures [22].

**Lemma 4.10.** The \( E_1, \ldots, E_\ell \) are irreducible signatures.

**Proof.** Let \( (u, v) \) be a semi-irreducible pair with signature \( E_i \) for some \( 1 \leq i \leq \ell \). As it is not possible to super top-reduce \( (u, v) \) by anything with smaller signature, \( (u, v) \) is irreducible. \( \square \)

One must be a little careful though, as \( (E_1, g_1), \ldots, (E_\ell, g_\ell) \) are not necessarily irreducible pairs, even when the \( g_1, \ldots, g_\ell \) are interreduced. To see this, consider a counterexample. Suppose \( \langle g_1, \ldots, g_{\ell - 1} \rangle = \langle 1 \rangle = R = F[x_1, \ldots, x_n] \). Also, suppose we are using \( \succ_{\text{pot}} \) with \( E_i \prec_{\text{pot}} E_j \) whenever \( 1 \leq i < j \leq \ell \). Then as the algorithm will process J-pairs in increasing order of signature, it will not encounter the pair \( (E_\ell, g_\ell) \) until a Gröbner basis for \( \langle g_1, \ldots, g_{\ell - 1} \rangle = R \) has been computed. Thus the irreducible pair having \( E_\ell \) as a signature will be a syzygy.

**Theorem 4.11.** Let \( s \) be an irreducible signature such that \( s \neq E_i \) for any \( 1 \leq i \leq \ell \). Then \( s \) is the signature of the J-pair of two non-syzygy irreducible pairs with smaller signatures.

**Proof.** As \( E_i, 1 \leq i \leq \ell \) are irreducible signatures, we may find some monomial \( m \succ 1 \) and irreducible pair \( (u, v) \) such that \( m(u, v) \) has signature \( s \). Of all such pairs, we choose the one with \( m \text{Im}(v) \) minimal. Note that \( mv \neq 0 \) or else \( s \) would not be an irreducible signature. As \( s = m u \) is an irreducible signature and \( m(u, v) \) is super top-reducible by \( (u, v) \), it must be the case that \( m(u, v) \) is not semi-irreducible. By corollary 4.4, \( m(u, v) \) is regular top-reducible by some irreducible
pair \((u_1, v_1) \in M\). By lemma 3.1, the J-pair of \((u, v)\) and \((u_1, v_1)\) is defined and must be of the form \(t(u, v)\) where \(t \mid m\). We suppose, for the sake of contradiction, that \(t\) properly divides \(m\). After a sequence of regular top-reductions on \(t(u, v)\), we get a semi-irreducible pair \((u_2, v_2) \in M\) with signature \(\text{lm}(u_2) = t \text{lm}(u)\) and \(\text{lm}(v_2) < t \text{lm}(v)\). Thus, by lemma 4.5, \((u_2, v_2)\) is equivalent to \(m'(u_3, v_3)\), a monomial multiple of some irreducible pair \((u_3, v_3) \in M\) and therefore \((m/t)m'(u_3, v_3)\) has signature \(s\) but smaller leading monomial than \(m \text{lm}(v)\),

\[
\begin{align*}
(m/t)m' \text{lm}(u_3) & = (m/t)\text{lm}(u_2) = m \text{lm}(u) = s \\
(m/t)m' \text{lm}(v_3) & = (m/t)\text{lm}(v_2) < m \text{lm}(v),
\end{align*}
\]

a contradiction to our choice of \(m(u, v)\) in the minimality of \(m \text{lm}(v)\). Hence, \(t = m\), and the signature \(s\) was obtained from the J-pair of two non-syzygy irreducible pairs of strictly smaller signature.

\[\square\]

Notice that in calculating the maximum of weighted signatures for the J-pair of \((u, v)\) and \((u_1, v_1)\), there was no tie as \((u_1, v_1)\) regular top-reduces the J-pair \(m(u, v)\). Thus, we do not define J-pairs that are not well defined from two irreducible pairs. We can not, however, rule out the J-signature as being irreducible as it may appear from some other J-pair calculation.

**Corollary 4.12.** We may obtain the irreducible pairs by processing a finite number of J-pairs.

**Proof.** A J-pair needs only to be calculated between irreducible pairs. Thus, if there are \(k\) irreducible pairs, then at most \(\binom{k}{2}\) J-pairs require processing. \(\square\)

We have seen that the J-signatures of irreducible pairs give new irreducible signatures. But not every J-signature is irreducible. Currently, one is required to fully regular top-reduce any J-pair to determine whether the resulting semi-irreducible pair is irreducible. The following criterion circumvents the necessity of regular top-reducing non-irreducible pairs.

Let \(s\) be any signature and \(P\) be the set of irreducible pairs with signature strictly smaller than \(s\). Of all the \((u, v) \in P\) that can be scaled to have signature \(s = m \cdot \text{lm}(u)\) (always with \(m > 1\)), the one with the smallest \(m \cdot \text{lm}(v)\) is called the M-pair for \(s\). When speaking in general terms, we say that \((u, v)\) is an M-pair if it is the M-pair for its signature \(\text{lm}(u)\). The following results characterize exactly when M-pairs and J-pairs are have irreducible signatures.
Lemma 4.13. An M-pair \((u, v)\) has an irreducible signature if and only if \((u, v)\) is not semi-irreducible.

Proof. For the forward direction, suppose that \(m(u, v), m > 1\) is an M-pair for \(m \cdot \text{lm}(u)\). If \(m(u, v)\) is semi-irreducible, then \(m(u, v)\) is super top-reducible by \((u, v)\). Thus if \(m(u, v)\) has an irreducible signature, then \(m(u, v)\) is regular top-reducible by some pair in \(M\).

For the reverse direction, suppose that \(m(u, v)\) is a regular top-reducible M-pair for \(m \cdot \text{lm}(u)\) so that \(m > 1\) and \((u, v)\) is irreducible. Suppose that \((u_1, v_1)\) is a semi-irreducible pair with signature \(m \cdot \text{lm}(u)\) but \(\text{lm}(v_1) < \text{lm}(v)\). Lemma 4.5 provides that \(\text{Imp}(u_1, v_1) = m_2 \cdot \text{Imp}(u_2, v_2)\) for some irreducible pair \((u_2, v_2)\). If \(m_2 > 1\), then as \(m_2(u_2, v_2)\) has the same signature as \(m(u, v)\) but smaller leading monomial, the choice of \(m(u, v)\) as M-pair for \(m \cdot \text{lm}(u)\) is contradicted. Thus, \(m_2 = 1\), \((u_1, v_1)\) is an irreducible pair, and \(m \cdot \text{lm}(u)\) is an irreducible signature. Therefore, not semi-irreducible M-pairs have irreducible signatures. \(\square\)

Lemma 4.14. Suppose \((u, v)\) is a non-syzygy M-pair for \(\text{lm}(u)\). Then \(\text{lm}(u)\) is an irreducible signature if and only if \(\text{lm}(u)\) is a J-signature of two irreducible pairs (of smaller signature).

Proof. The reverse direction follows from lemma 4.13 and the fact that every J-pair is regular top-reducible. For the forward direction, we suppose that \(m(u, v)\) is an M-pair for irreducible signature \(m \cdot \text{lm}(u)\). Lemma 4.13 requires that \(m(u, v)\) be regular top-reducible by some irreducible \((u_1, v_1)\). If we compute the J-pair between \((u, v)\) and \((u_1, v_1)\), we get something of the form \(m_1(u, v)\) where \(m_1\) divides \(m\) by lemma 3.1.

We suppose for the sake of contradiction that \(m_1\) properly divides \(m\). Letting \((u_2, v_2)\) be a semi-irreducible pair with the same signature as \(m_1(u, v)\), we see that the signatures of \((m/m_1)(u_2, v_2)\) and \((m/m_1)m_1(u, v)\) match, but the leading monomial of \((m/m_1)(u_2, v_2)\) is strictly smaller than \((m/m_1)m_1(u, v)\) (as \(m_1(u, v)\) is regular top-reducible by \((u_1, v_1)\)). The pair \((m/m_1)(u_2, v_2)\) contradicts the fact that \(m(u, v)\) is an M-pair for \(m \cdot \text{lm}(u)\).

Thus \(m_1 = m\). Note that the signatures of \((u, v)\) and \((u_1, v_1)\) are both strictly smaller than that of \(m(u, v)\). Also, as \((u_1, v_1)\) regular top-reduces \(m(u, v)\) (both of which are non-syzygy), the J-pair between \((u, v)\) and \((u_1, v_1)\) is defined. Therefore, \(m(u, v)\) is the J-pair of two irreducible pairs of smaller signature. \(\square\)

The notion and usefulness of M-pairs in the above form are due to Huang [22], but an equivalent condition was discovered independently by Arri [1].
4.3 Implementation

As described above, calculating a strong Gröbner basis for $M$ is accomplished by calculating the irreducible pairs of $M$. The following summarizes the results obtained so far.

**Corollary 4.15.** For each $2 \leq i \leq \ell$, the $i^{th}$ initial pair $(E_i, g_i)$ can each be transformed into an irreducible pair with only regular top-reductions by irreducible pairs whose signatures are strictly less than $E_i$. To calculate the irreducible pair for some irreducible signature $s \neq E_i$, $1 \leq i \leq \ell$, one needs only to calculate J-pairs among the irreducible pairs (of signature smaller than $s$) and discard all but the M-pairs. Fully regular top-reducing the M/J-pair with signature $s$ by the irreducible pairs of signature smaller than $s$ yields an irreducible pair with signature $s$.

**Proof.** For the initial pairs, this is a direct application of lemma 4.5. Only needing to calculate J-pairs among irreducible pairs whose signatures are strictly less is a rewording of theorem 4.11. One needs only fully regular top-reduce a given J-pair by irreducible pairs with smaller signature to obtain a semi-irreducible pair. When a J-pair that is not an M-pair is fully regular top-reduced, the result is semi-irreducible but not irreducible. Thus, only the M-pairs need to be top-reduced. □

Lemma 4.14 tells us exactly when a signature is irreducible. It is here that the GVW algorithm can be improved. We provide the new form of the algorithm in figure 4.1. When multiple J-pairs are found with the same signature, we store only the one with minimal $v$-part. Then in step 2 of figure 4.1, we proceed to regular top-reduce the J-pair only if the J-pair is also an M-pair.

We note that with each execution of step 3 of figure 4.1, a new irreducible pair is produced, whether it is a syzygy (step 4a) or otherwise (4b). Thus, the number of times step 3 is executed is bounded from above by the size of a minimal strong Gröbner basis. Next, we notice that step 4bi obtains new syzygies with every new $(u, v)$-pair generated. For every irreducible syzygy discovered by step 4bi, an execution of steps 3 and 4a is avoided. Thus, another reduction to zero is avoided. Finally, figure 4.1 executes line 4a fewer times than the size of a minimal Gröbner basis for the syzygy module and therefore executes line 3 fewer times than the size of a minimal strong Gröbner basis!

Figure 4.1 has only a few differences from our earlier version of GVW. First, our notion of regular top-reducibility changed a little. In chapter 3, we called a top-reduction regular if it preserved the signature of the pair being reduced. There were two cases that allowed this to happen. The first is the regular top-reducibility described in this chapter. The other is when the coefficients
of the \( v \)-part differed.

Next, we no longer define J-pairs when the J-pair calculation between \((u_i, v_i)\) and \((u_j, v_j)\) involves a tie as in \( t_i \text{lm}(u_i) = t_j \text{lm}(u_j) \). In the current chapter, theorem 4.11 and lemma 4.14 guarantee that such tied J-pairs are not required. In chapters 2 and 3, we had no such result. Ignoring such J-pairs prevents needless regular top-reductions to zero and the discarding of eventually super top-reducible pairs (as defined in chapter 3).

Another difference in terminology relates to “eventual super top-reducibility”. In chapter 3, we say that a \((u, v)\)-pair is eventually super top-reducible by a set \( S \) if \((u, v)\) is regular top-reduced by a sequence of pairs from \( S \) to get \((u_1, v_1)\) which is not regular top-reducible by \( S \) but is super top-reducible by some pair in \( S \). In the language of this chapter, we would say that the semi-irreducible pair \((u_1, v_1)\) is not irreducible (always with respect to \( M \)). It is lemmas 4.1 and 4.3 that allow this restatement. We believe this newer characterization provides a better understanding of why the algorithm works, while the chapter 3 characterization better describes how the algorithm performs the detection of redundant pairs.

Also, to be consistent with previous chapters, the algorithms presented here only retain signatures. If one needed the minimal Gröbner basis for the syzygy module, a slight modification to the algorithms would provide the generators. In earlier chapters (specifically section 3.3.2), the modification was not so immediate as there was no reason to perform the syzygy calculations on an eventually super top-reducible pair. Therefore, we recommended using a two-pass approach: get all the relevant signatures, and rerun the algorithm to compute syzygies for the appropriate signatures. In the current chapter, since figure 4.1 processes exactly the irreducible signatures, there is no wasted work by making the obvious change to retain the entire syzygy.

Step 0 of figures 4.1 and 3.1 are different. The latest version, figure 4.1, initializes the list \( JP \) with all the initial pairs \( g_1, \ldots, g_\ell \) as instructed by corollary 4.15. Doing so produces exactly the irreducible pairs (up to equivalence) of \( M \). Therefore, algorithm as presented in figure 4.1 produces exactly a minimal strong Gröbner basis for \( M \).

### 4.4 Conclusion

In the sense of proposition 4.6, the irreducible pairs are the smallest strong Gröbner basis one could hope to compute. Therefore, we call the irreducible pairs, up to equivalence, a minimal strong
Gröbner basis. Our updated algorithm (figure 4.1) produces exactly a minimal strong Gröbner basis for $M$.

Just as different term orders on $R$ have different sized minimal Gröbner bases for $\langle g_1, \ldots, g_\ell \rangle$, different term orders on $R^\ell$ has different sized minimal strong Gröbner bases on $M$. In table 4.1, we have listed the sizes of minimal strong Gröbner bases on $M$ for the four term orders we have tried (see chapter 3 for the descriptions). As the table makes clear, some term orders produce smaller minimal strong Gröbner bases than others. Thus, one would expect fewer computational resources should be required to arrive at a smaller minimal strong Gröbner basis.

Also, as we were able to implement G2V similarly enough to F5 and F5C, a direct comparison was possible. GVW was already a different enough algorithm that direct comparison was not appropriate. Moreover, since G2V and F5C calculate a reduced Gröbner basis at each iteration, the size of the resulting basis is not comparable with the results presented in table 4.1. We are able, however, to fairly compare the total number of J-pairs or S-polynomials processed throughout the course of the algorithms. Table 4.2 lists the results. GVWHS under the $g_2$ order seems to produce the smallest minimal strong Gröbner bases for the cases considered. Notice that summing the corresponding cells of table 4.1 gives larger values than the corresponding cells of table 4.2.

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>$g_1$</th>
<th>$g_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>67</td>
<td>64</td>
<td>64</td>
<td>27</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>73</td>
<td>91</td>
<td>91</td>
<td>44</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>224</td>
<td>175</td>
<td>175</td>
<td>80</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>448</td>
<td>343</td>
<td>343</td>
<td>151</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>398</td>
<td>133</td>
<td>133</td>
<td>134</td>
</tr>
<tr>
<td>$F_633$ (76)</td>
<td>135</td>
<td>184</td>
<td>170</td>
<td>106</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>155</td>
<td>1189</td>
<td>1189</td>
<td>188</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>749</td>
<td>9237</td>
<td>9237</td>
<td>846</td>
</tr>
</tbody>
</table>

Table 4.1: Sizes of minimal strong Gröbner bases for different term orders
Table 4.2: Counts of the J-pairs or S-polynomials processed by F5, F5C, G2V, and GVWHS under POT, TOP, $g_1$ and $g_2$ orders

Algorithm GVWHS regular top-reduces fewer J-pairs than the size of a minimal strong Gröbner basis.
**Algorithm for computing Gröbner bases**

| Variables: | 
|------------|---------------------------------------------------------------|
| U          | a list of terms $T_i$, representing signatures of $(u_i, v_i) \in M$, |
| V          | a list of polynomials for $v_i$ for $(u_i, v_i) \in M$, |
| H          | a list for $\text{lm}(u)$ were $u \in R$ is a syzygy found so far, |
| JP         | a list of pairs $(x^\alpha T_i, x^\alpha v_j)$, where $x^\alpha$ is a monomial so that $x^\alpha (u_i, v_i)$ is the J-pair of $(u_i, v_i)$ and $(u_j, v_j)$ for some $j \neq i$. |

**Step 0.**

$U = [\ ]$, and $V = [\ ]$. Find the leading terms of the principle syzygies $g_j E_i - g_i E_j$ for $1 \leq i < j \leq \ell$, and add them in $H$. Define $J$-pairs $(E_1, g_1), \ldots, (E_\ell, g_\ell)$ and store them into $JP$.

**Step 1.**

Take a minimal (in signature) pair $(T_1, v_1)$ from $JP$, and delete it from $JP$.

**Step 2.**

If $(T_1, v_1)$ is not an M-pair for $T_1$, go to step 1.

**Step 3.**

Reduce the pair $(T_1, v_1)$ repeatedly by the pairs in $(U, V)$, using regular top-reductions until it is semi-irreducible, say to get $(T, v)$.

**Step 4a.**

If $v = 0$, then append $T$ to $H$, and delete every J-pair $(T_2, v_2)$ in $JP$ whose signature $T_2$ is divisible by $T$.

**Step 4b.**

If $v \neq 0$, then

i) Add the leading terms of the principle syzygies, $vT_j - v_j T$ for $1 \leq j \leq |U|$, to $H$.

ii) Form new J-pairs of $(T, v)$ and $(T_j, v_j)$, $1 \leq j \leq |U|$, and

iii) Insert into $JP$ all such J-pairs that are not reducible by $H$ (storing only one J-pair for each distinct signature of minimal leading monomial).

iv) Append $T$ to $U$ and $v$ to $V$.

**Step 5.**

While $JP$ is not empty, go to step 1.

Return: $V$ and $H$.

---

Figure 4.1: The GVWHS algorithm
Chapter 5

Notes on Implementation

While the algorithms described in figures 2.1, 3.1, 3.2, and 4.1 describe the G2V, GVW, and GVWHS algorithms in enough detail to prove correctness, not enough detail is specified from an implementation point of view. Indeed, there are many choices to be made, and while the choices don’t affect the output (reduced) Gröbner basis, they can have a profound effect on the performance of the algorithm. Our answers to such choices are the detail of section 5.1.

In every version of the algorithms presented, the goal of the outermost while loop is to regular top-reduce polynomials as much as possible. While this is an expensive process, it is necessary to produce the irreducible pairs. Avoiding regular top-reductions to zero is the subject of section 5.2. If the resulting polynomial is no longer regular top-reducible but is super top-reducible, the polynomial is to be discarded (the M-pair criterion when employed prevents this). When dealing with dense polynomials, one would like to avoid performing the top-reductions as much as possible until it is known whether the resulting polynomial is to be retained or discarded. Avoiding such expensive computations is the topic of section 5.3.

Finally, when one adds the field equations $x_i^2 - x_i = 0$, $1 \leq i \leq n$ to the ideal $I \subset \mathbb{F}_2[x_1, \ldots, x_n]$, many time saving changes can be made to the algorithms. In particular, various computations can be avoided and much less memory is required. The representation of monomials and polynomials and changes to the GVW algorithm are discussed in section 5.4.
5.1 Choosing the Next J-Pair

We have already seen in the preceding chapters that J-pairs are always processed in order of increasing signature. We’ve also seen that only one J-pair needs to be retained for every signature. But when we have multiple J-pairs per signature, we need to choose the one to actually process. Although there are certainly many more, we mention eleven policies P1 through P11.

P1. Keep the first J-pair encountered with each signature,
P2. Keep the most recent J-pair encountered with each signature,
P3. Keep the J-pair $t_i(u_i, v_i)$ with the smallest index $i$,
P4. Keep the J-pair $t_i(u_i, v_i)$ with the largest index $i$,
P5. Keep the J-pair $t(u, v)$ with the smallest scaled polynomial $tv$,
P6. Keep the J-pair $t(u, v)$ with the largest scaled polynomial $tv$,
P7. Keep the J-pair $t(u, v)$ with the smallest scale $t$,
P8. Keep the J-pair $t(u, v)$ with the largest scale $t$,
P9. Keep the J-pair $t(u, v)$ where $v$ has the most terms, and
P10. Keep the J-pair $t(u, v)$ where $v$ has the fewest terms,

We implemented policies P1 through P10 for comparison. As guaranteed by lemma 4.2, all ten policies processed the exact same number of J-pairs (for the test cases listed in tables 5.1 and 5.2), but the runtimes and memory usages told a different story. Policies P2, P4, P5, and P7 all performed very similarly (in terms of both runtime and memory usage) and consumed fewer resources than policies P1, P3, P6, and P8 which also performed very similarly (again, in terms of both runtime and memory usage). Interestingly enough, policies P9 and P10 performed very similarly to each other and were both the worst policies observed. Table 5.1 lists the runtimes and memory usages of the GVW algorithm with policy P5 in use, while table 5.2 lists the analogous data with P1 in use. Such policies are easy to implement and compare, see code listing 5.1 for details.
Finally, it is interesting that although the GVW algorithm is invariant (everything outside the top-reduce function) to our choice of policy, the runtimes and memory usages can differ substantially. In the case of P5, lemma 4.2 tells us that we are choosing the pair that is closest to being semi-irreducible, and so P5 chooses the pair that should require the fewest (generally) number of regular top-reductions. Thus in the case of P5, it makes sense that the algorithm would run faster. Perhaps that same intuitive argument can be extended to the other policies since we would expect later J-polynomials to be reduced further than earlier J-polynomials.

Now that we have seen how to process distinct signatures in increasing order, one might wonder if in the course of processing a J-pair \((u_i, v_i)\), another J-pair with the same J-signature might be produced. Specifically, if the J-pair of \((u_i, v_i)\) and \((u_j, v_j)\) for \(j < i\) produces a J-signature equal to \(u_i\). It turns out that this is impossible under a very natural condition, and as a result, the above policies can be implemented in a consistent fashion.

**Proposition 5.1.** Suppose that \((u_i, v_i)\) and \((u_j, v_j)\) are unable to top reduce each other. Then the J-pair of \((u_i, v_i)\) and \((u_j, v_j)\) will have signature strictly larger than both \(u_i\) and \(u_j\).

---

Table 5.1: Runtimes and memory usages for policy P5. These values are also representative of policies P2, P4 and P7

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>0.34</td>
<td>0.36</td>
<td>0.36</td>
<td>0.37</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>3.22</td>
<td>2.91</td>
<td>2.93</td>
<td>3.13</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>1.74</td>
<td>3.66</td>
<td>3.65</td>
<td>3.80</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>0.08</td>
<td>0.44</td>
<td>0.35</td>
<td>0.09</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>0.34</td>
<td>3.25</td>
<td>3.20</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Maximal memory usage in megabytes

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>4.88</td>
<td>4.86</td>
<td>4.86</td>
<td>4.53</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>5.48</td>
<td>6.31</td>
<td>6.31</td>
<td>5.33</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>14.88</td>
<td>13.55</td>
<td>13.53</td>
<td>9.08</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>52.69</td>
<td>45.70</td>
<td>45.81</td>
<td>25.69</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>55.69</td>
<td>18.89</td>
<td>18.91</td>
<td>18.38</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>6.72</td>
<td>10.36</td>
<td>8.61</td>
<td>6.06</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>5.69</td>
<td>20.91</td>
<td>20.80</td>
<td>7.00</td>
</tr>
</tbody>
</table>
Table 5.2: Runtimes and memory usages for policy P1. These values are also representative of policies P3, P6 and P8.

```c++
void enqueue(const j_pair & j) { // enqueue for later processing
    const signature & s = j.uPart();
    j_pair * ptr = index[s];
    if (!ptr) { // First time seeing this signature
        index[s] = new j_pair(j);
        array.push_back(s);
        bubble_up(array.size());
        return;
    }
    // /////////////// insert replacement policy here ///////////////
    // if (false) { // P1
    //   if (true) { // P2
    //     if (ptr->idx() > j.idx()) { // P3
    //       if (ptr->idx() < j.idx()) { // P4
    //         if (ptr->vMonomial() > j.vMonomial()) { // P5 Winner
    //           if (ptr->vMonomial() < j.vMonomial()) { // P6
    //             if (ptr->theScale() > j.theScale()) { // P7
    //               if (ptr->theScale() < j.theScale()) { // P8
    //                 if (ptr->termCount() < j.termCount()) { // P9
    //                   if (ptr->termCount() > j.termCount()) { // P10
    //                     delete ptr;
    //                     index[s] = new j_pair(j);
    //                   }
    //                 }
    //               }
    //             }
    //           }
    //         }
    //       }
    //     }
    //   }
    // }
```

Listing 5.1: The implementation of policies in C++ code
Proof. We shall assume that \( u_i \prec u_j \). Set 
\[
t_i = \frac{\text{lcm}(\text{lm}(v_i), \text{lm}(v_j))}{\text{lm}(v_i)} \quad \text{and} \quad t_j = \frac{\text{lcm}(\text{lm}(v_i), \text{lm}(v_j))}{\text{lm}(v_j)},
\]
so that the J-pair of \((u_i, v_i)\) and \((u_j, v_j)\) will be one of \( t_i(u_i, v_i) \) or \( t_j(u_j, v_j) \).

For the first case, suppose that \( t_j u_j \succ t_i u_i \) so that the J-pair is \( t_j(u_j, v_j) \). If it is the case that \( t_j > 1 \) then there is nothing to show. We therefore assume for the remainder of this case that \( t_j = 1 \). Then we must have that \( \text{lm}(v_i) \mid \text{lm}(v_j) \). Notice that \( t_i = \text{lm}(v_j)/\text{lm}(v_i) \) and recall that \( t_i u_i \prec t_j u_j = u_j \). Thus \((u_i, v_i)\) regular top-reduces \((u_j, v_j)\), a contradiction to the statement of the proposition. Hence \( t_j > 1 \) and the J-signature must be larger than \( u_j \).

For the second case, \( t_i u_i \succ t_j u_j \) (so that the J-pair is \( t_i(u_i, v_i) \)), as \( t_i u_i \succ u_j \succ u_i \) this case is immediate.

For the third and final case (only applies to chapters 2 and 3), suppose that \( t_i u_i = t_j u_j \) so that either \( t_i(u_i, v_i) \) or \( t_j(u_j, v_j) \) can be retained as the J-pair. Just as in the first case above, if \( t_j = 1 \) then \( t_i = \text{lm}(v_j)/\text{lm}(v_i) \). Thus \((u_i, v_i)\) top-reduces (either regular or super depending on the coefficients) \((u_j, v_j)\) which is impossible. Therefore, the only possibility is that \( t_j > 1 \) which gives the result.

With the processing of every J-pair, it becomes fully reduced with respect to the previously stored \((u, v)\)-pairs. Thus, the only time the condition of the proposition is not satisfied is at the very beginning of the algorithms (as described in chapters 2 and 3) when producing J-pairs between the initial generator list.

One may be tempted to then interreduce the generator list before initiating the algorithm, but we have found this to weaken the algorithm’s performance. It would see that processing the initial J-pairs having signatures equal to the generating pairs \((E_i, g_i)\) \(1 \leq i \leq m\) provides many useful syzygies that avoid later computations.

### 5.2 Anticipating Syzygies

Every regular top-reduction to zero is expensive. It provides useful information about the syzygy module and helps prevent similar reductions to zero, but it is still expensive. If we had information about the syzygy module ahead of time, many such reductions to zero could be avoided.
Therefore, it is advantageous to collect as many syzygies as possible from various sources.

With each iteration of the main while-loop, the J-pair of smallest signature is selected and regular top-reduced as much as possible to get \((u, v)\). Suppose the \(U\) and \(V\) currently contain \((u_1, v_1), \ldots, (u_k, v_k)\). Then we have \(k\) new syzygies available, each given by the following.

\[
v_i(u, v) - v(u, v_i) = (v_i u - v u_i, 0), \quad 1 \leq i \leq k
\]  

(5.1)

Thus, we store its signature \(\text{lm}(v_i u - v u_i)\) into our array \(H\). If however, \(\text{lt}(v_i u)\) equals \(\text{lt}(v u_i)\) then we do not have enough information to calculate \(\text{lm}(v_i u - v u_i)\) as we only store the leading monomials of \(u\) and \(u_i\).

Even without the calculation of the above syzygies, algorithm GVW computes a minimal Gröbner basis for the syzygy module. As J-signatures are processed in increasing order, every relevant syzygy is available in time for the processing of every J-pair except the J-pairs that generate new syzygies. It is these J-pairs, those that regular top-reduce to zero, that syzygies of the form (5.1) anticipate. No other aspect of the algorithm is affected (other than its performance in terms of runtimes and memory usages).

Table 5.3 list the number of regular top-reductions to zero as seen by GVW with and without the calculation of syzygies of the form (5.1). Finally, we mention that if the syzygy module was known ahead of time, absolutely no regular top-reductions to zero would be necessary in GVW.

### 5.3 Polynomial Subtraction

The concept of super top-reductions is one of the aspects of the above algorithms that improves performance. If a J-pair, after a sequence of regular top-reductions, is no longer regular top-reducible but is super top-reducible, then we may discard the J-pair. Thus if a pair is eventually super top-reducible, we do not need to reduce it to zero as is done in earlier algorithms (e.g. Buchberger’s algorithm). Thus the earlier we can detect eventual super top-reducibility, the better.

The goal of this section is to describe how to perform a subset of the work required in calculating the sequence of regular top-reductions. For if the pair is eventually super top-reducible, then we do not need the actual fully regular top-reduced pair, we only need to be able to detect whether it is super top-reducible. Only when the pair is not super top-reducible does the remainder
### Table 5.3: The count of reductions to zero with and without syzygies of the form (5.1)

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>21</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>0</td>
<td>25</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>0</td>
<td>75</td>
<td>75</td>
<td>86</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>29</td>
<td>53</td>
<td>55</td>
<td>44</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>8</td>
<td>20</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>36</td>
<td>-</td>
<td>-</td>
<td>128</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>52</td>
<td>51</td>
<td>51</td>
<td>35</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>50</td>
<td>80</td>
<td>80</td>
<td>47</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>156</td>
<td>143</td>
<td>143</td>
<td>88</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>298</td>
<td>270</td>
<td>270</td>
<td>151</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>167</td>
<td>137</td>
<td>137</td>
<td>139</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>107</td>
<td>154</td>
<td>154</td>
<td>118</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>52</td>
<td>289</td>
<td>289</td>
<td>49</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>183</td>
<td>-</td>
<td>-</td>
<td>179</td>
</tr>
</tbody>
</table>

of the regular top-reductions need to be completed.

#### 5.3.1 Naive Polynomial Subtraction

To begin, we describe naive polynomial reduction. Suppose that \((u, v)\) is to be regular top-reduced by some sequence \((u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\) of pairs. The signature calculation is easy as it does not change. But \(v\) is to be reduced by \(v_1\) and its result is to again be reduced by \(v_2\) as in the following.

\[
\begin{align*}
  v'_1 & = v - \frac{\text{lt}(v)}{\text{lt}(v_1)} v_1, \\
  v'_2 & = v'_1 - \frac{\text{lt}(v'_1)}{\text{lt}(v_2)} v_2, \\
  & \vdots \\
  v'_k & = v'_{k-1} - \frac{\text{lt}(v'_{k-1})}{\text{lt}(v_k)} v_k.
\end{align*}
\]

Each of the \(k\) equations above involves multiplying a polynomial by a monomial followed by the subtraction of two polynomials. To calculate the amount of work required, we will assume that each
polynomial has $O(T)$ terms and that each of the polynomials is represented as a sorted (with respect to $\prec$) descending list of terms. Then multiplying a polynomial by a monomial takes $O(nT)$ where $n$ is the number of variables in our ring $\mathbb{F}[x_1, \ldots, x_n]$. Also, since the polynomials are in decreasing sorted order, polynomial subtraction takes $O(nT)$ where we are assuming that operations in $\mathbb{F}$ take constant time while the comparison of two monomials with $\prec$ takes $O(n)$. Thus equation (5.2) takes $O(nT)$ time and results in a polynomial with $O(2nT)$ terms. Since, in the worst case, polynomial $v'_k$ has $O(kT)$ terms, all $k$ reductions take $O(nk^2T)$ time. Notice that for general polynomials (without much cancelling), one can not hope to do better than $\Omega(nkT)$.

An alternate approach is to delay the actual polynomial subtractions until they are needed. If we are to discard $v'_k$ based only on its leading monomial, then there is no reason to calculate $v'_k - \text{lm}(v'_k)$ until we are certain that $v'_k$ is to be retained. To accomplish this feat, we will employ a generalization of the Merge sort algorithm which will enable us to do the $k$ reductions in $O(nkT \log k)$ time with a best case scenario of $O(nk \log k)$ when $v'_k$ can be discarded.

### 5.3.2 Merge Sort and a Generalization

Comparison based sorting (as in our case of working with monomials) can be accomplished no faster than $O(n \log n)$ time for a list of size $n$. Merge sort is one of many $O(n \log n)$ sorting algorithms with the advantage that it is easy to understand and implement. We assume throughout this section (5.3.2 only) that we are sorting elements that can be compared in $O(1)$ time. We also assume for clarity that we are producing lists sorted in increasing order although our application will involve lists (of monomials) sorted in decreasing order so that each polynomial’s leading term will be the first element of the list.

Given a list of $n$ elements, the merge sort algorithm says to split the list arbitrarily into two sublists of roughly the same size, sort the sublists in $O((n/2) \log (n/2)) = O(n \log n)$ each and splice them back together in $O(n)$ time. Of course, merge sort can be used to sort the two sublists making this a recursive algorithm. The number of recursions required to reach a base case of a very small list is $O(\log n)$. Each level of the recursion requires $O(n)$ work to split and assemble all lists (within that depth of recursion) making merge sort an $O(n \log n)$ algorithm.

We generalize merge sort by splitting a list of length $kT$ into $k$ sublists of roughly equal size. Each sublist of length $T$ can be sorted ascendingly in $O(T \log T)$ time so that all $k$ sublists can be sorted in $O(kT \log T)$. Now we have $k$ sorted lists. We say that list $L_1$ is smaller than list $L_2$
if \( L_1 \)'s first element is smaller than \( L_2 \)'s first element. With this \( \mathcal{O}(1) \) comparison defined for all \( k \) lists, we insert them into a minimum priority queue in \( \mathcal{O}(k) \) time. Priority queues are very useful data structures and are often implemented with min heaps\(^1\). For details on such data structures, see [7, 11], for example.

With the typical notion of a priority queue, the dequeue operation would return the smallest of the \( k \) lists in \( \mathcal{O}(\log k) \) time. We will modify that to give us (and remove) the smallest element of the smallest list in \( \mathcal{O}(1) \) time, but after such a dequeue, what was the smallest list may no longer be the smallest within the queue, so we must bubble it downward within the queue for a total of \( \mathcal{O}(\log k) \) time. If the list becomes empty, it can be removed from the queue at which point the “bottom right” element can be substituted in its place and bubbled down, again for \( \mathcal{O}(\log k) \) time. Since there are \( kT \) total elements within all the lists contained in the priority queue, it will take \( kT \) dequeue operations to sort all \( kT \) elements, thus taking \( \mathcal{O}(kT \log k) \) time. Therefore this generalized version of merge sort runs in \( \mathcal{O}(kT \log T + kT \log k) = \mathcal{O}(kT \log(kT)) \) time, which is optimal.

### 5.3.3 Applying Generalized Merge Sort to Polynomial Subtraction

We are now in a position to use our generalization of merge sort in order to delay the polynomial calculations. We will view polynomials as decreasing lists of terms with respect to our term order \( \prec \), and we will maintain a maximum priority queue of the \( k \) polynomials \( v_1, \ldots, v_k \) used in the reduction of \( v \) as in (5.2). The useful property of such a setup is that the leading term of \( v_k' \) will be available as the first element of the priority queue’s first list (polynomial). For convenience, we will refer to the leading monomial of the priority queue to mean the leading monomial of the queue’s largest polynomial.

We will allow multiple polynomials within the priority queue to contain the same monomials with the exception of the leading monomial of the queue. This ensures that the queue’s leading monomial is well defined, that is to make sure that the queue’s leading monomial should not be cancelled with any of its other polynomial’s leading monomials. This is actually an easy condition to maintain as we only need to compare the max-heap’s root node’s leading monomial with its immediate children.

Given the polynomial max-priority queue setup described above (representing the polyno-

\(^1\)A minimum priority queue uses a min-heap, while a maximum priority queue uses a max-heap. Our application in the next section will require the latter.
mial \( v \), if we want to reduce \( v \) by some \( v_i \) where \( \text{lm}(v_i) | \text{lm}(v) \), then we simply insert the polynomial 

\[-\text{lt}(v)/\text{lt}(v_i)v_i\]

into the queue. The only trouble is that there will then be two polynomials within the queue having the largest leading monomials one of which being the largest polynomial and the other being the polynomial we just inserted. One solution is to simply delete the largest polynomial’s leading monomial and bubble it down, followed by inserting the polynomial 

\[-\text{lm}(v)/\text{lm}(v_i)(v_i - \text{lm}(v_i))\]

That would perform the cancellation of leading terms that was the point of the polynomial subtraction. But there is still the possibility that after the bubbling down of the largest polynomial and the bubbling up of the newly inserted polynomial that the new largest polynomial’s leading monomial is not unique within the queue.

Whenever the largest polynomial’s leading monomial is not unique within the queue, it must be the case that one of the max-heap’s root node’s immediate children has the same leading monomial. Simply add the root children’s leading coefficient to the root node, and delete the child’s leading monomial. Once the child’s leading monomial is deleted, its new leading monomial will be smaller, so we must bubble it downward into the heap. If the queue’s leading term is now zero, we delete the root node’s leading monomial and bubble it downward. We must then recheck to make sure that the queue’s leading monomial is unique. Each time we make the queue’s leading monomial unique, it is the same amount of work as dequeuing an element. Combining coefficients is assumed to be \( \mathcal{O}(1) \), removing a monomial from a polynomial (using a linked list [7] representation for example) is \( \mathcal{O}(1) \), and bubbling down an element is \( \mathcal{O}(\log k) \). In fact, for the remainder of the analysis, we will consider this process as a dequeue operation.

Since we are inserting \( \mathcal{O}(k) \) polynomials (with comparisons costing \( \mathcal{O}(n) \)) into this priority queue, each insert (again, considering the checking and fixing of uniqueness of the queue’s leading monomial as dequeue operations) takes \( \mathcal{O}(n \log k) \) time. So in the best case, the insertion of \( k \) elements into the queue can cost as little as \( \mathcal{O}(nk \log k)^2 \) which has the potential of beating the naive \( \mathcal{O}(nkT) \) approach when the queue (or the polynomial it represents) can be thrown away. When the queue can not be discarded, we need to reconstruct the polynomial it represents. A sequence of \( \mathcal{O}(kT) \) dequeue operations will produce the desired polynomial whose terms are in decreasing sorted order. Since each dequeue (counting each fixing of uniqueness of the queue’s leading monomial as an individual dequeue) takes \( \mathcal{O}(n \log k) \), the total time to reconstruct the polynomial is \( \mathcal{O}(nkT \log k) \)

\[\text{\footnotesize[7]}\]

One might recall that a priority queue of \( k \) elements can be constructed in \( \mathcal{O}(k) \) time, but for our purpose, we need the queue’s leading monomial of each intermediate enqueue operation to determine which polynomial to reduce by next.
Timings in seconds

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
<td>0.32</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>4.90</td>
<td>4.72</td>
<td>4.87</td>
<td>3.51</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>2.14</td>
<td>3.22</td>
<td>3.26</td>
<td>3.33</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>0.10</td>
<td>1.50</td>
<td>0.95</td>
<td>0.11</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>0.28</td>
<td>3.46</td>
<td>3.43</td>
<td>0.13</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>160.14</td>
<td>-</td>
<td>-</td>
<td>51.59</td>
</tr>
</tbody>
</table>

Maximal memory usage in megabytes

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>4.73</td>
<td>4.72</td>
<td>4.72</td>
<td>4.45</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>5.34</td>
<td>5.62</td>
<td>5.61</td>
<td>5.05</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>10.98</td>
<td>10.19</td>
<td>10.17</td>
<td>8.30</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>33.23</td>
<td>32.98</td>
<td>33.08</td>
<td>21.25</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>53.97</td>
<td>15.61</td>
<td>15.59</td>
<td>15.64</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>6.38</td>
<td>8.81</td>
<td>7.98</td>
<td>5.75</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>5.62</td>
<td>20.11</td>
<td>19.89</td>
<td>6.92</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>39.80</td>
<td>-</td>
<td>-</td>
<td>38.56</td>
</tr>
</tbody>
</table>

Table 5.4: Runtimes and memory usages for naive polynomial subtraction as in equation (5.2)

which beats $O(nk^2T)$.

One might notice that this process stores many monomials multiple times. In fact, a single monomial can be present in each of the $k$ polynomials until either the queue is discarded or its polynomial is reconstructed. When dealing with dense polynomials, one might wonder if the memory usage is worth the speedup. The answer turns out to be simple: only one of these queues are maintained at a time. The vast majority of the memory used by the GVW algorithm is in the storage of all our reduced J-pairs. Tables 5.4 and 5.5 compare runtimes and memory usages for GVW with naive and queue-based polynomial subtraction respectively. The queue based approach results in nearly across the board speed improvements for a small memory penalty.

5.4 GVW with Binary Field Equations

One of the most important scenarios to be able to calculate Gröbner bases is in $R = \mathbb{F}_2[x_1, \ldots, x_n]$ with the presence of the field equations $x_i^2 + x_i = 0$, $1 \leq i \leq n$. Binary equations with binary solutions appear in many places. One application of Gröbner bases described in section 1.2 involved breaking the cryptographic standard AES. Similar equations will come from any modern
Table 5.5: Runtimes and memory usages for queue based polynomial subtraction

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>0.01</td>
<td>0.0</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>0.02</td>
<td>0.05</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>0.34</td>
<td>0.37</td>
<td>0.37</td>
<td>0.36</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>3.25</td>
<td>2.94</td>
<td>3.00</td>
<td>3.20</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>1.76</td>
<td>3.65</td>
<td>3.67</td>
<td>3.87</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>0.08</td>
<td>0.45</td>
<td>0.36</td>
<td>0.09</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>0.34</td>
<td>3.26</td>
<td>3.21</td>
<td>0.15</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>142.20</td>
<td>-</td>
<td>-</td>
<td>37.12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case (# gen)</th>
<th>POT/G2V</th>
<th>TOP</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katsura5 (22)</td>
<td>4.89</td>
<td>4.86</td>
<td>4.88</td>
<td>4.53</td>
</tr>
<tr>
<td>Katsura6 (41)</td>
<td>5.50</td>
<td>6.33</td>
<td>6.33</td>
<td>5.33</td>
</tr>
<tr>
<td>Katsura7 (74)</td>
<td>14.86</td>
<td>13.53</td>
<td>13.53</td>
<td>9.06</td>
</tr>
<tr>
<td>Katsura8 (143)</td>
<td>52.69</td>
<td>45.69</td>
<td>45.81</td>
<td>25.69</td>
</tr>
<tr>
<td>Schrans-Troost (128)</td>
<td>55.69</td>
<td>18.91</td>
<td>18.91</td>
<td>18.36</td>
</tr>
<tr>
<td>F633 (76)</td>
<td>6.70</td>
<td>10.38</td>
<td>8.62</td>
<td>6.06</td>
</tr>
<tr>
<td>Cyclic 6 (99)</td>
<td>5.67</td>
<td>20.91</td>
<td>20.81</td>
<td>7.00</td>
</tr>
<tr>
<td>Cyclic 7 (443)</td>
<td>40.08</td>
<td>-</td>
<td>-</td>
<td>45.02</td>
</tr>
</tbody>
</table>

Thus, it is of particular interest to be able to calculate Gröbner basis for binary systems with the field equations.

Given how much trust we put in these cryptographic schemes, one would accurately suspect that calculating a Gröbner basis for such systems is computationally difficult. The example provided in section 1.3 will attest to its complexity, but in general, calculating a Gröbner basis in this setting is \( \mathcal{NP} \)-hard.

**SAT** Given a system of quadratic polynomials \( a_1, \ldots, a_m \in \mathbb{F}_2[x_1, \ldots, x_n] \), deciding if they have any common solutions in \( \mathbb{F}_2^n \) is \( \mathcal{NP} \)-complete.

**Proof.** We proceed by finding a polynomial time reduction of an instance of SAT to a system of polynomial equations. Given an instance of SAT,

\[
s = a_1 \land a_2 \land a_3 \land \cdots \land a_k = \bigwedge_{i=1}^{k} (t_{i,1} \lor \cdots \lor t_{i,m_i}) ,
\]

where each \( t_{ij} \in \{b_1, \ldots, b_N, \neg b_1, \ldots, \neg b_N\} = \{b_1, \ldots, b_N, 1 - b_1, \ldots, 1 - b_N\} \), each clause can be converted into a system of binary quadratic polynomials by the following algorithm. Given a generic
clause \( a_i = t_{i,1} \lor t_{i,2} \lor \cdots \lor t_{i,m_i} \), we can define our \( i^{th} \) system of equations by

\[
\begin{align*}
  x_{i,1} &= t_{i,1} \\
  x_{i,2} &= t_{i,2} + x_{i,1} + t_{i,2} \cdot x_{i,1} \quad (= t_{i,2} \lor x_{i,1}) \\
  & \vdots \quad \vdots \\
  x_{i,m_i} &= t_{i,m_i} + x_{i,m_i-1} + t_{i,m_i} \cdot x_{i,m_i-1} \quad (= t_{i,m_i} \lor x_{i,m_i-1})
\end{align*}
\]

Here, each \( t_{i,j} \) is interpreted as a 0 or 1 in \( \mathbb{F}_2 \). Any assignment of the \{\( b_i \}\) force the values of \{\( t_{i,j}, x_{i,j} \}\). Thus, this system is exactly the same as interpreting \( a_i = t_{i,1} \lor t_{i,2} \lor \cdots \lor t_{i,m_i} \) as \( x_{i,m_i} = t_{i,1} \lor (t_{i,2} \lor \{t_{i,4} \lor \cdots \lor t_{i,m_i}\}) \). The cost of constructing this system is exactly \( \mathcal{O}(m_j) \), and each equation is a quadratic equation in \( \mathbb{F}_2[x_{i,1}, \ldots, x_{i,m_i}, b_1, \ldots, b_N] \). The cost of constructing all \( k \) systems is \( \mathcal{O}(m_1 + m_2 + \cdots + m_k) \).

Next, given clauses \( a_1, \ldots, a_k \) and their associated systems of equations, we can couple them together by making a \((k+1)^{th}\) system of equations as the following:

\[
\begin{align*}
  y_1 &= x_{1,m_1} \\
  y_2 &= x_{2,m_2} \cdot y_1 \quad (= x_{2,m_2} \land y_1) \\
  & \vdots \quad \vdots \\
  y_k &= x_{k,m_k} \cdot y_{k-1} \quad (= x_{k,m_k} \land y_{k-1}) \\
  0 &= y_k - 1
\end{align*}
\]

Thus, the above system is solvable if and only if \( a_1 \land \cdots \land a_k = 1 \) for some assignment of \{\( b_i \}\}. Each of these equations are quadratic, and the system took exactly \( \mathcal{O}(k) \) steps to construct. All \( k+1 \) systems of equations form one big quadratic system in \( \mathbb{F}[y_1, \ldots, y_k, b_1, \ldots, b_N, x_{i,j}, 1 \leq i \leq k, 1 \leq j \leq m_i] \).

Finally, an instance of SAT can be reduced to an a polynomial system of size no more than a polynomial (actually, linear) multiple of the SAT instance size in polynomial (actually, linear) time, \( \mathcal{O}(m_1 + \cdots + m_k)^3 \).

As the calculation of a Gröbner basis is one way to answer SAT, we now know that calculating a Gröbner basis in this setting is \( \mathcal{NP} \)-hard.

\[ \mathcal{O}(m_1 + \cdots + m_k) = \mathcal{O}((m_1 + 1) + \cdots + (m_k + 1)) = \mathcal{O}(k + m_1 + \cdots + m_k) \]
For the purposes of this section, suppose are given \( g_1, \ldots, g_m \in \mathbb{F}_2[x_1, \ldots, x_n] \). We can calculate a Gröbner basis directly for the ideal \( \langle g_1, \ldots, g_m, x_1^2 - x_1, \ldots, x_n^2 - x_n \rangle \) in \( R \). But computationally, it is more efficient to calculate a Gröbner basis for \( \langle g_1, \ldots, g_m \rangle \) in the quotient ring \( R/J \) where \( J = \langle x_i^2 - x_i, \ 1 \leq i \leq n \rangle \). Thus, we proceed by modifying figure 3.2.

5.4.1 Representing Monomials

For general polynomials (specifically, the \( v \) part of \((u, v)\)) in this setting, there will never be an exponent larger than one. Therefore, it will be convenient and efficient for us to use a single bit per variable to represent monomials. For example, the monomial \( x_5x_3 \in \mathbb{F}_2[x_0, \ldots, x_7] \) can be represented as an (unsigned) 8-bit integer.

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7
\end{array}
\]

This representation is particularly convenient in 32-bit (analogously, 64-bit) computers because operations on 32 variables can be done just as quickly as with 1 variable. For example, still in the ring \( \mathbb{F}_2[x_0, \ldots, x_7] \) with lexicographic order \( \succ \) with \( x_0 \succ x_1 \succ \ldots \succ x_7 \) and the representation above we know that \( x_5x_3 \succ x_4x_5x_6x_7 \). To determine this order from the binary representation, a simple (unsigned) integer comparison is required.

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7
\end{array} \quad \succ \quad \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7
\end{array}
\]

Thus, in the analysis above when we assumed that monomial comparison takes \( \mathcal{O}(n) \) time, the hidden constant is quite small in this situation. Reverse lexicographic orderings are accomplished by reversing the order of the bits throughout. Graded orderings require only a check of the degree before using integer comparison. Of course, the degree of a monomial can be maintained in an auxiliary variable, only to be updated whenever the monomial is changed.

There are several methods one can use to calculate the number of ones (a monomial’s degree) in a binary variable. The simplest method is to loop through all the bits and sum number of nonzero bits. See [25] for several approaches.

The most interesting approach gives us a runtime logarithmic in the number of bits (of a
The key observation is to notice that adding two one bit integers requires no more than two bits to represent the result. Adding two two-bit integers requires no more than four bits to represent the result. In general, adding two $n$-bit integers requires at most $2^n$ bits of storage. Thus, “in parallel” we consider each bit as a 1-bit integer. We sum the even positions with the odd positions and deposit the results into pairs of bits (going from the first row to the second row of figure 5.1). We then think of those pairs of bits as 2-bit integers and we then sum consecutive pairs of those 2-bit integers and deposit the results into the 4-bit blocks shown in the third row. We continue in this fashion for $\log_2 n$ iterations for an $n$-bit input. See figure 5.1 and code listing 5.2.

Finally, the GNU C/C++ compiler exposes a builtin command, that when its corresponding assembly instruction is available, it is compiled into a single machine instruction for $O(1)$. See code listing 5.3 or [15] for a faster approach. Also, if such a command (or machine instruction) is not available, one always has the option of using in-line assembly (when the machine instruction is available, but the C function is not) or a lookup table. Such a lookup table will cost a linear, in the number of variables, number of memory accesses, but the multiplicative constant will be quite small.

```
int count_ones(unsigned int bits) {
    return builtin_popcount(bits);
}
```

Listing 5.3: Calculating one-bits using the builtin command
The multiplication of two monomials is accomplished with the bitwise OR. For example, two monomials $x_3x_5$ and $x_4x_5x_6$ are multiplied together to get $x_3x_4x_5x_6$. Each exponent, when viewed as a bit, is obtained by the inclusive-OR of the corresponding input monomials’ exponents.

<table>
<thead>
<tr>
<th>$x_3x_5$</th>
<th>$x_4x_5x_6$</th>
<th>$x_3x_4x_5x_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 1 0 0</td>
<td>0 0 0 0 1 1 1 0</td>
<td>0 0 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

Monomial division follows roughly along the same lines. If we already know that monomial $m_1$ divides monomial $m_2$, then $m_2/m_1 = m_2 \land \neg m_1$ where the left hand side is written in the standard monomial notation while the right hand side is to be interpreted as logical bit-wise operations on bit strings. For example, if $m_1 = x_3x_5$ and $m_2 = x_3x_4x_5x_6$, then the following table illustrates the approach.

<table>
<thead>
<tr>
<th>$x_3x_5$</th>
<th>$\neg x_3x_5$</th>
<th>$x_3x_4x_5x_6$</th>
<th>$x_3x_4x_5x_6/x_3x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 1 0 0</td>
<td>1 1 1 0 1 0 1 1</td>
<td>0 0 0 1 1 1 1 0</td>
<td>0 0 0 0 1 0 1 0</td>
</tr>
</tbody>
</table>

Determining whether one monomial divides another monomial is a little different. A monomial $m_1$ divides $m_2$ if and only if $m_1 \land m_2 = m_1$. The following truth tables each illustrate the four possible combinations of exponents for variables $x$ and $y$.

<table>
<thead>
<tr>
<th>$x^0$</th>
<th>$x^1$</th>
<th>$y^0$</th>
<th>$x \mid y$</th>
<th>$x \uparrow y$</th>
<th>$y^1$</th>
<th>$x \mid y$</th>
<th>$x \mid y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$’s exponent:</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y$’s exponent:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x \land y$:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x \land y == x$:</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result:</td>
<td>$x \mid y$</td>
<td>$x \mid y$</td>
<td>$x \uparrow y$</td>
<td>$x \mid y$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thus, checking divisibility is equivalent to checking if $\neg((x \land y) - x)$ is nonzero.

A few other operations are useful. Two monomials are equal if and only if their bit-strings are equal. Calculating greatest common divisors (gcd) and least common multiples (lcm) is as simple
as applying bitwise AND and bitwise inclusive OR (respectively) to the two input bit strings.

### 5.4.2 Representing Polynomials

Everything we have seen so far with the binary field equations seems very convenient. Bit operations are simple to describe and computers execute them very quickly. Even more attractive is the fact that we can handle 32 variables in only one clock cycle. But that convenience has its limits, as is illustrated in the following example. Suppose that \( x \succ y \) and that we intend to multiply both sides by \( x \). A term order requires that the order be preserved under multiplication so that \( x^2 \succ xy \). But we skip this step. Because of our representation of monomials, when we multiply through by \( x \), we see \( x \succ xy \). But this last statement is certainly in error.

The problem lies not with our term order, but rather that the left hand side of \( x^2 \succ xy \) is automatically reduced by \( x^2 = x \). Thus expecting the preservation of \( x \succ y \) after multiplication by \( x \) is something we should not do. In particular, suppose we have a polynomial whose terms are in decreasing sorted order. If we are to multiply the polynomial by some monomial, then the result is not automatically going to be in sorted order. So if there are \( T \) terms in the polynomial in \( n \) variables, multiplying through by the monomial is going to take \( \mathcal{O}(nT) \), but resorting the terms in decreasing sorted order is going to take \( \mathcal{O}(nT \log T) \) for general comparison sorting.

One possible way to speed this up is to avoid doing a total resorting of the terms. Suppose we have a polynomial \( p(x) \) with \( T \) terms in decreasing sorted order which is to be multiplied by some monomial \( x_1x_3x_5 \), for example. Calculating \( x_1x_3x_5p(x) \) is the same as calculating \( x_5(x_3(x_1p(x))) \). If we restrict our attention momentarily to the product \( x_1p(x) \), we will see that the resulting list of terms will be in nearly sorted order. So before calculating the product, we split \( p(x) \) into two lists: one whose terms are relatively prime to \( x_1 \) and the other whose terms are divisible by \( x_1 \). While the latter list does not need to be multiplied by \( x_1 \), the former does. What is nice here is that upon multiplying the former list by \( x_1 \), the terms remain in sorted order. Thus we only need to splice the two sorted lists together in \( \mathcal{O}(nT) \) time. Repeat this process for \( x_3 \) then \( x_5 \). In general, for any monomial \( x^\alpha \) having degree \( k \) to be multiplied by a polynomial with \( T \) terms, the multiplication will take \( \mathcal{O}(nkT) \) time by this method.

In general, this method does not save us a lot of computations. If we are multiplying by monomials of degree \( \mathcal{O}(k) \) where \( k \) is the number of variables, then \( k \) is going to be about the same as \( \log T \). In fact, one would expect general polynomials to have roughly \( 2^k \) terms, and so \( T \in \mathcal{O}(2^k) \).
or \( \log T \in \mathcal{O}(k) \). These similarities in asymptotic bounds manifest very nicely in empirical runtimes. The runtimes are strikingly similar with the simpler multiply-and-sort method taking slightly less time and memory\(^4\).

The algorithm for monomial polynomial multiplication might be useful for special cases. For example, if one were trying to calculate a partial Gröbner basis where leading terms were not allowed to get very big, then this method might be useful. But for the general case, simple multiplication followed by sorting can not be improved upon.

**Theorem 5.2.** *Multiplication of a polynomial of \( T \) terms by a general monomial in \( \mathbb{F}_2[x_1, \ldots, x_n] \) with automatic reduction by field equations producing a sorted sequence of terms as output can not be accomplished faster than the \( \Omega(T \log T) \) comparisons sorting bound.*

*Proof.* Let \( b_1, \ldots, b_T \in \mathbb{N} \) be a sequence of integers to be sorted, and let \( x^{\alpha_1}, \ldots, x^{\alpha_T} \) be \( T \) random monomials in \( \mathbb{F}_2[x_1, \ldots, x_n] \) in decreasing sorted order according to \( \prec_x \). Let \( m \) be such that \( 0 \leq b_i < 2^m, 1 \leq i \leq T \), i.e. that each of our integers to be sorted requires at most \( m \) bits to represent.

Let \( \beta_i \in \{0,1\}^m, 1 \leq i \leq T \) be the binary representation of each \( b_i \). We will represent each integer \( b_i \) as a monomial \( z^{\beta_i} \) so that comparison of integers \( b_i > b_j \) is equivalent to lexicographic monomial comparison \( x^{\beta_i} \succ_{\text{lex}} x^{\beta_j} \). We will use \( \succ \) to represent the elimination order such that \( x^{\alpha_i} z^{\beta_j} \succ x^{\alpha_k} z^{\beta_l} \) whenever \( x^{\alpha_i} \succ_x x^{\alpha_k} \) or \( x^{\alpha_i} = x^{\alpha_k} \) and \( z^{\beta_j} \succ_{\text{lex}} z^{\beta_l} \).

If \( p(x) = \sum_{i=1}^T x^{\alpha_i} \) is our polynomial written in \( \succ \) sorted decreasing order in \( \mathbb{F}_2[x_1, \ldots, x_n] \), then \( \tilde{p}(x,z) = \sum_{i=1}^T x^{\alpha_i} z^{\beta_i} \) is a polynomial in \( \mathbb{F}_2[x_1, \ldots, x_n, z_1, \ldots, z_m] \) also in \( \succ \) decreasing sorted order. We will multiply \( x_1 x_2 \cdots x_n \) by \( \tilde{p} \) to get \( (x_1 x_2 \cdots x_n) \sum_{i=1}^T z^{\beta_i} \), which is only in \( \succ_{\text{lex}} \) decreasing sorted order if the \( b_i \) are. Any algorithm which performs this multiplication to produce output in \( \succ \) decreasing sorted order can be used to sort monomials into \( \succ_{\text{lex}} \) decreasing sorted order or equivalently integers into \( \succ \) decreasing sorted order (using comparisons only), and must therefore perform at least \( \Omega(T \log T) \) monomial comparisons. \( \square \)

### 5.4.3 Representing Signatures

Per our earlier notations and definitions, a signature is of the form \( x^{\alpha} E_i \), \( 1 \leq i \leq m \) where \( x^{\alpha} \) is a monomial in \( \mathbb{R} = \mathbb{F}_2[x_1, \ldots, x_n] \) and \( E_i \) is the \( i^{th} \) canonical unit vector. Thus a signature is simply a monomial in \( \mathbb{R}^m \). Thus one might conclude that a signature can be represented with

---

\(^4\)In the multiply-and-sort version, a profiler found the program spending 24% (Cyclic10, g2, no optimizations flags given to g++) of the runtime sorting terms after these multiplications.
a single integer $1 \leq i \leq m$ in addition to the monomial representation used above. Unfortunately, there are a couple subtleties that must be accounted for.

First, we should notice that no $(u, v)$-pair will ever be stored unless its signature is square free. This is because the syzygy basis will always include $x_i^2 E_j$, $1 \leq i \leq n, 1 \leq j \leq m$. Thus the problem is more about the temporary monomials that result from multiplications that are then used for comparisons. We have identified various instances of when this is an issue, but only two of them are active at any time. An example of the first is in finding a reductor. Given a pair $(u, v)$, we look for a stored pair $(u_k, v_k)$ such that

$$\text{lcm}(v_k) \mid \text{lcm}(v) \quad \text{and} \quad \frac{\text{lcm}(v)}{\text{lcm}(v_k)} u_k \prec u.$$  

We may assume that both $u$ and $u_k$ are square-free, for if they were not, both would have been discarded. But if the greatest common divisor of $\text{lcm}(v)/\text{lcm}(v_k)$ and the monomial part of $u_k$ is non-constant, then the product will not be square-free. If the resulting product is automatically reduced by the field equations, then the monomial comparison may give the wrong answer allowing us to perform an invalid top-reduction. Thus, we need to be able to store exponents as large as two.

Another instance of the above problem is with the $g^2$ monomial comparison. Recall that if we are calculating a Gröbner basis for $\langle g_1, \ldots, g_m \rangle$ with respect to $\succ$ and extended to $R^m$ as $\succ_{g^2}$, then we say that $x^\alpha E_i \succ_{g^2} x^\beta E_j$ when $x^\alpha g_i \succ x^\beta g_j$ or $x^\alpha g_i = x^\beta g_j$ and $x^\alpha E_i \succ_{pot} x^\beta E_j$. While it must be the case that $x^\alpha$ and $x^\beta$ are cube-free (as described above) and $g_i$ and $g_j$ are square-free, the products may have exponents as large as 3. As above, the higher exponents must be retained for the sake of the comparison. Thus when using the $g^2$ (or any similar) ordering on $R^m$, exponents as large as cubes must be handled for temporary calculations.

But all is not lost. Instead of storing exponents as a single bit, we store them as pairs of bits. Thus we have adequate storage for any exponents whose binary representation is in the set \{0,1\}^2. Now multiplications become additions (where in section 5.4.1 they were bitwise inclusive OR operations). Monomial comparisons can still use the same trick as above as long as the variables are ordered appropriately. Counting degrees is nearly as simple as above, but without the automatic reduction by field equations, one may now simply add degrees with every multiplication. Least common multiples are not needed for signatures, and neither is division. Greatest common divisors and is-divisible-by are only needed between two multilinear signatures, thus our same bitwise operations
works as expected.

The only operations that remains to be defined is the process of multiplying a binary (1-bit) monomial (as described in 5.4.1) with our new (2-bit) signature representation. It is enough to simply explain how to convert a binary monomial into our new 2-bit representation. As an example, we want to convert the binary string $11001001^5$ into $010100001000001$, i.e. we want to interleave $11001001$ with a string of all zeros. One method is to use lookup tables, again for a linear runtime with small constant. Unfortunately, we were unable to find a constant-time operation like the bit counting methods above. We do, however, have a logarithmic time algorithm that is almost the reverse of the trick used in listing 5.2. Such an algorithm is implemented easily enough in C/C++. See code listing 5.4.

### 5.4.4 Calculating J-pairs and Syzygies

The field equations have a very simple form which allows us to rule out the necessity of several calculations. For example, we obtain a new, fully regular top-reduced, $(u, v)$-pair which is not super top-reducible. We store the pair and proceed to calculate new J-pairs. Just as in the quotient ring version of GVW, we begin by calculating J-pairs between $(u, v)$ and $(0, x_i^2 + x_i)$, $1 \leq i \leq n$. Such a J-pair will necessarily be of the form

$$t_i(u, v) \quad \text{where} \quad t_i = \frac{\text{lcm} \left( \text{lm}(u), x_i^2 \right)}{\text{lm}(v)}.$$

If $x_i \nmid \text{lm}(v)$ then $x_i^2 \mid t_i$ and the resulting J-signature will be divisible by an element of $H$, our stored list of syzygies. Therefore, we only calculate the J-pair between $(u, v)$ and $(0, x_i^2 + x_i)$ when the greatest common divisor of $\text{lm}(v)$ and $x_i$ is non-constant. Further, in the process of calculating the J-pair, if the greatest common divisor of $t_i$ and the monomial part $x^\alpha$ of the signature $u = x^\alpha E_j$ is non-constant, we may throw away the pair as its signature is divisible the same element in $H$. In fact, this last observation holds for calculating J-pairs in general.

Suppose we are to calculate the J-pair of $(u_i, v_i)$ and $(u_j, v_j)$. Intermediate calculations yield the following.

$$t_i = \frac{\text{lcm} \left( \text{lm}(v_i), \text{lm}(v_j) \right)}{\text{lm}(v_i)} \quad \text{and} \quad t_j = \frac{\text{lcm} \left( \text{lm}(v_j), \text{lm}(v_i) \right)}{\text{lm}(v_j)}$$

\(^5\)See http://en.wikipedia.org/wiki/11001001 for this significance of this binary number. It is my “go to” binary number for examples.
monomial4(const monomial2& mon2) { // sizeof(int)=4
    const int array2Length = (n + 31) / 32; // round up
    const unsigned int L16 = 0xFFFF0000, R16 = 0x0000FFFF;
    const unsigned int L8 = 0xFF00FF00, R8 = 0x00FF00FF;
    const unsigned int L4 = 0xF0F0F0F0, R4 = 0x0F0F0F0F;
    const unsigned int L2 = 0xCCCCCCCC, R2 = 0x33333333;
    const unsigned int L1 = 0xAAAAAAAA, R1 = 0x55555555;

    const unsigned int * expons2 = mon2.getExponents();
    int i, k = 0;
    for (i = 0; i < array2Length; i++) {
        unsigned int left = (expons2[i] & L16) >> 16,
            right = expons2[i] & R16;
        // rewrite this using a macro
        left = (left & R8) | ((left & L8) << 8);
        right = (right & R8) | ((right & L8) << 8);
        left = (left & R4) | ((left & L4) << 4);
        right = (right & R4) | ((right & L4) << 4);
        left = (left & R2) | ((left & L2) << 2);
        right = (right & R2) | ((right & L2) << 2);
        left = (left & R1) | ((left & L1) << 1);
        right = (right & R1) | ((right & L1) << 1);

        exponents[k++] = right;
        if (k < arrayLength) exponents[k++] = left;
    }
}

Listing 5.4: The conversion of multilinear monomials to multicubic monomials
inline bool square_free() const {
    int answer = 0;
    for (int i = 0; i < arrayLength; i++)
        answer |= exponents[i] & 0xAAAAAAAA;
    return answer;
}

Listing 5.5: Determining whether a multicubic monomial is square-free

We assume that $t_i u_i \geq t_j u_j$ (another place where exponents become larger than one as discussed in section 5.4.3). If the greatest common divisor of $t_i$ and the monomial part of $u_i$ is non-constant, we may discard the pair. Alternatively (to calculating greatest common divisors, in which case the gcd operation will not be needed on signature monomials), we may quickly check to see if the resulting signature monomial is square-free. To do this, one needs only restrict their attention to the more significant bit of the exponent string as in code listing 5.5.

With the production of every new $(u_{k+1}, v_{k+1})$, we have several new syzygies to consider. If $(u_1, v_1), \ldots, (u_k, v_k)$ are the previously computed, reduced, and stored J-pairs, then we observe that $v_{k+1}(u_i, v_i) - v_i(u_{k+1}, v_{k+1}), 1 \leq i \leq k$ are new syzygies. We need only determine which of $\text{lm}(v_{k+1})u_i$ and $\text{lm}(v_i)u_{k+1}$ are larger and store it into $H$. Of course, if they are equal and $\text{lc}(v_i) = \text{lc}(v_{k+1})$, we can not say anything unless we store lower order terms of the $u$. This is the only price we pay for keeping only the signature part of $u$. But in our current scenario, we can say more. Suppose that the larger of $\text{lm}(v_i)u_i$ and $\text{lm}(v_j)u_i$ is $\text{lm}(v_i)u_j$, then $\text{lm}(v_i)u_j$ is to be inserted into $H$. But if $\text{lm}(v_i)$ and the monomial part of $u_j$ are not relatively prime, then their presence in $H$ will be redundant. Thus we needn’t store the syzygy. Finally, we needn’t ever calculate syzygies with the field equations because they would necessarily be of the form $x_j^2(u_i, v_i)$, whose signature is reducible by a field equation syzygy.
Appendices
Appendix A  C++ Code for GVW

The following is C++ code for figure 4.1 presented in a top-down fashion. The code is very templated so that many factors, including number of variables and term orderings, are compile time decisions. While templated code is nice to read, the compiler errors are quite difficult to understand. If we had to rewrite this, we would use more global constants and less templates.

A.1 The Makefile

exList = katsura5 katsura6 katsura7 katsura8 schrans–troost \ cyclic6 cyclic7 cyclic8 cyclic9 cyclic10
shortList = katsura5 katsura6 katsura7 katsura8 schrans–troost \ cyclic6 cyclic7 cyclic8
longList = cyclic7 cyclic8 #cyclic9 cyclic10
modeList = POT TOP g1 g2

optflags = -O3 -funroll-loops

fast:
    @for i in $(shortList) :
    do \ 
      echo "Compiling: $$i$$..." ;
      g++ -o "run$i"-POT -DBRIEF $(optflags) "$i-POT.cpp" &
      g++ -o "run$i"-TOP -DBRIEF $(optflags) "$i-TOP.cpp" &
      wait ;
    done

sturmfels: sturmfels.cpp gvw.h
    g++ -o sturmfels $(optflags) sturmfels.cpp

bigFast:
    @for i in $(longList) :
    do \ 
      echo "Compiling: $$i$$..." ;
      g++ -o "run$i"-POT -DBRIEF $(optflags) "$i-POT.cpp" &
      g++ -o "run$i"-g2 -DBRIEF $(optflags) "$i-g2.cpp" &
      wait ;
    done

clean:
    @for i in $(exList) :
    do \ 
      echo "Deleting $$i$$..." ;

rm "run\{i\}-POT";
rm "run\{i\}-TOP";
rm "run\{i\}-g1";
rm "run\{i\}-g2";
done
rm * . out

all:  
fast
@echo "Big List" > output.txt

@for i in $(shortList) ;
do 
  echo "Running: $$ \{i\} \ldots"
  echo "-----------------------------$$ \{i\}"
  /usr/bin/time -o output.txt -a --format=",%U,%M,%S) ". /run\{i\}-POT" >> output.txt ;
  /usr/bin/time -o output.txt -a --format=",%U,%M,%S) ". /run\{i\}-TOP" >> output.txt ;
  /usr/bin/time -o output.txt -a --format=",%U,%M,%S) ". /run\{i\}-g1" >> output.txt ;
  /usr/bin/time -o output.txt -a --format=",%U,%M,%S) ". /run\{i\}-g2" >> output.txt ;
done
continue: #bigFast
#  @echo "Running: 2 x cyclic7 \ldots"
#  @echo "-----------------------------cyclic7"
#  @/usr/bin/time -o output.txt -a --format=",%U,%M,%S) . /runcyclic7-POT >> output.txt
#  @/usr/bin/time -o output.txt -a --format=",%U,%M,%S) . /runcyclic7-POPT >> output.txt
#  @/usr/bin/time -o output.txt -a --format=",%U,%M,%S) . /runcyclic7-g1" >> output.txt
#  @/usr/bin/time -o output.txt -a --format=",%U,%M,%S) . /runcyclic8-g2" >> output.txt

#  @echo "Running: 1 x cyclic9 \ldots"
#  @echo "-----------------------------cyclic9"
#  @/usr/bin/time -o output.txt -a --format=",%U,%M,%S) . /runcyclic9-POT >> output.txt
#  @/usr/bin/time -o output.txt -a --format=",%U,%M,%S) . /runcyclic9-g2" >> output.txt

#  @echo "Running: 1 x cyclic10 \ldots"
#  @echo "-----------------------------cyclic10"
#  @/usr/bin/time -o output.txt -a --format=",%U,%M,%S) . /runcyclic10-g2" >> output.txt

cyclic8POT:
@echo "Running: cyclic8 POT..."
@/usr/bin/time -o output.txt -a --format=",%U,%M,%S)"
./runcyclic8-POT >> output.txt

tex:
g++ -o txt2tex tally/tally.cpp

rounds: all #continue
@echo"

dpdf:
@./txt2tex output.txt output.tex
@pdflatex output.tex 2> /dev/null > /dev/null
@evince output.pdf &

prof:
g++ -pg sturmfels.cpp
time --portability ./a.out
@echo ""
gprof -b > prof.txt
kprof -f prof.txt -p gprof 2> /dev/null &
@echo
@echo
Listing 6: A sample makefile showing the compiler flags used for the code to follow

A.2 Example Main File: Katsura5, GRevLex, TOP

#include <fstream>
#include <vector>
#include <iostream>
#include "modp.h"
#include "monomial.h"
#include "polynomial.h"
#include "module-monomial.h"
#include "module-order.h"
#include "j_pair.h"
#include "pqueue.h"
#include "singular.h"
#include "gvw.h"

using namespace std;

// characteristic : 7583
// number of vars : 7
// block 1 : ordering dp
// : names x y z t u v h
// block 2 : ordering C
// i[1]=2x2+2y2+2z2+2t2+2u2+v2−vh
// i[2]=xy+yz+2zt+2tu+2uv+uh
// i[3]=2xz+2yt+2zu+u2+2tv+th
// i[4]=2xt+2yu+2tu+2zv-zh
// i[5]=t2+2xv+2yv+2zv-yh
// i[6]=2x+2y+2z+2t+2u+v-h

string vars[] = {"x", "y", "z", "t", "u", "v", "h"};
string j[] = {"2x2+2y2+2z2+2t2+2u2+v2-vh",
              "xy+yz+2zt+2tu+2uv+uh",
              "2xz+2yt+2zu+u2+2tv+th",
              "2xt+2yu+2tu+2zv-zh",
              "t2+2xv+2yv+2zv-yh",
              "2x+2y+2z+2t+2u+v-h"};

typedef monomial<sizeof(vars)/sizeof(vars[0]), GrRevLex> mon;
typedef aTerm<mon, modp<7583>> term;
typedef polynomial<term> poly;
typedef module_monomial<mon, TOP> mod;
typedef j_pair<mod, poly> uv;

template<class MONOM, class ORD> vector<MONOM> module_monomial<MONOM, ORD>::generator_monomials;
template<class MONOM, class ORD> vector<int> module_monomial<MONOM, ORD>::generator_degrees;

int main() {
    singular<poly> singularFormatter;
    // ALL 2 OF THE NEXT LINES ARE EXAMPLE DEPENDENT
    singularFormatter.addVariables(vars, sizeof(vars)/sizeof(vars[0]));
    singularFormatter.addPolynomials(j, sizeof(j)/sizeof(j[0]));

    // cout << "Original polynomials: " << endl;
    // for(int i = 0; i < sizeof(j)/sizeof(j[0]); i++)
    //     cout << "j[" << (i+1) << "] = " << j[i] << endl;
    // cout << endl;

    // cout << "Polynomials loaded. they print as: " << endl;
    // singularFormatter.printBasis(cout, "j");
    // cout << endl;

    gvw<term, mod> gb(&singularFormatter, "katsura5");

    return 0;
}

Listing 7: A simple function main that utilizes the code to follow

A.3 The Main Algorithm
```cpp
#ifndef GVW_H
#define GVW_H

#include <sstream>
#include <fstream>
#include <iosfwd>
#include <vector>
#include <list>
#include <queue>
#include "j−pair.h"
#include "polynomial.h"
#include "polyQueue.h"
#include <iterator>
#include <algorithm>
#include <utility>
#include <cstdlib>
#include "poly_formatter.h"
#include "pqueue.h"

using namespace std;

extern const char * sing;
int jPairs = 0;

template<class term, class mod_elmt>
class gvw {
  public:

      typedef polynomial<term> poly;
      typedef vector<poly> polyList;
      typedef j_pair<mod_elmt, poly> jpair;
      typedef typename mod_elmt::monomial mod_mon;
      typedef typename poly::coefficient coefficient;
      typedef typename poly::monomial monom;
      typedef typename mod_elmt::mod_order module_ordering;

      gvw(poly_formatter<poly> * format, const char * name = "gb") :
        reg0red(0), sup0red(0), polyFormatter(format),
        isReduced(false), isGroebner(false) {
          polyList  vee = polyFormatter−>getBasis();

          /////////////// RUN THE ALGORITHM ///////////////
          // ///// THIS STUFF COULD BE IN FUNCTION MAIN() /////
          calculateGB(vee);

          #ifdef BRIEF
            printBriefStatistics(cout);
          #else
            printStatistics(cout, true);
          #endif
```

97
#endif

// uncomment for the basis output to a file

/*
 * ofstream file(str.str().c_str()); */
/*
 * file << sing << endl; */
/*
 * // polyFormatter->updateBasis(getBasis()); */
/*
 * // polyFormatter->printBasis(file, "j"); */
/*
 * makeMinimal(); */
*/

/*
 * vector<poly>* vec = &V; */
/*
 * if(vec->empty()) vec = &minimalBasis; */
*/

/*
 * for(int i = 0; i < vec->size(); i++) { */
/*
 * file << "j"<< (i+i) << "] = " ; */
/*
 * polyFormatter->printPoly(file, vec->operator[]()[i]); */
/*
 * file << ";" << endl; */
/*
 * } */
/*
 * // file << "validate2ideals(sortIdeal(j), makeMonic(groebner(i)));" << endl; */
*/

const polyList & getBasis() {
    if(!minimalBasis.empty()) return minimalBasis; else return V; }

const polyList & getMinimalBasis() {
    makeMinimal();
    return minimalBasis;
}

const polyList & getReducedBasis() {
    makeReduced();
    return minimalBasis;
}

void calculateGB(const polyList & vee) {
    if(isGroebner) return;
    mod_elt::set_generators(vee);
    H.resize(vee.size());

    for(int i = 0; i < vee.size(); i++)
        JPairs.enqueue(jpair(mod_elt(i), vee[i]));

    while(!JPairs.empty())
        processJPair(JPairs.dequeue());

    isGroebner = true;
    sizeOfBasis = V.size();
private:

int findReductor(const mod_elt & u, const term & v, monom & scale){
// -1 = No Reductor, -2 = Super Top-Reducible
int answer = -1;
for(int i = 0; i < VLeadMonoms.size(); i++) {
    if(v.m().isDivisibleBy(VLeadMonoms[i])) {
        scale = v.m() / VLeadMonoms[i];
        mod_elt scaledSig = U[i] * scale;
        if(scaledSig < u) return i;
        if(scaledSig == u) answer = -2;
    }
}
return answer;
}

int regReduce(mod_elt & u, poly & v) {
int result = topReduce(u,v);
if(result < 0) return result;
poly vee;
while(!v.isZero()) {
    vee.push_back(v.lt());
    v.pop_front();
    topReduce(u,v);
}
return -1;
}

int topReduce(mod_elt & u, poly & v) { // POLYQUEUE
const coefficient one(1);
if(v.isZero()) return -1;
monom scale;
polyQueue<poly> queue(v);
while(true) {
    if(queue.isZero()) { // new syzygy
        v = poly();
        return -1;
    }
    int reductor = findReductor(u, queue.lt(), scale);
    if(reductor == -2) { // irreducible but not primitive
        return -2;
    }
    if(reductor == -1) { // primitive
        v = queue.toPoly();
        return reductor;
    }
    coefficient c = queue.lc() / V[reductor].lc();
    queue.sub2cancel(V[reductor]*term(scale,c));
}
void anticipateSyzygies(mod_elt u, term v) {
    int j, size = U.size();
    for(j = 0; j < size; j++) {  // Calculate a new Syzygy
        mod_elt syzScaleSig[2] = { U[j]*v.m(), u*VLeadMonoms[j] };
        if(syzScaleSig[0] != syzScaleSig[1] || V[j].lc() != v.c()) {
            int largerSig = (syzScaleSig[0] > syzScaleSig[1]) ? 0 : 1;
            if(!reducibleByH(syzScaleSig[largerSig]))
                insertH(syzScaleSig[largerSig]);
        }
    }
}

void updatePairs(int i) {
    for(int j = 0; j < i; j++) {
        monom lcm = VLeadMonoms[i].lcm(VLeadMonoms[j]);
        monom scale[2] = { lcm / VLeadMonoms[i], lcm / VLeadMonoms[j] };
        int indices[2] = { i, j };
        if(scaleSig[0] == scaleSig[1]) continue;
        int largerSig = (scaleSig[0] > scaleSig[1]) ? 0 : 1;
        if(!reducibleByH(scaleSig[largerSig])) {
            jpair jp(scale[largerSig], indices[largerSig],
                    U[indices[largerSig]],
                    VLeadMonoms[indices[largerSig]],
                    V[indices[largerSig]].size());
            JPairs.enqueue(jp);
        }
    }
}

void processJPair(jpair jp) {
    mod_elt u = jp.uPart();
    if(!reducibleByH(u) && isMPair(u, jp.vMonomial())) {
        jPairs++;
        poly v = jp.vPolynomial(V);
        int result = regReduce(u,v);
        if(result == -2) {  // super top-reducible
            sup0red++;
            return;
        }
        if(v.isZero()) {
            reg0red++;
            insertH(u);
            return;
        }
    }
    anticipateSyzygies(u,v.lt());
    U.push_back(u);
    V.push_back(v);

100
VLeadMonoms.push_back(v.lm());
updatePairs(U.size()−1);
}
}

bool isMPair(const mod_elt & u, const monom & lm) {
    for(int i = 0; i < U.size(); i++) {
        if(u.isDivisibleBy(U[i])) {
            monom m = u.m() / U[i].m();
            if(lm > VLeadMonoms[i] * m) return false;
        }
    }
    return true;
}

void printH() {
    for(int i = 0; i < H.size(); i++) {
        cout << "H[" << (i + 1) << "]\'.size() = " << H[i].size() << endl;
        for(int j = 0; j < H[i].size(); j++) {
            cout << "\(" << (i + 1) << ")\'."; polyFormatter->printMonom(cout, H[i][j], false);
            cout << endl;
        }
    }
}

bool reducibleByH(const mod_elt & m) {
    const mod_mon & mon = m.m();
    int pos = m.idx();
    const vector<mod_mon> & HList = H[pos];
    for(int i = 0; i < HList.size(); i++)
        if(mon.isDivisibleBy(HList[i])) return true;
    return false;
}

void insertH(const mod_elt & m) {
    vector<mod_mon> & targetList = H[m.idx()];
    const mod_mon & mon = m.m();
    int size = targetList.size();
    vector<mod_mon> newList;
    for(int i = 0; i < size; i++) {
        if(!targetList[i].isDivisibleBy(mon)) {
            newList.push_back(targetList[i]);
        }
    }
    newList.push_back(mon);
    targetList.swap(newList);
}

void makeMinimal() {
    if(!minimalBasis.empty()) return;
}
if(!isGroebner) {
    cerr << "ERROR: non-Groebner basis!" << endl;
    exit(1);
}
list<poly> minimal;
while(!V.empty()) { // save memory
    minimal.push_front(V.back());
    V.pop_back();
}
vector<poly>().swap(V);
minimal.sort();
typename list<poly>::iterator divisor, dividend;
for(divisor=minimal.begin(); divisor!=minimal.end(); ++divisor) {
    dividend = divisor;
    for(++dividend; dividend != minimal.end();)
        if(dividend->isDivisibleBy(*divisor))
            dividend = minimal.erase(dividend);
        else ++dividend;
}
typename list<poly>::iterator itr;
for(itr = minimal.begin();
    itr != minimal.end();
    itr = minimal.erase(itr))
    minimalBasis.push_back(itr->makeMonic());
}

void makeReduced() {
    if(isReduced) return;
    if(!isGroebner) {
        cerr << "ERROR: non-Groebner basis!" << endl;
        exit(1);
    }
    makeMinimal();
    int i, j;
    int redo = 0;
    for(i = 1; i < minimalBasis.size(); i++) {
        // generator i is now fully reduced
        int changed = 0;
        for(j = i - 1; j >= 0; j--)
            changed += minimalBasis[i].divideThrough(minimalBasis[j]);
        if(changed) --i; // redo it
    }
    isReduced = true;
    for(int i = 0; i < minimalBasis.size(); ++i)
        minimalBasis[i].makeMonic();
}

void printStatistics(ostream & o, bool min = true) {
    o << "\nAlgorithm Statistics:" << endl;
    o << "- JPairs processed: " << jPairs << endl;
    if(min && minimalBasis.empty()) makeMinimal();
Listing 8: The GVWHS algorithm at a high level as presented in figure 4.1
A.4 The Priority Queue for the Next J-Pair

@implementation pqueue.h
#ifndef PQUEUE_H
#define PQUEUE_H  // MIN Queue

#include <cstdlib>
#include <iosfwd>
#include <vector>
#include <map>
using std::map;
using std::vector;

template<class J_Pair>
class pqueue {
 public:

typedef typename J_Pair::mod_mon sig;

~pqueue() {
    while (!array.empty()) {
        delete index[array.back()];
        array.pop_back();
    }
}

void enqueue(const J_Pair & j) {
    sig & s = j.uPart();
    J_Pair * ptr = index[s];
    if (!ptr) {
        index[s] = new J_Pair(j);
        array.push_back(s);
        bubble_up(array.size());
        return;
    }
    // insert replacement policy here
    // if (false) {
    //     // P1
    //     if (true) {
    //         // P2
    //         if (ptr->idx() > j.idx()) {
    //             // P3
    //             if (ptr->idx() < j.idx()) {
    //                 // P4
    //                 if (ptr->vMonomial() > j.vMonomial()) {
    //                     // P5 Winner
    //                     if (ptr->vMonomial() < j.vMonomial()) {
    //                         // P6
    //                         if (ptr->theScale() > j.theScale()) {
    //                             // P7
    //                             if (ptr->theScale() < j.theScale()) {
    //                                 // P8
    //                                 if (ptr->tc() < j.tc()) {
    //                                     // P9
    //                                     if (ptr->tc() > j.tc()) {
    //                                         // P10
    //                                         delete ptr;
    //                                         index[s] = new J_Pair(j);
    //                                     }
    //                                 }
    //                             }
    //                         }
    //                     }
    //                 }
    //             }
    //         }
    //     }
    // }
}

104
J_Pair dequeue() {
    sig s = array[0];
    J_Pair answer = *index[s];
    delete index[s];
    index.erase(s);

    array[0] = array.back();
    array.pop_back();
    bubble_down();
    return answer;
}

int size() const { return array.size(); }

bool empty() const { return array.empty(); }

void reHeapify() { // call whenever our term ordering changes (g1/g2)
    for (int i = array.size(); i > 0; i--)
        bubble_down(i);
}

private:

void bubble_up(int start_pos) { // receives 1-based indexing
    if (start_pos > 1 && array[start_pos - 1] < array[start_pos / 2 - 1]) {
        sig temp = array[start_pos - 1];
        array[start_pos - 1] = array[start_pos / 2 - 1];
        array[start_pos / 2 - 1] = temp;
        bubble_up(start_pos / 2);
    }
}

void bubble_down(int start_pos = 1) { // receives 1-based indexing
    if (start_pos * 2 <= array.size()) {
        int theSmaller = start_pos * 2;
        if (theSmaller + 1 <= array.size() &&
            array[theSmaller] > array[theSmaller + 1]) theSmaller++;
        if (array[theSmaller - 1] < array[start_pos - 1]) {
            sig temp = array[theSmaller - 1];
            array[theSmaller - 1] = array[start_pos - 1];
            array[start_pos - 1] = temp;
            bubble_down(theSmaller);
        }
    }
}

std::ostream & assertMinHeap(int j, std::ostream & o) {
    for (int i = 1; i <= array.size(); ++i) {
        if (2*i <= array.size() && array[i-1] > array[2*i-1]) {

Listing 9: A priority queue for the selection of the next J-pair using policies from section 5.1

A.5 The Mergesort Generalization Polynomial Subtractor

/////////////////////////////////////////// polyQueue.h ///////////////////////////////////////////
#ifndef POLYQUEUE_H
#define POLYQUEUE_H

#include <cstdlib>
#include <iostream>
#include <vector>
#include <algorithm>

template<class poly>
class polyQueue {
  public:

    typedef typename poly::term term;
    typedef typename poly::monomial monom;
    typedef typename poly::coefficient coeff;

    polyQueue(const poly & p) {
      array.push_back(new poly(p));
      mults.push_back(coef(1));
    }

    ~polyQueue() {
      for(int i = 0; i < array.size(); i++) delete array[i];
    }

};

#endif
inline const coeff lc() const { return array[0]->lc(); }
inline const monom & lm() const { return array[0]->lm(); }
inline const term & lt() const { return array[0]->lt(); }
inline bool isZero() const { return array.empty(); }

void operator/=(const coeff & c) {
    const coeff one(1);
    if(!array.empty()) {
        if(c != one) {
            coeff cinv = c.inv();
            for(int i = 0; i < mults.size(); i++)
                mults[i] *= cinv;
        }
        scaleZeroth();
    }
}

void sub2cancel(const poly & p) {
    poly * q = new poly(-p);
    array[0]->pop_front(); // leading terms just cancelled
    if(array[0]->isZero()) {
        delete array[0];
        array[0] = array.back();
        mults[0] = mults.back();
        array.pop_back();
        mults.pop_back();
    }
    bubble_down(1);
    q->pop_front();
    if(!q->isZero()) {
        array.push_back(q);
        mults.push_back(coeff(1));
        bubble_up(array.size());
    } else delete q;
    fix_lm();
}

poly toPoly() {
    poly answer;
    while(!isZero()) {
        answer.push_back(lt());
        array[0]->pop_front();
        if(array[0]->isZero()) {
            delete array[0];
            array[0] = array.back();
            mults[0] = mults.back();
            array.pop_back();
            mults.pop_back();
        }
    }
}
bubble_down(1);
fix_lm();

return answer;
}

private:

void scaleZereth() {
    const coeff one(1);
    if(mults[0] != one) {
        array[0]->operator*(mults[0]);
        mults[0] = one;
    }
}

void fix_lm() {
    bool repeat = true;
    while(!array.empty() && repeat) {
        repeat = false;
        for(int pos = 1; pos <= 2; pos++) {
            if(array.size() > pos && array[pos]->lm() == array[0]->lm()) {
                repeat = true;
                array[0]->addCoeff2LC(array[pos]->lc() * mults[pos] / mults[0]);
                array[pos]->pop_front();
                if(array[pos]->isZero()) {
                    delete array[pos];
                    array[pos] = array.back();
                    mults[pos] = mults.back();
                    array.pop_back();
                    mults.pop_back();
                }
            } else {
                bubble_down(pos + 1);
            }
        }
        if(array[0]->lc().isZero()) {
            array[0]->pop_front();
            if(array[0]->isZero()) {
                delete array[0];
                array[0] = array.back();
                mults[0] = mults.back();
                array.pop_back();
                mults.pop_back();
            }
        } else {
            bubble_down(1);
        }
    }
}

const coeff one(1);
if(mults[0] != one && !array.empty()) scaleZereth();}
void bubble_up(int pos) { // one based argument
    while (pos > 1) {
        if (array[(pos >> 1) - 1] -> lm() <= array[(pos >> 1) - 1] -> lm()) break;
        std::swap(array[pos - 1], array[(pos >> 1) - 1]);
        std::swap(mults[pos - 1], mults[(pos >> 1) - 1]);
        pos >>= 1;
    }
}

void bubble_down(int pos = 1) { // one based argument
    if (pos <= array.size()) {
        int theLarger = pos << 1;
        if (theLarger + 1 <= array.size()) {
            array[theLarger - 1] -> lm() < array[theLarger] -> lm())
                theLarger++;
            if (array[theLarger - 1] -> lm() > array[pos - 1] -> lm()) {
                std::swap(array[theLarger - 1], array[pos - 1]);
                std::swap(mults[theLarger - 1], mults[pos - 1]);
                bubble_down(theLarger);
            }
        }
    }
}

std::vector<poly*>& array;
std::vector<coeff> mults;

#endif

Listing 10: Polynomial subtraction as described in section 5.3

A.6 The J-Pair Class

//----------------------------------------------------------------------------- j-pair.h
#ifndef JPAIR_H
#define JPAIR_H

#include <iosfwd>
#include <vector>

template<class mod_elt, class poly>
class j_pair {
    public:

        typedef typename poly::monomial monomial;
        typedef typename poly::term term;
        typedef mod_elt mod_mon;

        j_pair(const monomial & theirScale, int index, const mod_elt & usig,

109
const monomial & vpoly, int tc) : u(sig*theirScale), v(vpoly*theirScale), origUVIdx(index), scale(theirScale), termCount(tc) {
}

j_pair(const mod_elt & u1, const poly & v1) : u(u1), v(v1.lm()), origUVIdx(-1), p(v1) {
    termCount = p.size();
}

inline bool operator<=(const j_pair & r) const { return u <= r.u; }
inline bool operator<=(const j_pair & r) const { return u < r.u; }
inline bool operator>=(const j_pair & r) const { return u >= r.u; }
inline bool operator>=(const j_pair & r) const { return u > r.u; }
inline bool operator==(const j_pair & r) const { return u == r.u; }

// inline int idx() const { return origUVIdx; }
inline const mod_elt & uPart() const { return u; }
inline const monomial & vMonomial() const { return v; }
inline const poly & vPolynomial(const std::vector<poly> & Vlist) {
    if (origUVIdx >= 0) {
        p = Vlist[origUVIdx] * term(scale);
        origUVIdx = -1;
    }
    return p;
}
inline bool isDivisibleBy(const mod_elt & mon) const { return u.isDivisibleBy(mon); }
inline const monomial & theScale() const { return scale; }
inline int tc() const { return termCount; }

void print(std::ostream & o) const {
    o << "( " << u << ', ' << v << ' )" << std::endl;
}

private:

    poly p;
    mod_elt u;
    monomial v;
    int origUVIdx, termCount;
A.7 The Polynomial Class

Listing 11: The J-pair class

```cpp
#include <iostream>
#include <list>
#include <iosfwd>
#include <cstdlib>
#include "aTerm.h"

using std::list;

extern int jPairs;

template<class aterm>
class polynomial {
    public:

    typedef aterm term;
    typedef typename term::monomial monomial;
    typedef typename term::coefficient coefficient;

    inline const coefficient & lc() const { return terms.front().c(); }
    inline const monomial & lm() const { return terms.front().m(); }
    inline const term & lt() const { return terms.front(); }
    inline const list<term> & getList() const { return terms; }
    inline void pop_front() { terms.pop_front(); }
    inline void push_back(const aterm & t) { terms.push_back(t); }
    inline bool isZero() { return terms.empty(); }
    inline void addCoeff2LC(const coefficient & c) {
        terms.front().addCoeff(c);
    }
    inline int size() const { return terms.size(); }

    inline bool operator<(const polynomial & p) const { return lm() < p.lm(); }
    inline bool operator>(const polynomial & p) const { return lm() > p.lm(); }
};
```
inline bool operator<=(const polynomial & p) const {
    return lm() <= p.lm();
}

inline bool operator>=(const polynomial & p) const {
    return lm() >= p.lm();
}

inline polynomial makeMonic() {
    operator/=(lc());
    return *this;
}

polynomial & operator/=(const coefficient & c) {
    if(!c) {
        terms.clear();
        return *this;
    }
    typename list<term>::iterator itr;
    coefficient d = coefficient(1)/c;
    for(itr = terms.begin(); itr != terms.end(); ++itr)
        *itr *= d;
    return *this;
}

inline polynomial operator/(const coefficient & c) {
    polynomial answer = *this;
    answer /= c;
    return answer;
}

polynomial & operator+=(const term & t) {
    if(terms.empty()) {
        terms.push_front(t);
        return *this;
    }
    typename list<term>::iterator itr = terms.begin();
    while(itr != terms.end() && *itr > t) ++itr;
    if(itr == terms.end() || *itr < t) {
        terms.insert(itr, t);
    } else {
        itr->addCoeff(t.c());
        if(!static_cast<bool>(*itr))
            terms.erase(itr);
    }
    combineTerms();
    return *this;
}

polynomial & operator+=(const polynomial & p) {
typename list<term>::iterator myItr = terms.begin();
typename list<term>::const_iterator pltr = p.terms.begin();
for(pltr = p.terms.begin(); pltr != p.terms.end(); ++pltr) {
    while(myItr != terms.end() && *myItr > *pltr) ++myItr;
    myItr = ++terms.insert(myItr, *pltr);
}
combineTerms();
return *this;
}

inline polynomial operator+(const polynomial & p) const {
polynomial answer = *this;
return answer += p;
}

polynomial & operator=(const polynomial & p) {
    typename list<term>::iterator myItr = terms.begin();
typename list<term>::const_iterator pltr = p.terms.begin();
for(pltr = p.terms.begin(); pltr != p.terms.end(); ++pltr) {
    while(myItr != terms.end() && *myItr > *pltr) ++myItr;
    myItr = ++terms.insert(myItr, -*pltr);
}
combineTerms();
return *this;
}

void assertDecreasing(const char * desc) const {
    if(terms.size() < 2) return;
typename list<term>::const_iterator lead, follow, end = terms.end();
    follow = lead = terms.begin();
for(++lead; lead != end; ++lead, ++follow)
    if(*lead >= *follow) {
        std::cerr << "Term list non-decreasing: " << std::endl;
        std::cerr << "called inside: " << desc << std::endl;
exit(0);
    }
}

polynomial operator-() const {
polynomial answer = *this;
typename list<term>::iterator itr;
for(itr = answer.terms.begin(); itr != answer.terms.end(); ++itr)
    itr->neg();
return answer;
}

inline polynomial operator-(const polynomial & p) const {
polynomial answer = *this;
return answer -= p;
}
polynomial & operator\*=\((\text{const} \ \text{term} \ & \ t)\) {
  \textit{typename} list<term>::iterator \textit{itr} ;
  for( \textit{itr} = \textit{terms}.begin() ; \textit{itr} != \textit{terms}.end() ; ++\textit{itr})
    *\textit{itr} *= \textit{t} ;
  \textbf{return} *\textit{this} ;
}

polynomial & operator\*=\((\text{const} \ \text{coefficient} \ & \ c)\) {
  \textit{typename} list<term>::iterator \textit{itr} ;
  for( \textit{itr} = \textit{terms}.begin() ; \textit{itr} != \textit{terms}.end() ; ++\textit{itr})
    \textit{itr}->operator\*=\((c)\);
  \textbf{return} *\textit{this} ;
}

\textit{inline} polynomial operator*\((\text{const} \ \text{term} \ & \ t)\) \textit{const} {
  polynomial answer = *\textit{this} ;
  \textbf{return} answer *= \textit{t} ;
}

\textbf{void} print(\textit{std::ostream} & \textit{o}) \textit{const} {
  if( \textit{terms}.empty() ) {
    \textit{o} \ll 0 ;
    \textbf{return} ;
  }
  \textit{typename} list<term>::const_iterator \textit{itr} ;
  \textit{int} \textit{i} = 1 , \textit{size} = \textit{terms}.size() ;
  for( \textit{itr} = \textit{terms}.begin() ; \textit{itr} != \textit{terms}.end() ; ++\textit{itr}) {
    \textit{itr}->print(\textit{o}) ;
    if( \textit{i++} != \textit{size} ) \textit{o} \ll "+" ;
  }
}

\textit{inline} \textit{bool} isDivisibleBy(\textit{const} polynomial & \textit{p}) \textit{const} {
  \textbf{return} \textit{lm}().isDivisibleBy(\textit{p}\.\textit{lm}()) ;
}

\textbf{int} divideThrough(\textit{const} polynomial & \textit{p}) {
  \textit{typename} std::list<term>::iterator \textit{t} ;
  \textit{int} changed = 0 ;
  for( \textit{t} = \textit{terms}.begin() ; \textit{t} != \textit{terms}.end() \&\& \textit{t}->\textit{m}() \geq \textit{p}\.\textit{lm}() ; ++\textit{t}) {
    if( \textit{t}->\textit{m}()\:\textit{isDivisibleBy}(\textit{p}\.\textit{lm}()) ) {
      \textbf{changed}++ ;
      \textit{term} \textit{factor} = *\textit{t} / \textit{p}\.\textit{lt}() ;
      --\textit{t} ;
      \textbf{operator\-=}(\textit{p} \* \textit{factor}) ;
    }
  }
  \textbf{return} \textit{changed} ;
}
private:

void combineTerms() {
    if (terms.size() > 1) {
        typename list<term>::iterator lead, follow, end = terms.end();
        lead = follow = terms.begin();
        ++lead;
        while (lead != end && follow != end) {
            if (*lead == *follow) {
                follow->addCoeff(lead->c);
                lead = terms.erase(lead);
                if (follow->isZero()) terms.erase(follow);
            }
            follow = lead;
            ++lead;
        }
    }

    list<term> terms;
};

template<class term>
std::ostream & operator<<(std::ostream &, const polynomial<term> & poly) {
    poly.print(o);
    return o;
}

Listing 12: The polynomial class

A.8 The Term Class
inline const field & c() const { return myCoefficient; }  
inline const monomial & m() const { return myMonomial; }  
inline bool isZero() { return myCoefficient.isZero(); }  
inline void neg() { myCoefficient = -myCoefficient; }  
inline bool operator<(const aTerm & right) const {
  return myMonomial < right.myMonomial;
}  
inline bool operator<=(const aTerm & right) const {
  return myMonomial <= right.myMonomial;
}  
inline bool operator>(const aTerm & right) const {
  return myMonomial > right.myMonomial;
}  
inline bool operator>=(const aTerm & right) const {
  return myMonomial >= right.myMonomial;
}  
inline aTerm operator*(const aTerm & term) const {
  return aTerm(myMonomial*term.myMonomial,
  myCoefficient*term.myCoefficient);
}  
inline aTerm operator*(const coefficient & c) {
  return aTerm(myMonomial, myCoefficient*c);
}  
aTerm & operator*=(const aTerm & right) {
  myCoefficient *= right.myCoefficient;
  myMonomial *= right.myMonomial;
  return *this;
}  
inline aTerm & operator*=(const coefficient & c) {
  myCoefficient *= c;
  return *this;
}  
inline aTerm & operator/=(const coefficient & c) {
  myCoefficient /= c;
  return *this;
}  
aTerm operator/(const aTerm & term) const {
  // Implementation
aTerm temp = *this;
temp.myMonomial /= term.myMonomial;
temp.myCoefficient /= term.myCoefficient;
return temp;
}

aTerm & operator/=(const aTerm & term) {
    myMonomial /= term.myMonomial;
    myCoefficient /= term.myCoefficient;
    return *this;
}

inline aTerm operator−() const {
    aTerm answer = *this;
    answer.myCoefficient = −myCoefficient;
    return answer;
}

inline bool operator==(const aTerm & right) const {
    return myMonomial == right.myMonomial;
}

inline bool operator!=(const aTerm & right) const {
    return myMonomial != right.myMonomial;
}

inline operator bool() const {
    return static_cast<bool>(myCoefficient);
}

inline bool isDivisibleBy(const aTerm & x) const {
    return myMonomial.isDivisibleBy(x.myMonomial);
}

inline field addCoeff(const field & c) {
    return myCoefficient += c;
}

inline bool printsNeg() const {
    return myCoefficient.printsNeg();
}

void print(std::ostream & o) const {
    if (myMonomial.isConstant() || !myCoefficient.isOne()) {
        o << myCoefficient;
    }
    if (!myMonomial.isConstant()) o << myMonomial;
}

private:
monom myMonomial;
field myCoefficient;
};

template<class monom, class field>
std::ostream & operator<<(std::ostream & o, 
    const aTerm<monom, field> & term ) {
    term.print(o);
    return o;
}

Listing 13: The term class

A.9 The Monomial Class

chooser monomial.h

ifndef MONOMIAL_H
define MONOMIAL_H

#include<vector>
#include<iosfwd>
#include "term-order.h"

template<int n, class ORDERING, class T = short>
class monomial {
    public:

    typedef T expType;

    static const int numVars = n;

    monomial() {
        for(int i = 0; i < n; i++) exponents[i] = 0;
        degree = 0;
    }

    monomial(const T * expons) {
        degree = 0;
        for(int i = 0; i < n; i++) degree += exponents[i] = expons[i];
    }

    monomial(const std::vector<int> & expons) {
        degree = 0;
        for(int i = 0; i < n; i++) degree += exponents[i] = expons[i];
    }

}
monomial operator*(const monomial & x) const {
    monomial y = x;
    for (int i = 0; i < n; i++) y.exponents[i] += exponents[i];
    y.updateDegree();
    return y;
}

monomial & operator*(const monomial & x) {
    for (int i = 0; i < n; i++)
        exponents[i] += x.exponents[i];
    updateDegree();
    return *this;
}

inline bool isDivisibleBy(const monomial & x) const {
    if (x.degree > degree) return false;
    for (int i = 0; i < n; i++)
        if (x.exponents[i] > exponents[i]) return false;
    return true;
}

monomial operator/((const monomial & x) const {
    monomial y = *this;
    for (int i = 0; i < n; i++)
        y.exponents[i] -= x.exponents[i];
    y.degree -= x.degree;
    return y;
}

monomial & operator/=(const monomial & x) {
    for (int i = 0; i < n; i++)
        exponents[i] -= x.exponents[i];
    degree -= x.degree;
    return *this;
}

inline bool operator==(const monomial & x) const {
    for (int i = 0; i < n; i++)
        if (exponents[i] != x.exponents[i]) return false;
    return true;
}

inline bool operator!=(const monomial & x) const {
    return !operator==(x);
}

inline const T* getExponents() const {
    return exponents;
}

std::vector<int> getExpVector() const {
std::vector<int> answer;
for (int i = 0; i < n; i++)
    answer.push_back(exponents[i]);
return answer;
}

inline int getDegree() const {
    return degree;
}

inline int updateDegree() {
    degree = 0;
    for (int i = 0; i < n; i++) degree += exponents[i];
    return degree;
}

monomial lcm(const monomial & rhs) {
    monomial answer = *this;
    for (int i = 0; i < n; i++)
        if (rhs.exponents[i] > exponents[i])
            answer.exponents[i] = rhs.exponents[i];
    answer.updateDegree();
    return answer;
}

inline bool isConstant() const { return degree == 0; }

void print(std::ostream & o) const {
    const T * e = getExponents();
    const int oneBased = 1;
    for (int i = 0; i < n; i++) {
        if (e[i] > 0) {
            if (i < 10) o << "x_" << (i+oneBased);
            else o << "x_"["" << (i+oneBased) << "]";
            if (e[i] > 1 && e[i] < 10) o << "^" << e[i];
            else if (e[i] > 9) o << "^{" << e[i] << "}";
        }
    }
}

private:

    T exponents[n];
    int degree;
};

template<int n, class ORDERING, class T>
std::ostream & operator<<(std::ostream & o, const monomial<n, ORDERING, T> & x) {
    x.print(o);
};
Listing 14: The monomial class

A.10 The Field Class

const bool negs = false;  // print smallest number in magnitude

template<int p, class T = short>
class modp {
    public:
        modp(T enn = 0) : n(enn%p) { if(n < 0) n = n + p; }

        inline modp operator+(modp enn) const { return modp((enn.n+n)%p); }

        inline modp & operator+=(modp enn) {
            n += enn.n; n %= p; return *this;
        }

        inline modp operator-() const { return modp(p-n); }

        inline modp operator-(modp enn) const {
            return modp((n-enn.n+p)%p);
        }

        inline modp & operator-=(modp enn) {
            n = (n - enn.n + p)%p; return *this;
        }

        inline modp operator*(modp enn) const { return modp((enn.n*n)%p); }

        inline modp & operator*=(modp enn) { n = (n * enn.n) % p; }

        inline modp inv() const { return modp(egcd(p, n, 0, 1)); }

        modp operator/(modp enn) const {
            if(enn.n == 0) {
                return o;
            }
std::cerr << "Division by zero (modp::operator/)
" << std::endl;
exit(1);
return *this;
}
modp answer = *this;
return answer * enn.inv();
}

modp & operator/=(modp enn) {
if(enn.n == 0) {
std::cerr << "Division by zero (modp::operator/=)"
<< std::endl;
exit(1);
return *this;
}
operator*=(enn.inv());
return *this;
}

inline bool operator==(modp enn) const { return n == enn.n; }
inline bool operator!=(modp enn) const { return n != enn.n; }
inline operator bool() const { return n != 0; }

// inline T getInt() const { return n; }

inline bool isOne() const { return n == 1; }
inline bool isZero() const { return n == 0; }
inline bool printsNeg() const {
const T p_over_two = p/2;
if(!negs) return false;
return n > p_over_two;
}

void print(std::ostream & o) {
const int p_over_two = p/2;
T enn = n;
if(negs && enn > p_over_two) enn -= p;
o << enn;
}

private:

T n;

T egcd(T x, T y, T i, T j) const {
if(y == 0) return i;
return egcd(y, x%y, j, i-(x/y)*j);
}
template<int p>
std::ostream & operator<<(std::ostream & o, mod<p> en) {
    en.print(o);
    return o;
}

Listing 15: The field class

A.11 The Term Orders

#ifndef TERMORDER_H
#define TERMORDER_H

template<int n, class ORDERING, class T>
class monomial;

class Lex;
class GrRevLex;
class BackwardLex;

// class Lex

template<int n, class T>
bool inline operator<=(const monomial<n, Lex, T> & l,
                        const monomial<n, Lex, T> & r) {
    const T * left = l.getExponents();
    const T * right = r.getExponents();
    for(int i = 0; i < n; i++) {
        if(left[i] > right[i]) return false;
        if(left[i] < right[i]) return true;
    }
    return true;
}

template<int n, class T>
bool inline operator<=(const monomial<n, Lex, T> & l,
                        const monomial<n, Lex, T> & r) {
    const T * left = l.getExponents(), * right = r.getExponents();
    for(int i = 0; i < n; i++) {
        if(left[i] > right[i]) return false;
        if(left[i] < right[i]) return true;
    }
    return false;
}
// BackwardLex

template<int n, class T>
bool inline operator<=(const monomial<n, BackwardLex,T> & l, const monomial<n, BackwardLex, T> & r) {
    const T * left = l.getExponents();
    const T * right = r.getExponents();
    for(int i = n-1; i >= 0; i--) {
        if(left[i] > right[i]) return false;
        if(left[i] < right[i]) return true;
    }
    return true;
}

template<int n, class T>
bool inline operator<=(const monomial<n, BackwardLex,T> & l, const monomial<n, BackwardLex, T> & r) {
    const T * left = l.getExponents(), * right = r.getExponents();
    for(int i = n-1; i >= 0; i--) {
        if(left[i] > right[i]) return false;
        if(left[i] < right[i]) return true;
    }
    return false;
}

// GrRevLex

template<int n, class T>
bool inline operator<=(const monomial<n, GrRevLex,T> & l, const monomial<n, GrRevLex, T> & r) {
    if(l.getDegree() < r.getDegree()) return true;
    if(l.getDegree() > r.getDegree()) return false;
    const T * left = l.getExponents();
    const T * right = r.getExponents();
    for(int i = n-1; i >= 0; i--) {
        if(left[i] > right[i]) return true;
        if(left[i] < right[i]) return false;
    }
    return true;
}

template<int n, class T>
bool inline operator<=(const monomial<n, GrRevLex,T> & l, const monomial<n, GrRevLex, T> & r) {
    if(l.getDegree() < r.getDegree()) return true;
    if(l.getDegree() > r.getDegree()) return false;
    const T * left = l.getExponents();
    const T * right = r.getExponents();
    for(int i = n-1; i >= 0; i--) {
        if(left[i] > right[i]) return true;
        if(left[i] < right[i]) return false;
    }
}


Listing 16: The term orders

A.12 The Module Term Class

#include <iosfwd>
#include <vector>

template<class monom, class module_ordering>
class module_monomial {

public:

typedef monom monomial;
typedef module_ordering mod_order;

module_monomial(const monom & m, int idx) {
    myMon = m;
    myIdx = idx;
    updateGWeighted();
}

module_monomial(int idx) { // constructs a unit vector
    myIdx = idx;
    myWeightedMon = generator_monomials[idx];
    myWeightedDegree = generator_degrees[idx];
}
inline const monom & m() const { return myMon; }
inline int idx() const { return myIdx; }
inline const monom & wm() const { return myWeightedMon; }
inline int wd() const { return myWeightedDegree; }

inline bool isDivisibleBy(const module_monomial & m) const {
    if(m.myIdx != myIdx) return false;
    return myMon.isDivisibleBy(m.myMon);
}

inline bool isDivisibleBy(const monom & m) const {
    return myMon.isDivisibleBy(m);
}

template<class poly>
static void set_generators(const std::vector<poly> & g) {
    for(int i = 0; i < g.size(); i++) {
        generator_monomials.push_back(monom(g[i].lm().getExponents()));
        generator_degrees.push_back(generator_monomials[i].getDegree());
    }
}

inline bool operator==(const module_monomial & m) const {
    return myIdx == m.myIdx && myMon == m.myMon;
}

inline bool operator!=(const module_monomial & m) const {
    return myIdx != m.myIdx || myMon != m.myMon;
}

template<class otherMonomial>
module_monomial operator*(const otherMonomial & mon) const {
    module_monomial answer = *this;
    monom castedMonom(mon.getExponents());
    answer.myMon *= castedMonom;
    answer.updateGWeighted();
    return answer;
}

void print(std::ostream & o) const {
    const int oneBased = 1;
    o << '( ' << (myIdx+oneBased) << ',' << myMon << ' )';
}

private:

void updateGWeighted() {
    myWeightedDegree = myMon.getDegree() + generator_degrees[myIdx];
    myWeightedMon = myMon * generator_monomials[myIdx];
}
monom myMon;
monom myWeightedMon;
int myIdx;   // index within R'm, 1 <= myIdx <= m
int myWeightedDegree;
static std::vector<monom> generator_monomials;
static std::vector<int> generator_degrees;
}

template<class monom, class module_ordering>
std::ostream & operator<<(std::ostream & o,
            const module_monomial<monom, module_ordering> & m) {
        m.print(o);
        return o;
    }
#endif

Listing 17: The module term class

A.13 The Module Term Orders

#ifndef MODULEORDER_H
#define MODULEORDER_H
#include "module-monomial.h"

class POT { public: static const int mode = 1; };
class TOP { public: static const int mode = 2; };  
class g1 { public: static const int mode = 3; };  
class g2 { public: static const int mode = 4; };  

// class POT

template<class monomial>
bool inline operator<=(const module_monomial<monomial, POT> & m1,
            const module_monomial<monomial, POT> & m2) {
        if(m1.idx() < m2.idx()) return true;
        if(m1.idx() > m2.idx()) return false;
        return m1.m() <= m2.m();
    }

template<class monomial>
bool inline operator<(const module_monomial<monomial, POT> & m1,
            const module_monomial<monomial, POT> & m2) {
        if(m1.idx() < m2.idx()) return true;
        if(m1.idx() > m2.idx()) return false;
        return m1.m() < m2.m();
    }

127
// greater than

template<class monomial, class mod_ord>
bool inline operator > (const module_monomial<monomial, mod_ord> & m1,
                     const module_monomial<monomial, mod_ord> & m2) {
  return !(m1 <= m2);
}

template<class monomial, class mod_ord>
bool inline operator >= (const module_monomial<monomial, mod_ord> & m1,
                        const module_monomial<monomial, mod_ord> & m2) {
  return !(m1 < m2);
}

// class TOP

template<class monomial>
bool inline operator <= (const module_monomial<monomial, TOP> & m1,
                      const module_monomial<monomial, TOP> & m2) {
  if (m1.m() < m2.m()) return true;
  if (m1.m() > m2.m()) return false;
  return m1.idx() <= m2.idx();
}

template<class monomial>
bool inline operator < (const module_monomial<monomial, TOP> & m1,
                      const module_monomial<monomial, TOP> & m2) {
  if (m1.m() < m2.m()) return true;
  if (m1.m() > m2.m()) return false;
  return m1.idx() < m2.idx();
}

// class g1

template<class monomial>
bool inline operator <= (const module_monomial<monomial, g1> & m1,
                      const module_monomial<monomial, g1> & m2) {
  if (m1.wd() < m2.wd()) return true;
  if (m1.wd() > m2.wd()) return false;
  if (m1.m() < m2.m()) return true;
  if (m1.m() > m2.m()) return false;
  return m1.idx() <= m2.idx();
}

template<class monomial>
bool inline operator < (const module_monomial<monomial, g1> & m1,
                      const module_monomial<monomial, g1> & m2) {
  if (m1.wd() < m2.wd()) return true;
  if (m1.wd() > m2.wd()) return false;
  if (m1.m() < m2.m()) return true;
}
if (m1.m() > m2.m()) return false;
return m1.idx() < m2.idx();
}

// class g2

template<class monomial>
bool inline operator<=(const module_monomial<monomial, g2>& m1,
                          const module_monomial<monomial, g2>& m2) {
    if (m1.wm() < m2.wm()) return true;
    if (m1.wm() > m2.wm()) return false;
    if (m1.idx() < m2.idx()) return true;
    if (m1.idx() > m2.idx()) return false;
    return m1.m() <= m2.m();
}

template<class monomial>
bool inline operator<=(const module_monomial<monomial, g2>& m1,
                          const module_monomial<monomial, g2>& m2) {
    if (m1.wm() < m2.wm()) return true;
    if (m1.wm() > m2.wm()) return false;
    if (m1.idx() < m2.idx()) return true;
    if (m1.idx() > m2.idx()) return false;
    return m1.m() < m2.m();
}

#endif

Listing 18: The module term orders as described in section 3.3.3

A.14 The Polynomial Formatter Base Class

### poly_formatter.h

```cpp
#ifndef POLY_FORMATTER_H
#define POLY_FORMATTER_H

#include <string>
#include <iosfwd>
#include <map>
#include <vector>

template<class poly>
class poly_formatter {
    public:
        typedef typename poly::monomial monomial;
        typedef typename poly::coefficient coefficient;
        typedef typename poly::term term;

        virtual void printMonom(std::ostream & o, const monomial & m,
                                bool printedCoeff) = 0;
}
```

129
virtual void printPoly(std::ostream & o, const poly & p) = 0;
virtual void printBasis(std::ostream & o,
    const std::string & name) = 0;
virtual int addPolynomial(const std::string & p) = 0;

poly_formatter() : varNumber(0) {
}

int addVariable(const std::string & s) {
    varMapping[s] = varNumber++;
    revMapping.push_back(s);
    if (revMapping.size() > monomial::numVars) {
        std::cerr << "ERROR: Inserted more than " << monomial::numVars
            << " variables.\n\n" << std::endl;
        exit(1);
    }
    return 1;
}

int addVariables(std::string c[], int n) {
    for (int i = 0; i < n; i++) addVariable(c[i]);
}

int addVariables(const char * f, int start, int end) {
    char buffer[256];
    for (int i = start; i <= end; i++) {
        sprintf(buffer, f, i);
        addVariable(buffer);
    }
    return end - start + 1;
}

const std::vector<poly> & getBasis() { return theBasis; }

void printVariables() {
    std::map<std::string, int>::iterator itr;
    std::cout << "\nvarMapping\nrevMapping" << std::endl;
    for (itr = varMapping.begin(); itr != varMapping.end(); ++itr)
        std::cout << itr->first << "\n\n" << itr->second << "\n\n" << revMapping[itr->second] << std::endl;
}

void updateBasis(const std::vector<poly> & basis) {
    theBasis = basis;
}

void addPolynomials(const std::string * p, int n) {
    for (int i = 0; i < n; i++) {
        addPolynomial(p[i]);
    }
}
protected:

    int varNumber;
    std::vector<poly> theBasis;
    std::map<std::string, int> varMapping;
    std::vector<std::string> revMapping;
};

#endif

Listing 19: The polynomial formatter base class to read and write polynomials

A.15 The Polynomial Formatter Singular Specialization Class

#ifndef SINGULAR_H
#define SINGULAR_H

#include "poly_formatter.h"
#include <iostream>
#include <map>
#include <list>
#include <vector>
#include <iosfwd>
#include <cstdlib>
#include <cstdio>
#include <cctype>
#include <string>

const bool debug = false;

typedef std::string::size_type size_t;

template<class poly>
class singular : public poly_formatter<poly> {
public:

    typedef typename poly::monomial monomial;
    typedef typename monomial::expType expType;
    typedef typename poly::coefficient coefficient;
    typedef typename poly::term term;

    void printMonom(std::ostream & o, const monomial & m,
        bool printedCoeff) {
        std::vector<int> expons = m.getExpVector();
        const int numVars = monomial::numVars;
        bool timesFlag = printedCoeff;
        int lastPrt = -1;
        for(int j = 0; j < numVars; j++) {
            if(expons[j] == 0) continue;
if (timesFlag) {
    if ((lastPrt >= 0
         && poly_formatter<poly>::revMapping[lastPrt].size() > 1)
        || poly_formatter<poly>::revMapping[j].size() > 1)
        o << "*";
    }

    lastPrt = j;
    o << poly_formatter<poly>::revMapping[j];
    if (expons[j] > 1) {
        if (poly_formatter<poly>::revMapping[j].length() > 1) o << "\n";
        o << expons[j];
    }
}
timesFlag = true;
}

void
printPoly(std::ostream & o, const poly & p) {
    const std::list<term> & termList = p.getList();
    typename std::list<term>::const_iterator itr;
    bool needPlus = false;
    for (itr = termList.begin(); itr != termList.end(); ++itr) {
        if (needPlus && !itr->printsNeg()) o << "+">
        needPlus = true;
        if (itr->m().isConstant() || !itr->c().isOne()) o << itr->c();
        if (!itr->m().isConstant()) {
            printMonom(o, itr->m(), !itr->c().isOne());
        }
    }
}

void
printBasis(std::ostream & o, const std::string & idealName) {
    for (int i = 0; i < poly_formatter<poly>::theBasis.size(); i++) {
        o << idealName << "'" << (i+1) << "\n";
        printPoly(o, poly_formatter<poly>::theBasis[i]);
        o << " '; " << std::endl;
    }
}

int
addPolynomial(const std::string & p) {
    if (debug) std::cerr << "———Calling addPolynomial\n" << p
                        << std::endl;
    size_t start = 0, end = p.length();
    int monomialCount = 0;
    // term is negative, the '-' needs to be position 0
    poly thisPoly;
    while (start <= p.length() - 1) {
        monomialCount++;
        end = p.find_first_of("+-", start + 1);
        if (end > p.length()) end = p.length();
        if (debug) std::cerr << "———Calling getNextMon\n" << p.substr(start, end - start) << " ",

    }
}
<< std::endl;
  thisPoly += getNextMon(p, start, end);
  start = end;
}
poly_formatter<poly>::theBasis.push_back(thisPoly);
return monomialCount;
}

private:

term getNextMon(const std::string &p, size_t &start, size_t end) {
  if (debug) std::cerr << "Over[" << start << "," << end << "]:"
  if (p[start] == '+') ++start;
  bool negative = (p[start] == '-') ? true : false;
  if (debug && negative) std::cerr << "NEG;";
  if (negative) ++start;

  int intCoeff = 1;
  if (!isdigit(p[start])) {
    sscanf(p.substr(start, end-start).c_str(), "%d", &intCoeff);
    start = p.find_first_not_of("0123456789", start);
    if (debug) std::cerr << "Coef(" << intCoeff << ")";
  }
  if (debug) std::cerr << std::endl;
  if (negative) intCoeff = -intCoeff;
  const static int numVars = monomial::numVars;
  std::vector<int> expons(numVars);
  while (start < end) {
    start = p.find_first_not_of("*", start);

    std::string varName(1, p[start++]);
    if (p[start] == '(') {
      int lastParen = p.find_first_not_of("(0123456789)", start);
      varName += p.substr(start, lastParen - start);
      start = lastParen;
    }
    if (debug) std::cerr << \\
      varName(" << varName << ")\n  if (debug) std::cerr << "varIdx(" << poly_formatter<poly>::varMapping[varName] << ");
  if (p[start] == '^') ++start;
  if (isdigit(p[start])) {
    int lastOfExp = p.find_first_not_of("1234567890", start);
    int exp;
    sscanf(p.substr(start, lastOfExp-start).c_str(), "%d", &exp);
    expons[poly_formatter<poly>::varMapping[varName]] += exp;
    start = lastOfExp;
  } else expons[poly_formatter<poly>::varMapping[varName]] += 1;
  if (debug) std::cerr << "exp(" << expons[poly_formatter<poly>::varMapping[varName]]
}
Listing 20: The polynomial formatter Singular class
Appendix B  C++ Code for Binary GVW as Described in Section 5.4

B.1 The Main Algorithm as Described in Section 5.4

```cpp
# ifndef GVWH
# define GVWH

#include <sstream>
#include <fstream>
#include <iosfwd>
#include <cstdio>
#include <vector>
#include <list>
#include <queue>
#include "j_pair.h"
#include "polynomial2.h"
#include "monomial4.h"
#include "polyQueue.h"
#include <iterator>
#include <algorithm>
#include <utility>
#include <cstdlib>
#include "poly_formatter.h"

extern const char * sing;

using namespace std;

template<class mon, class mod_elt>
class gvw {

public:

typedef polynomial<mon> poly;
typedef vector<poly> polyList;
typedef j_pair<mod_elt, poly> jpair;
typedef typename mod_elt::monomial mod_mon;
typedef typename poly::monomial monom;
typedef typename mod_elt::mod_order module_ordering;

const polyList & getBasis() { return V; }
const polyList & getMinimalBasis() {
    makeMinimal();
    return minimalBasis;
}
const polyList & getReducedBasis() {
    makeReduced();
}
```
\begin{verbatim}
return minimalBasis;
}

gvw(poly_formatter<poly> * format, const char * name = "gb") :
JPairs(0), reg0red(0), sup0red(0), polyFormatter(format),
isReduced(false), isGroebner(false) {
    for(int i = 0; i < monom::numVars; i++) {
        fieldMonomials4.push_back(mod_mon(i));
        fieldMonomials2.push_back(monom(i));
    }
    V = polyFormatter->getBasis();
    mod_elt::set_generators(V);
    H.resize(V.size());
    ///////////// RUN THE ALGORITHM /////////////
    // THIS STUFF COULD BE IN FUNCTION MAIN() ///////////
    calculateGB();
    if defined BRIEF
        printBriefStatistics(cout);
    else
        printStatistics(cout);
    endif

    ostringstream str;
    str << "output/" << name << ".txt";
    ofstream file(str.str().c_str());
    file << sing << endl;
    file << "ideal j = ideal();" << endl;
    polyFormatter->updateBasis(getReducedBasis());
    polyFormatter->printBasis(file, "j", 2);
    file << "j = sortIdeal(j);" << endl;
    file << "validate2ideals(j, groebner(i));" << endl;
}

void calculateGB() {
    if(isGroebner) return;
    int i, size = V.size();
    for(i = 0; i < size; i++) {
        VLeadMonoms2.push_back(V[i].lm());
        VLeadMonoms4.push_back(mod_mon(V[i].lm()));
        U.push_back(mod_elt(i));
    }

    for(i = 0; i < size; i++) {
        jpair jp(monom(), i, U[i], VLeadMonoms2[i]);
        JPairs.enqueue(jp);
        updatePairs(i);
    }
    while(!JPairs.empty()) processJPair(JPairs.dequeue());
    isGroebner = true;
\end{verbatim}
private:

int findReductor(const mod_elt & u, const mon & v, monom & scale) {
    // -1 = No Reductor, -2 = Super Top-Reducible
    int answer = -1;
    for (int i = 0; i < VLeadMonoms2.size(); i++) {
        if (v.isDivisibleBy(VLeadMonoms2[i])) {
            monom scaledSig = U[i] * mod_mon(scale);
            if (scaledSig < u) return i;
            if (scaledSig == u) answer = -2;
        }
    }
    if (answer == -2) return answer;
    for (int i = 0; i < VLeadMonoms2.size(); i++) {
        if (v.isDivisibleBy(VLeadMonoms2[i]) && u.isDivisibleBy(U[i]))
            return -2;
    }
    return answer;
}

/* int regReduce(mod_elt & u, poly & v) */
/* result = topReduce(u,v); */
/* if(result < 0) return result; */
/* poly vee; */
/* while(!v.isZero()) */
/* vee.push_back(v.lm()); */
/* v.pop_front(); */
/* topReduce(u,v); */
/* v = vee; */
/* return -1; */
/* */

/* int topReduce(mod_elt & u, poly & v) */
/* // NAIVE POLY SUBTRACTION */
/* if(v.isZero()) return -1; */
/* monom scale; */
/* while(true) */
/* if(v.isZero()) return -1; */
/* int reductor = findReductor(u, v.lm(), scale); */
/* if(reductor < 0) return reductor; */
/* v += V[reductor]*scale; */
/* */
/* */

int topReduce(mod_elt & u, poly & v) { /* POLYQUEUE */
    if(v.isZero()) return -1;
monom scale;
polyQueue<poly> queue(v);
while(true) {
    if(queue.isZero()) {
        v = poly();
        return -1;
    }
    int reducer = findReductor(u, queue.lm(), scale);
    if(reducer == -2) {
        return -2;
    }
    if(reducer == -1) {
        v = queue.toPoly();
        return reducer;
    }
    queue.sub2cancel(V[reductor]*scale);
}

void updatePairs(int i) {
    int j, largerSig;
    static int fieldSize = fieldMonomials2.size();
    for(j = 0; j < fieldSize; j++) { // J-pairs with field equations
        if(!VLeadMonoms2[i].isDivisibleBy(fieldMonomials2[j]))
            continue;
        mod_elt scaledSig = U[i]*fieldMonomials4[j];
        if(!scaledSig.m4().squareFree()) continue;
        if(reducibleByH(scaledSig)) continue;
        jpair jp(fieldMonomials2[j], i, scaledSig, VLeadMonoms2[i]);
        JPairs.enqueue(jp);
    }
    for(j = 0; j < i; j++) { // syzygies with previous generators
        mod_elt syzScaleSig[2] = { U[i]*VLeadMonoms4[j],
                                   U[j]*VLeadMonoms4[i] };
        if(syzScaleSig[0] != syzScaleSig[1]) {
            largerSig = (syzScaleSig[0] > syzScaleSig[1]) ? 0 : 1;
            mod_elt & m = syzScaleSig[largerSig];
            if(m.m4().squareFree() && !reducibleByH(m)) insertH(m);
        }
    }
    for(j = 0; j < i; j++) { // J-pairs with previous generators
        monom lcm = VLeadMonoms2[i].lcm(VLeadMonoms2[j]);
        monom scale[2] = { lcm / VLeadMonoms2[i],
                           lcm / VLeadMonoms2[j] };
mod_elt scaleSig[2] = { U[i] * mod_mon(scale[0]), U[j] * mod_mon(scale[1]) };
int indices[2] = { i, j };

if(scaleSig[0] == scaleSig[1]) continue;
largerSig = (scaleSig[0] > scaleSig[1]) ? 0 : 1;

if(!scaleSig[largerSig].m4().squareFree()) continue;
if(reducibleByH(scaleSig[largerSig])) continue;

jpair jp(scale[largerSig], indices[largerSig],
        scaleSig[largerSig],
        VLeadMonoms2[indices[largerSig]]);
JPairs.enqueue(jp);
}

void processJPair(jpair jp) {
    mod_elt u = jp.uPart();
    if(!reducibleByH(u)) {
        jPairs++;
        poly v = jp.vPolynomial(V);
        int result = topReduce(u, v);
        if(result == -2) { // super top-reducible
            sup0red++;
            return;
        }
    }
    if(v.isZero()) {
        reg0red++;
        insertH(u);
        return;
    }
    U.push_back(u);
    V.push_back(v);
    VLeadMonoms2.push_back(v.lm());
    VLeadMonoms4.push_back(mod_mon(v.lm()));
    updatePairs(U.size() - 1);
}

bool reducibleByH(const mod_elt & m) {
    const mod_mon & m1 = m.m4();
    const vector<mod_mon> & HList = H[m.idx()] ;
    for(int i = 0; i < HList.size(); i++)
        if(m1.isDivisibleBy(HList[i])) return true;
    return false;
}

void insertH(const mod_elt & m) {
    vector<mod_mon> & targetList = H[m.idx()];
const mod_mon & ml = m.m4();
int size = targetList.size();
vector<mod_mon> newList;
for (int i = 0; i < size; i++) {
    if (!targetList[i].isDivisibleBy(ml)) {
        newList.push_back(targetList[i]);
    }
}
newList.push_back(ml);
targetList.swap(newList);
}

void makeMinimal() {
    if (!minimalBasis.empty()) return;
    if (!isGroebner) {
        cerr << "ERROR: called makeMinimal() on non-Groebner basis!"
<< endl;
        exit(1);
    }
    list<poly> minimal(V.begin(), V.end());
    minimal.sort();
typename list<poly>::iterator divisor, dividend;
    for (divisor = minimal.begin(); divisor != minimal.end(); ++divisor) {
        dividend = divisor;
        for (++dividend; dividend != minimal.end();)
            if (dividend->isDivisibleBy(*divisor))
                dividend = minimal.erase(dividend);
            else ++dividend;
    }
    typename list<poly>::iterator itr;
    for (itr = minimal.begin(); itr != minimal.end(); ++itr)
        minimalBasis.push_back(*itr);
}

void makeReduced() {
    if (isReduced) return;
    if (!isGroebner) {
        cerr << "ERROR: called makeReduced() on non-Groebner basis!"
<< endl;
        exit(1);
    }
    makeMinimal();
    int i, j;
    int redo = 0;
    for (i = 1; i < minimalBasis.size(); i++) {
        // generator i is now fully reduced
        int changed = 0;
        for (j = i - 1; j >= 0; j--)
            changed += minimalBasis[i].divideThrough(minimalBasis[j]);
        if (changed) --i; // redo it

isReduced = true;

void printH(ostream & o = cout) {
    for(int i = 0; i < H.size(); i++) {
        o << ", size() = " << H[i].size() << endl;
        for(int j = 0; j < H[i].size(); j++) {
            o << "H" << (i+1+monom::numVars) << "", j < H[i].size() ? " size() = " << H[i][j].size() << endl : " 1 = " << H[i].size() << endl;
            o << polyFormatter->printMonom(o, H[i][j], false);  
        }
    }
}

void printStatistics(ostream & o, bool min = true) {
    o << "\nAlgorithm Statistics: " << endl;
    o << "JPairs.processed: " << jPairs << endl;
    if(min && minimalBasis.empty()) makeMinimal();
    if(min) {
        int count = monom::numVars;
        for(int i = 0; i < minimalBasis.size(); i++)
            if(minimalBasis[i].lm().getDegree() > 1) count++;
        o << "Size of Groebner Basis (minimal): " << count << endl;
    }  
    o << "Size of Groebner Basis (computed): " << V.size() << endl;
    o << "Regular Zero Reductions: " << reg0red << endl;
    o << "Super Top-reductions: " << sup0red << endl;
    stringstream str;
    int sizeOfH = 0;
    str << ",
    for(int i = 0; i < H.size(); i++) {
        if(i) str << ",
        sizeOfH += H[i].size();
        str << H[i].size();
    }
    str << ",
    o << "Size of H: " << sizeOfH << " 1 = " << str.str() << endl;
    o << endl;
}

void printBriefStatistics(ostream & o) {
    o << "\nMODE = module ordering::mode = ";
    if(minimalBasis.empty()) makeMinimal();
    int count = monom::numVars;
    for(int i = 0; i < minimalBasis.size(); i++)
        if(minimalBasis[i].lm().getDegree() > 1) count++;
    o << jPairs << ", count = " << count << "/" << V.size() << endl;
    sizeOfH = 0;
    for(int i = 0; i < H.size(); i++) sizeOfH += H[i].size();  
}
Listing 21: The GVW algorithm at a high level as described in section 5.4

B.2 The Polynomial Class as Described in Section 5.4.2

```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#include <iostream>
#include <list>
#include <iosfwd>
#include <cstdlib>
#include "poly_formatter.h"

using std::list;

template<class mon>
class polynomial { 
  public:
    
    typedef mon monomial;

    inline const mon & lm() const { return terms.front(); } 
    inline const list<mon> & getList() const { return terms; } 
    inline void pop_front() { terms.pop_front(); } 

```
inline void push_back(const mon & t) {
    terms.push_back(t);
}
inline bool isZero() { return terms.empty(); }

inline bool operator<(const polynomial & p) const {
    return lm() < p.lm();
}

inline bool operator>(const polynomial & p) const {
    return lm() > p.lm();
}

inline bool operator<=(const polynomial & p) const {
    return lm() <= p.lm();
}

inline bool operator>=(const polynomial & p) const {
    return lm() >= p.lm();
}

void assertDecreasing(const char * desc) const {
    if (terms.size() < 2) return;
    typename list<mon>::const_iterator lead, follow, end = terms.end();
    follow = lead = terms.begin();
    for (++lead; lead != end; ++lead, ++follow)
        if (*lead >= *follow) {
            std::cerr << "Term list nondescending:
            " << std::endl;
            std::cerr << "called inside:
            " << desc << std::endl;
            exit(0);
        }
    }

polynomial & operator+=(const mon & t) {
    if (terms.empty()) {
        terms.push_front(t);
        return *this;
    }
    typename list<mon>::iterator itr = terms.begin();
    while (itr != terms.end() && *itr > t) ++itr;
    if (itr == terms.end() || *itr < t)
        terms.insert(itr, t);
    else
        terms.erase(itr);
    return *this;
}

polynomial & operator+=(const polynomial & p) {
    typename list<mon>::iterator myItr = terms.begin();
    typename list<mon>::const_iterator pItr = p.terms.begin();
    }
for (pItr = p.terms.begin(); pItr != p.terms.end(); ++pItr) {
    while (myItr != terms.end() && *myItr > *pItr) ++myItr;
    if (myItr == terms.end() || *myItr < *pItr)
        myItr = ++terms.insert(myItr, *pItr);
    else
        myItr = terms.erase(myItr); // remove duplicates over F_2
}
return *this;
}

inline polynomial operator+(const polynomial & p) const {
    polynomial answer = *this;
    return answer += p;
}

polynomial & operator*(const mon & t) {
    typename list<mon>::iterator itr;
    for (itr = terms.begin(); itr != terms.end(); ++itr)
        *itr *= t;
    terms.sort();
    terms.reverse();
    combineTerms();
    return *this;
}

/* polynomial & operator*=(const mon & t) { // KEEP SORTED */
/* // make sure coefficient is nonzero */
/* mon multiplier = t, single; */
/* while (multiplier.stripNext(single)) { */
/* list<mon> divisible, notDivisible; */
/* while (!terms.empty()) { */
/* if (terms.front().isDivisibleBy(single)) */
/* divisible.push_back(terms.front()); */
/* else */
/* notDivisible.push_back(terms.front() * single); */
/* terms.pop_front(); */
/* } // split into two sorted lists, now merge them */
/* while (!divisible.empty() && !notDivisible.empty()) { */
/* if (divisible.front() > notDivisible.front()) { */
/* terms.push_back(divisible.front()); */
/* divisible.pop_front(); */
/* continue; */
/* } */
/* if (divisible.front() < notDivisible.front()) { */
/* terms.push_back(notDivisible.front()); */
/* notDivisible.pop_front(); */
/* continue; */
/* } */
/* divisible.pop_front(); notDivisible.pop_front(); */
/* // equal and cancel */
/* } */

144
inline polynomial operator*(const mon & t) const {
    polynomial answer = *this;
    answer *= t;
    return answer;
}

void print(std::ostream & o) const {
    if (terms.empty()) {
        o << 0;
        return;
    }
    typename list<mon>::const_iterator itr;
    int i = 1, size = terms.size();
    for (itr = terms.begin(); itr != terms.end(); ++itr) {
        itr->print(o);
        if (i++ != size) o << " + ";
    }
}

inline bool isDivisibleBy(const polynomial & p) const {
    return lm().isDivisibleBy(p.lm());
}

int divideThrough(const polynomial & p) {
    typename std::list<mon>::iterator t;
    int changed = 0;
    for (t = terms.begin(); t != terms.end() && (*t) >= p.lm(); ++t) {
        if (t->isDivisibleBy(p.lm())) {
            changed++;
            mon factor = *t / p.lm();
            --t;
            // assertDecreasing("Inside poly::dividethrough");
            operator+=(p * factor);
        }
    }
    return changed;
}

private:

void combineTerms() {

if (terms.size() > 1) {
    typename list<mon>::iterator lead, follow, end = terms.end();
    lead = follow = terms.begin();
    ++lead;
    do {
        if (*lead == *follow) {
            terms.erase(follow);
            lead = terms.erase(lead);
        }
        follow = lead;
        ++lead;
    } while (lead != end && follow != end);
}

list<mon> terms;
}

template<class mon>
std::ostream & operator<<(std::ostream & o,
                          const polynomial<mon> & poly) {
    poly.print(o);
    return o;
}

Listing 22: The polynomial class as described in section 5.4.2

B.3 The Monomial Class as Described in Section 5.4.1

// //////////////////////////////////////////////////////////////////////////// monomial2.h ////////////////////////////////////////////////////////////////////////////
#ifndef MONOMIAL2_H
#define MONOMIAL2_H

#include<vector>
#include<iosfwd>
#include"term-order2.h"
#include<stdio.h>

template<int n, class ORDERING, class T = unsigned int>
class monomial2 {
    public:

    typedef T expType;
    static const int numVars = n;
    static const int Tsize = sizeof(T) * 8;
    static const int arrayLength = (n + Tsize - 1) / Tsize;

146
monomial2() {
    for (int i = 0; i < arrayLength; i++) exponents[i] = 0;
    degree = 0;
}

monomial2(int k) {
    for (int i = 0; i < arrayLength; i++) exponents[i] = 0;
    exponents[k/Tsize] = 1<<(k%Tsize);
    degree = 1;
}

monomial2(const T * expons) { // takes bit strings
    for (int i = 0; i < arrayLength; i++)
        exponents[i] = expons[i];
    updateDegree();
}

monomial2(const std::vector<int> & expons) {
    // takes integers/exponents
    int k = 0; degree = 0;
    for (int i = 0; i < arrayLength; i++)
    {
        exponents[i] = 0;
        for (int j = 0; j < Tsize; j++)
        {
            degree += expons[k]?1:0;
            exponents[i] |= (expons[k]?1:0)<<j;
            k++;
            if(k == expons.size()) break;
        }
    }
}

inline const T * getExponents() const {
    return exponents;
}

std::vector<int> getExponVector() const {
    std::vector<int> answer;
    for (int i = 0; i < arrayLength; i++)
    {
        for (int j = 0; j < Tsize; j++)
        {
            answer.push_back((exponents[i]>>j)&1);
            if(answer.size() == n) return answer;
        }
    }
    return answer;
}

monomial2 operator*(const monomial2 & x) const {
    monomial2 y = x;
    for (int i = 0; i < arrayLength; i++)
```cpp
    y.exponents[i] &= exponents[i];
y.updateDegree();
    return y;
}

monomial2 & operator*=(const monomial2 & x) {
    for(int i = 0; i < arrayLength; i++)
        exponents[i] &= x.exponents[i];
    updateDegree();
    return *this;
}

inline bool isDivisibleBy(const monomial2 & x) const {
    if(x.degree > degree) return false;
    for(int i = 0; i < arrayLength; i++)
        if((x.exponents[i] & exponents[i]) != x.exponents[i])
            return false;
    return true;
}

monomial2 operator/(const monomial2 & x) const {
    monomial2 y = *this;
    for(int i = 0; i < arrayLength; i++)
        y.exponents[i] &= ~x.exponents[i];
y.degree -= x.degree;
    return y;
}

monomial2 & operator/=(const monomial2 & x) {
    for(int i = 0; i < arrayLength; i++)
        exponents[i] &= ~x.exponents[i];
    degree -= x.degree;
    return *this;
}

inline bool operator==(const monomial2 & x) const {
    for(int i = 0; i < arrayLength; i++)
        if(exponents[i] != x.exponents[i]) return false;
    return true;
}

inline bool operator!=(const monomial2 & x) const {
    return !operator==(x);
}

inline int getDegree() const {
    return degree;
}

inline int updateDegree() {
    degree = 0;
```
```cpp
for (int i = 0; i < arrayLength; i++)
    degree += __builtin_popcount(exponents[i]);
return degree;
}

monomial2 lcm(const monomial2 & rhs) {
    monomial2 answer = *this;
    for (int i = 0; i < arrayLength; i++)
        answer.exponents[i] |= rhs.exponents[i];
    answer.updateDegree();
    return answer;
}

bool stripNext(monomial2 & fillInto) {
    int i, j;
    for (i = 0; i < arrayLength; i++) {
        j = __builtin_ffs(exponents[i]) - 1;
        if (j == -1) continue;
        fillInto = monomial2(Tsize*i+j);
        operator/=(fillInto);
        return true;
    }
    return false; // no more found
}

template<class ORD2>
monomial2 gcd(const monomial2<n,ORD2,T> & rhs) const {
    monomial2 answer = *this;
    const T * rhsExpons = rhs.getExponents();
    for (int i = 0; i < arrayLength; i++)
        answer.exponents[i] &= rhsExpons[i];
    answer.updateDegree();
    return answer;
}

inline bool isConstant() const { return degree == 0; }

void print(std::ostream & o) const {
    const T * e = getExponents();
    const int oneBased = 1;
    for (int i = 0; i < arrayLength; i++) {
        for (int j = 0; j < Tsize; j++) {
            if((e[i]>>(j)&1)) {
                if(i*Tsize+j+oneBased < 10) o << "x_" << (i*Tsize+j+oneBased);
                else o << "x_{" << (i*Tsize+j+oneBased) << '};
            }
        }
    }
}
```
private:

  T  exponents[arrayLength];
  int  degree;
};

template<int n, class ORDERING, class T>
std::ostream & operator<<(std::ostream & o,
  const monomial2<n, ORDERING, T> & x) {
  x.print(o);
  return o;
}

Listing 23: The monomial class as described in section 5.4.1

B.4 The Term Orders as Described in Section 5.4.1

#ifndef TERM2ORDER_H
#define TERM2ORDER_H

template<int n, class ORDERING, class T>
class monomial2;

class Lex;
class GrRevLex;

// class Lex

template<int n, class T>
  bool inline operator<=(const monomial2<n,Lex,T> & l,
    const monomial2<n,Lex, T> & r) {
    const T * left = l.getExponents();
    const T * right = r.getExponents();
    for(int i = l.arrayLength-1; i >= 0; i--) {
      if(left[i] > right[i]) return false;
      if(left[i] < right[i]) return true;
    }
    return true;
  }

template<int n, class T>
  bool inline operator<(const monomial2<n,Lex,T> & l,
    const monomial2<n,Lex, T> & r) {
    const T * left = l.getExponents(), * right = r.getExponents();
    for(int i = l.arrayLength-1; i >= 0; i--) {

if(left[i] > right[i]) return false;
if(left[i] < right[i]) return true;
}
return false;
}

// greater than

template<int n, class ORDERING, class T>
bool inline operator>(const monomial2<n, ORDERING, T> & l, const monomial2<n,ORDERING, T> & r) {
    return !(l <= r);
}

template<int n, class ORDERING, class T>
bool inline operator>=(const monomial2<n,ORDERING, T> & l, const monomial2<n,ORDERING, T> & r) {
    return !(l < r);
}

// class GrRevLex:

template<int n, class T>
bool inline operator<=(const monomial2<n, GrRevLex, T> & l, const monomial2<n,GrRevLex, T> & r) {
    if(l.getDegree() < r.getDegree()) return true;
    if(l.getDegree() > r.getDegree()) return false;
    const T * left = l.getExponents(); const T * right = r.getExponents();
    for(int i = l.arrayLength-1; i >= 0; i--)
    {
        if(left[i] > right[i]) return true;
        if(left[i] < right[i]) return false;
    }
    return true;
}

template<int n, class T>
bool inline operator<(const monomial2<n, GrRevLex, T> & l, const monomial2<n,GrRevLex, T> & r) {
    if(l.getDegree() < r.getDegree()) return true;
    if(l.getDegree() > r.getDegree()) return false;
    const T * left = l.getExponents(); const T * right = r.getExponents();
    for(int i = l.arrayLength-1; i >= 0; i--)
    {
        if(left[i] > right[i]) return true;
        if(left[i] < right[i]) return false;
    }
    return false;
}

//////// reverse variable order

template<int n, class T>
bool reverseOrder(const monomial2<
const monomial2<n, Lex, T> & mon) {
    return false;
}

template<int n, class T>
bool reverseOrder(const monomial2<n, GrRevLex, T> & mon) {
    return true;
}
#endif

Listing 24: The term as described in section 5.4.1

B.5 The Signature Class as Described in Section 5.4.3

// monomial4.h
#ifndef MONOMIAL4_H
#define MONOMIAL4_H

#include<vector>
#include<iosfwd>
#include"term-order4.h"
#include"monomial2.h"
#include<stdio.h>
#include<error.h>

template<int n, class ORDERING, class T = unsigned int>
class monomial4 {
public:

typedef T expType;
static const int numVars = n;
static const int Tsize = sizeof(T) * 4;
static const int arrayLength = (n + Tsize - 1) / Tsize;

    monomial4() {
        for(int i = 0; i < arrayLength; i++) exponents[i] = 0;
        degree = squares = 0;
    }

    monomial4(int k) {
        for(int i = 0; i < arrayLength; i++) exponents[i] = 0;
        exponents[k/Tsize] = 1<<(k%Tsize)<<(1);
        degree = 1;
        squares = 0;
    }

    monomial4(const T * expons) { // takes bit strings
        degree = squares = 0;
        for(int i = 0; i < arrayLength; i++)
```cpp
exponents[i] = expons[i];
updateDegree();
}

monomial4(const std::vector<int> & expons) {
    // takes integers/exponents
    int k = 0; degree = squares = 0;
    for(int i = 0; i < arrayLength; i++) {
        exponents[i] = 0;
        for(int j = 0; j < Tsize; j++) {
            degree += expons[k];
            exponents[i] |= (expons[k] & 3) << (j << 1);
            if(expons[k] > 1) squares++;
            k++;
            if(k == expons.size()) break;
        }
    }
}

monomial4(const monomial2<n,ORDERING,T> & mon2) {
    const int array2Length = (n + 2 * Tsize - 1) / (2 * Tsize);
    const T L16= (T)0xffff0000, R16= (T)0x0000ffff;
    const T L8 = (T)0xff00ff00, R8 = (T)0x00ff00ff;
    const T L4 = (T)0xf0f0f0f0, R4 = (T)0x0f0f0f0f;
    const T L2 = (T)0xc0cccccc, R2 = (T)0x33333333;
    const T L1 = (T)0xaaaaaaaa, R1 = (T)0x55555555;

    const T * expons2 = mon2.getExponents();
    int i, k = 0;
    if(sizeof(T) != 4) error(1,0,
        "Alter monomial4(monomial2) to take different size T");
    for(i = 0; i < array2Length; i++) {
        T left = (expons2[i] & L16)>>16, right = expons2[i] & R16;
        // rewrite this using a macro??
        left = (left & R8) | ((left & L8) << 8);
        right = (right & R8) | ((right & L8) << 8);
        left = (left & R4) | ((left & L4) << 4);
        right = (right & R4) | ((right & L4) << 4);
        left = (left & R2) | ((left & L2) << 2);
        right = (right & R2) | ((right & L2) << 2);
        left = (left & R1) | ((left & L1) << 1);
        right = (right & R1) | ((right & L1) << 1);

        exponents[k++] = right;
        if(k < arrayLength) exponents[k++] = left;
    }
}

std::vector<int> getExponVector() const {
    std::vector<int> answer;
    return answer;
}
```

153
for (int i = 0; i < arrayLength; i++) {
    for (int j = 0; j < Tsize; j++) {
        answer.push_back(((exponents[i] >> (j << 1)) & 3);
        if (answer.size() == n) return answer;
    }
}
error(0, 0, "Got here : _monomial4::getExponVector");
return answer;

monomial4 operator*(const monomial4 & x) const {
    monomial4 y = x;
    for (int i = 0; i < arrayLength; i++)
        y.exponents[i] += exponents[i];
    y.updateDegree(); // recalculate squares
    return y;
}

monomial4 & operator*=(const monomial4 & x) {
    for (int i = 0; i < arrayLength; i++)
        exponents[i] += x.exponents[i];
    updateDegree();
    return *this;
}

inline bool operator==(const monomial4 & x) const {
    for (int i = 0; i < arrayLength; i++)
        if (exponents[i] != x.exponents[i]) return false;
    return true;
}

inline bool operator!=(const monomial4 & x) const {
    return !operator==(x);
}

inline const T * getExponents() const {
    return exponents;
}

inline int getDegree() const {
    return degree;
}

inline int updateDegree() {
    const T odds = static_cast<T>(0x55555555);
    const T evens = static_cast<T>(0xaaaaaaaa);
    degree = squares = 0;
    for (int i = 0; i < arrayLength; i++) {
        degree += __builtin_popcount(exponents[i] & odds);
        squares += __builtin_popcount(exponents[i] & evens);
    }
}
degree += squares << 1;
return degree;
}

inline bool isDivisibleBy(const monomial4 & x) const {
    // only works for multilinear x
    if(squares) error(1, 0,
    "called monomial4::isDivisibleBy on non-squarefree monomial4”);
    if(x.degree > degree) return false;
    for(int i = 0; i < arrayLength; i++)
        if(((x.exponents[i] & exponents[i]) != x.exponents[i])
            return false;
    return true;
}

inline bool squareFree() const {
    return !squares;
}

void print(std::ostream & o) const {
    const T * e = getExponents();
    const int oneBased = 1;
    for(int i = 0; i < arrayLength; i++) {
        for(int j = 0; j < Tsize; j++) {
            if(((e[i]>>(j<<1)&3) {  // if e[i,j] > 0
                if((i*Tsize+j+oneBased < 10) o << "x_
                " << (i*Tsize+j+oneBased);
            else o << "x_{" << (i*Tsize+j+oneBased) << ’}’;
            if(((e[i]>>(j<<1))&3) > 1)
                o << ’\"’ << ((e[i]>>(j<<1))&3);
        }
    }
}

private:

T exponents[arrayLength];
int degree, squares;
};
Listing 25: The signature class as described in section 5.4.3
Appendix C  Singular Code

C.1  G2V as in Figure 2.1

```plaintext
option(redSB);  // cause interred to give reduced groebner basis
option(noredefine);  // quit saying:  // ** redefining this—and—that

// Our version of Singular had a HUGE memory leak in the insert
// function. These next 3 functions are the workaround. F5/F5C code
// was modified to use these 3 functions appropriately.
// insertPosition inserts element i into list l1 at position pos.
proc insertPosition(list l1, def i, int pos) {
    if(pos == 0) { return(insertBeginning(l1,i)); }
    int s = size(l1);
    if(pos == s) { return(insertEnd(l1,i)); }
    return(l1[1..pos],i,l1[(pos+1)..s]);
}

// insertBeginning inserts element i into the beginning of list l1
proc insertBeginning(list l1, def i) {
    return(list(i)+l1);
}

// insertEnd inserts element i into the end of list l1
proc insertEnd(list l1, def i) {
    int s = size(l1);
    l1[s+1] = i;
    return(l1);
}

// gBasis accepts an Ideal I and returns a Groebner basis for I
// gBasis repeatedly calls incBasis to incrementally build a GB
proc gBasis(ideal I) {
    list F; int idx, jdx = 1,0;
    for(int i = 1; i <= size(I); i++) {  // size only counts nonzero elts
        jdx++;
        if(I[jdx] != 0) {
            F[idx] = I[jdx];
            idx++;
        }
        else { continue; }
    }  // ideal to list
    createGlobals(); createStatsVariables();
    Flength = 1;
    list basis = F[1];
}
```

157
list added;
poly next;

system("—ticks—per—sec",1000);

int cputime = timer;

int prev_size = 0;

for(i = 2; i <= size(F); i++) {
    list temp = incBasis(basis,F[i]);
    prev_size = size(temp);
    basis = makeReduced(temp);
    Flength = size(basis);
    if(size(V) > maxVlength) {
        maxVlength = size(V);
    }
}

printf("%");
printf("(\%s,\%s,\%s[\%s],\%s/%s)=(time,

calls_to_reduce[total],\#G(after/before))", 
timer−cputime,totalSprocessed,count_reduce,
count_reduce+count_other_reduce,
size(basis),prev_size);

printf("%s")

ideal answer:
for(i = 1; i <= size(basis); i++) { answer = answer + basis[i]; } 
attrib(answer,"isSB",1); // mark it as a standard basis

return(answer);
}

number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(number(...
i++;
V[i]= v;
U[i] = u;
insertPairs(i);
} else {
  zeroReductions++;
  if (u != 0) {
    H = H + u;
    attrib(H,"isSB",1);
    H = interred(H);
  } // else eventually super top-reducible
}
}
return(V);


proc createGlobals() {
  list U, V, JPidx, JPsig;
  export(U); export(V);
  export(JPidx); export(JPsig);
  ideal H; export(H);
  ideal Fideal; export(Fideal);
  int Flength; export(Flength);
}

proc createStatsVariables() {
  int maxVlength = 0; export(maxVlength);
  int maxSlength = 0; export(maxSlength);
  int totalSprocessed = 0; export(totalSprocessed);
  int zeroReductions = 0; export(zeroReductions);
  int count_reduce=0; export(count_reduce);
  int count_other_reduce = 0; export(count_other_reduce);
}

proc setupVariables(list F, poly g) {
  U, JPidx, JPsig = 1, list(), list();
  H = ideal();
  Fideal = ideal();
  for(int i = 1; i <= size(F); i++) {
    H = H + leadmonom(F[i]);
    U = insertBeginning(U,0);
    Fideal = Fideal + F[i];
  }
  // attrib(H, "isSB", 1);
  attrib(Fideal,"isSB",1);
  poly last = reduce(g,Fideal); count_reduce++;
  if(last != 0) {
\[ V = \text{insertEnd}(F, \text{last}); \]
\{ 
\}
\{ \text{else} \{ \text{return}(0); \} \} \text{ // no need to process this generator} 
\text{return}(1); 
\}

// ///////////////////////////////////////////////////////////////////
// ///////////////////////////////////////////////////////////////////
// anticipating syzygies prevents regular reductions to zero

\text{proc \text{anticipateSyzygies}(\text{poly } u, \text{poly } v) \{ 
  \text{for}(\text{int } i = 1; i <= \text{size}(U); i++) \{ 
    \text{poly } syz1 = u*\text{lead}(V[i]); 
    \text{poly } syz2 = U[i]*\text{lead}(v); 
    \text{if} (syz1 == syz2) \{ i++; \text{continue}; \} 
    \text{if} (syz1 > syz2) \{ \text{H} = \text{H} + \text{leadmonom}(syz1); \} 
    \text{else} \{ \text{H} = \text{H} + \text{leadmonom}(syz2); \} 
  \}
  \text{attrib}(\text{H}, \text{"isSB"}, 1); 
  \text{H} = \text{interred}(\text{H}); 
\}}

// ///////////////////////////////////////////////////////////////////
// ///////////////////////////////////////////////////////////////////
// insertPairs accepts an index and inserts J-pairs between 1 <= j < i 
// and i — implementing steps 0 and 3c of Figure 1. 
// insertPairs calculates all these J-pairs and then inserts them into 
// This function should be a lot simpler, but we wanted it to look 
// like F5/F5C’s implementation.

\text{proc \text{insertPairs}(\text{int } i) \{ 
  \text{attrib}(\text{H}, \text{"isSB"}, 1); 
  \text{poly } t1 = \text{leadmonom}(V[i]); 
  \text{for}(\text{int } j = 1; j < i; j++) \{ 
    \text{poly } t2 = \text{leadmonom}(V[j]); 
    \text{poly } g1 = \text{gcd}(t1, t2); 
    \text{poly } t = (t1*t2)/g1; 
    \text{poly } s1 = t/t1*U[i]; 
    \text{poly } s2 = t/t2*U[j]; 
    \text{poly } s = s1; 
    \text{int } g = i; 
    \text{if}(s2 > s) \{ 
      s = s2; g = j; \text{ // insert this one} 
    \}
  \}
  \text{if}(0 != \text{reduce}(s, \text{H})) \{ 
    \text{int } k, \text{found} = \text{binSearch}(s, \text{JPsig}); 
    \text{if}(\text{found}) \{ \text{JPidx}[k] = g; \} \} \text{ // keep most recent} 
  \text{else} \{ \text{ // not in the list, insert at position } k 
    \text{JPsig} = \text{insertPosition}(\text{JPsig}, s, k-1); 
    \text{JPidx} = \text{insertPosition}(\text{JPidx}, g, k-1); 
    \text{if} (\text{size}(\text{JPsig}) > \text{maxSlength}) \{ 
      \text{maxSlength} = \text{size}(\text{JPsig}); 
    \}
  \}
\}
proc binSearch (poly s, list S) { // returns ([new] location, found)
    if (size(S) == 0) { return(1,0); }
    if (s < S[1]) { return(1,0); }
    int l = 1;
    int r = size(S);
    int k;
    if (s > S[r]) { return(r+1,0); }
    if (s == S[r]) { return(r,1); }
    while (1 < r) { // with each iteration, we have S[1] <= s < S[r]
        k = (l+r) / 2;
        if (S[k] == s) { return(k,1); }
        if (S[k] < s) { l = k + 1; } // go right
        else { r = k; } // go left
    }
    return(1,0); }
}

proc findReductor (poly u, poly v) {
    if (v == 0) { return(0); }
    int answer = 0; // becomes -1 if super top-reducible
    poly t1, t2;
    for (int j = Flength + 1; j <= size(V); j++) {
        t1 = leadmonom(v)/leadmonom(V[j]);
        if (0 != t1) {
            t2 = t1*U[j];
            if (t2 < u) { return(j); }
            if (t2 == u) { answer = -1; }
        }
    }
    return(answer); }

proc topReduce (poly u, poly v) { int j; poly ct1, u1, v1;
    while (1) {
        j = findReductor(u,v);
        if (j==1) {
            return(0,0); } // super top-reducible
        if (j==0) {
            return(u,v); } // not top-reducible
        ct1 = lead(v)/lead(V[j]);
\[ v = \text{reduce}(v - c_{1} \ast V[j], \text{Fideal}); \text{count\_reduce}++; \]

if \((v == 0)\) \{ return\((u, 0)\); \}

Listing 26: G2V in Singular as in figure 2.1

C.2 GVW as in Figure 3.1

option(redSB); // cause \text{interred} to give reduced Groebner basis
option(noredefine); // quit saying: // ** redefining this–and–that
// < "ex.sing";

// We version of Singular had a HUGE memory leak in the insert
// function. These next 3 functions are the workaround. F5/F5C code
// was modified to use these 3 functions appropriately.
// \text{insertPosition} inserts element \( i \) into list \( l1 \) at position \( pos \).

proc insertPosition(list l1, def i, int pos) { 
  if(pos == 1) \{ return(l1[i]+l1); \}
  int s = size(l1);
  if(pos == s+1) \{ l1[pos]=i; return(l1); \}
  return(l1[1..(pos-1)],i,l1[pos..s]);
}

// mode = 1 \Rightarrow \text{POT} \quad < = > \quad \text{is} \quad -1 \quad 0 \quad 1
// mode = 2 \Rightarrow \text{TOP}
// mode = 3 \Rightarrow \text{g–weighted degree then TOP}
// mode = 4 \Rightarrow \text{g–weighted < then POT}

proc monCompare(poly p1, int idx1, poly p2, int idx2) { 
  if(mode == 1) \{ // \text{POT}
    if(idx1 < idx2) \{ return(-1); \}
    if(idx1 > idx2) \{ return(1); \}
    if(p1 < p2) \{ return(-1); \}
    if(p1 > p2) \{ return(1); \}
  }
return(0);
}
if(mode == 2) { //TOP
    if(p1 < p2) { return(-1); }
    if(p1 > p2) { return(1); }
    if(idx1 < idx2) { return(-1); }
    if(idx1 > idx2) { return(1); }
    return(0);
}
poly q1 = p1 * gMon[idx1];
poly q2 = p2 * gMon[idx2];
if(mode == 3) { // g-weighted degree then TOP
    int temp = deg(q1) - deg(q2);
    if(temp < 0) { return(-1); }
    if(temp > 0) { return(1); }
    if(p1 < p2) { return(-1); }
    if(p1 > p2) { return(1); }
    if(idx1 < idx2) { return(-1); }
    if(idx1 > idx2) { return(1); }
    return(0);
}
if(mode == 4) { // g-weighted < then POT
    if(q1 < q2) { return(-1); }
    if(q1 > q2) { return(1); }
    if(idx1 < idx2) { return(-1); }
    if(idx1 > idx2) { return(1); }
    if(p1 < p2) { return(-1); }
    if(p1 > p2) { return(1); }
    return(0);
}
if(mode == 5) { // g-weighted < then TOP
    if(q1 < q2) { return(-1); }
    if(q1 > q2) { return(1); }
    if(p1 < p2) { return(-1); }
    if(p1 > p2) { return(1); }
    if(idx1 < idx2) { return(-1); }
    if(idx1 > idx2) { return(1); }
    return(0);
}
if(mode == 6) { // g-weighted degree then POT
    int temp = deg(q1) - deg(q2);
    if(temp < 0) { return(-1); }
    if(temp > 0) { return(1); }
    if(idx1 < idx2) { return(-1); }
    if(idx1 > idx2) { return(1); }
    if(p1 < p2) { return(-1); }
    if(p1 > p2) { return(1); }
    return(0);
}
}
calcBasis accepts a list of generators and a comparison mode. It returns a Groebner basis for the list. JPidx and JPsigmon are lists implementing JP in Figure 1.

```plaintext
proc calcBasis(ideal g, int compMode) {
    system("--ticks-per-sec",100);
    int beginTime = timer;
    createGlobals();
    mode = compMode;
    option(redSB);
    setupVariables(ideal2list(g));
    int j,k;
    int i = size(V);
    for(j = 2; j <= i; j++) {updateJPairs(j); }
	poly u1, v1, u, v; int sigidx;
    while(size(JPidx)>0) {
        u1 = JPsigmon[1]; JPsigmon = delete(JPsigmon,1);
        k = JPidx[1]; JPidx = delete(JPidx,1);
        sigidx = Usigidx[k];
        if(0 != reduce(u1, H[sigidx])) {
            totalSprocessed++;
            v1 = u1/Usigmon[k]*V[k];
            u,v = topReduce(u1, sigidx, v1);
            if(v != 0) {
                i++;
                V[i]= v;
                Usigmon[i] = u;
                Usigidx[i] = sigidx;
                updateJPairs(i);
            } else {
                if (u != 0) {
                    reg0Red++;
                    for(j = 1; j <= size(JPsigmon); j++) {
                        if(sigidx == Usigidx[JPidx[j]] && 0 != JPsigmon[j]/u) {
                            JPsigmon = delete(JPsigmon, j);
                            JPidx = delete(JPidx, j);
                            continue;
                        }
                    }
                    insertH(u, sigidx, u, sigidx, 0);
                } else { sup0Red++;
            }
        }
    }
    ideal answer = makeReduced(V); attrib(answer, "isSB", 1);
    int countH = 0;
    for(j = 1; j <= size(g); j++) { countH = countH + size(H[j]); }
    printf("MODE,%s,%s,%s/%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,\n
#G(aft/bef), \n
...size(H), _reg0Red, _sup0Red, _maxmem") ,
    compMode, timer-beginTime, totalSprocessed, size(answer),
    )

164
```
size(V), countH, reg0Red, sup0Red, memory(2));

return(answer);
}

proc createGlobals() {
    createStatsVariables();
    list Usigmon, Usigidx, V, JPidx, JPsigmon;
    export(Usigmon); export(Usigidx);
    export(V); export(JPidx); export(JPsigmon);
    list H; export(H);
    list g, gMon; export(g); export(gMon);
    int mode; export(mode);
}

proc createStatsVariables() {
    int maxVlength = 0; export(maxVlength);
    int maxSlength = 0; export(maxSlength);
    int totalSprocessed = 0; export(totalSprocessed);
    int reg0Red = 0; export(reg0Red);
    int sup0Red = 0; export(sup0Red);
    int count_reduce = 0; export(count_reduce);
    int count_other_reduce = 0; export(count_other_reduce);
}

proc setupVariables(list gee) {
    Usigmon, Usigidx, JPidx, JPsigmon = list(), list(), list(), list();
    H = list();
    g = gee; V = gee;
    for(int i = 1; i <= size(g); i++) {
        H[i] = ideal();
        gMon[i] = leadmonom(g[i]);
        Usigmon[i] = 1; Usigidx[i] = i;
    }
}

proc insertH(poly mon1, int idx1, poly mon2, int idx2, int same) {
    int idx; poly mon;
    int comp = monCompare(mon1, idx1, mon2, idx2);
    if(comp == 0 && same != 0) { return(); } if(comp == -1) {
        idx = idx2;
        mon = mon2;
    } else {
        idx = idx1;
        mon = mon1;
    }
    H[idx] = interred(H[idx] + mon);
attrib(H[1], "isSB", 1);

insertPairs accepts an index and inserts J-pairs between 1 <= j < i
and i– implementing steps 0 and 3c of Figure 1.
insertPairs calculates all these J-pairs and then inserts them into
This function should be a lot simpler, but we wanted it to look
like F5/F5C's implementation.

proc updateJPairs(int i) {
    poly t1 = leadmonom(V[i]);
    for(int j = 1; j < i; j++) {
        poly t2 = leadmonom(V[j]);
        poly lcm = t1*(t2/gcd(t1,t2));
        poly s1 = lcm/t1*Usigmon[i];
        poly s2 = lcm/t2*Usigmon[j];
        poly sigmon = s1;
        int id1 = 1;
        if(-1 == monCompare(s1, Usigidx[i], s2, Usigidx[j])) {
            sigmon = s2; id1 = j;
        }
        int sigidx = Usigidx[id1];
        insertH(t1*Usigmon[j], Usigidx[j], t2*Usigmon[i],
                Usigidx[i], leadcoef(V[i])==leadcoef(V[j]));
        if(0 != reduce(sigmon, H[sigidx])) {
            int pos, found = binSearch(sigmon, sigidx);
            if(found) {
                JPsigmon[pos] = s1;
                JPidx[pos] = id1;
            } else {
                JPsigmon = insertPosition(JPsigmon, sigmon, pos);
                JPidx = insertPosition(JPidx, id1, pos);
            }
        }
    }
}

binSearch returns ([new] location, found?)

proc binSearch(poly sigmon, int sigidx) {
    if(size(JPsigmon) == 0) { return(1,0); }
    if(-1 == monCompare(sigmon, sigidx, JPsigmon[1], Usigidx[JPidx[1]])) {
        return(1,0); }
    int l = 1;
    int r = size(JPsigmon);
    int k;
    int compare = monCompare(sigmon, sigidx, JPsigmon[r],
Usigidx[JPidx[r]]);

if(compare == 1) { return(r+1,0); }
if(compare == 0) { return(r,1); }
while(1 < r) { // JPsig[l] <= sig < JPsig[r]
k = (1+r) / 2;
    int compare = monCompare(sigmon, sigidx, JPsigmon[k],
                           Usigidx[JPidx[k]]);
    if(compare == 0) { return(k,1); }
    if(compare == 1) { l = k + 1; } // go right
    else { r = k; } // go left
}
return(1,0);

// findReductor finds a (u1,v1) pair that will top-reduce (u,v)
// Called by topReduce to simply find the index of a top-reducer.
proc findReductor(poly u, int sigidx, poly v) {
    if(v == 0) { return(0); }
    int superFlag = 0; // becomes −1 if super top-reducible
    poly t1, s1;
    for(int j = 1; j < size(V); j++) {
        t1 = leadmonom(v)/leadmonom(V[j]);
        if(0 != t1) {
            s1 = t1*Usigmon[j];
            int comp = monCompare(s1, Usigidx[j], u, sigidx);
            if(-1 == comp) { return(j); }
            if(0 == comp) { superFlag = -1; }
        }
    }
    return(superFlag);
}

// topReduce performs step 2 in Figure 1.
// topReduce calls findReductor, does the reduction and repeats.
proc topReduce(poly u, int sigidx, poly v) {
    int j; poly ct1, v1;
    while(1) {
        j = findReductor(u, sigidx, v);
        if(j==-1) { return(0,0); } // super top-reducible
        if(j==0) { return(u,v); } // not top-reducible
        ct1 = lead(v)/lead(V[j]);
        v = v-ct1*V[j];
    }
}

// makeReduced makes a GB into a reduced Groebner basis
proc makeReduced(list basis) {
    int i; ideal j;
for (i = 1; i <= size(basis); i++) { j = j + basis[i]; }  
return(interged(j));

proc ideal2list(ideal input) {
    list output;
    int i1, i2 = 1,1;
    for (i1 = 1; i1 <= size(input); i1++) {
        if (input[i2] == 0) {
            i2++;
            continue;
        }
        output[i1] = input[i2];
        i2++;
    }
    return(output);
}

Listing 27: GVW as in figure 3.1

C.3 Examples Used in Singular

option(redSB); // cause intered to give reduced groebner basis
option(noredefine); // quit saying: // ** redefining this–and–that // < "f5_library.sing"

proc prep1(int p) {
    ring R = p,(x,y,u,v),lp;
    list iList = x^p-x, y^p-y, u^p-u, v^p-v, xy+uv-1;
    ideal i = x^p-x, y^p-y, u^p-u, v^p-v, xy+uv-1;
    export(p);
    export(R);
    export(iList);
    export(i);
}

proc prep2() { // example from CLO bottom of page 75.  
    //Should add at least x^2 to generators
    ring R = 0,(x,y),Dp;
    list iList = x3-2xy, x2y-2y2+x;
    ideal i = x3-2xy, x2y-2y2+x;
    export(R);
    export(iList);
    export(i);
}

proc prep3(int n_even) {
    int n_over_two = n_even / 2;
    ring R = 2,(x(1..(n_over_two*2))),lp; // default: lp; // lex
    list iList; ideal i;
    poly p_big, p1, p2;
    int idx = 1;
}
for(int j = 1; j <= n_over_two; j++) {
    p_big = p_big + x(2*j-1)*x(2*j);
    p1 = x(2*j-1)*2-x(2*j-1);
    p2 = x(2*j)*2-x(2*j);
    i = (i+p1)+p2;
    iList[idx] = p1;
    iList[idx+1] = p2;
    idx = idx + 2;
    // iList = insert(insert(iList,p1),p2);
}

iList[idx] = p_big;
// iList = insert(iList,p_big);
i = i + p_big;
export(R);
export(iList);

proc forceError(int vars) {
    int j,k,numTerms, 1, numFactors, idx;
    while(1) {
        prep3(vars);
        poly p, currentTerm;
        idx = vars + 2;
        for(j = 0; j < 3; j++) { // add a generator
            p = 0;
            numTerms = random(1,vars);
            for(k = 0; k < numTerms; k++) { // construct a monomial
                currentTerm = 1;
                for(l = 0; l < vars; l++){
                    if(random(0,1)) {
                        currentTerm = currentTerm * x(1); }
                }
                p = p + currentTerm;
            }
            if(p!=0) { i[idx+j]=p; } else { continue; }
            if(!validate()) {
                printf("Found counterexample. See 1.");
                return();
            }
        }
    }
}

proc other() {
    printf("Other functions available are:");
    printf("katsura7, katsura8, schransTroost, f633, f744, cyclic_n(int) ");
}

// proctype bench_n() {


169
list n = [29, 53, 127, 227];
for(int i = 1; i <= size(n); i++) {
    printf("pre1(%s)", n[i]);
    prep1(n[i]);
    benchmark();
}

list n2 = [10, 12, 14, 16, 18, 20];
for(i = 1; i <= size(n2); i++) {
    printf("pre3(%s)", n2[i]);
    prep3(n2[i]);
    benchmark();
}

proc prof() {
    list n = [29, 53, 127, 227];
    for(int j = 1; j <= size(n); j++) {
        printf("pre1(%s)", n[j]);
        prep1(n[j]);
        ideal k = calcBasis(i, 1);
    }
}

proc prof2() {
    list n2 = [10, 12, 14, 16, 18, 20];
    for(int j = 1; j <= size(n2); j++) {
        printf("pre3(%s)", n2[j]);
        prep3(n2[j]);
        ideal k = calcBasis(i, 1);
    }
}

proc validateFast() {
    printf("TIMES ARE IN HUNDREDS OF A SECOND ! ! ! ! ! ! ! ! ! !");
    katsura5();
    validate6();
    katsura6();
    validate6();
    katsura7();
    validate6();
    katsura8();
    validate6();
schrans_troost(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Schrans-Troost '\n"); validate6();
f633(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'F633 '\n"); validate6();
cyclic_n(6); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Cyclic (n=6) '\n"); validate6();
cyclic_n(7); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Cyclic (n=7) '\n"); validate6();
}

proc benchmarkFast() {
  katsura5(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Katsura5 '\n"); benchmark();
  katsura6(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Katsura6 '\n"); benchmark();
  katsura7(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Katsura7 '\n"); benchmark();
  katsura8(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Katsura8 '\n"); benchmark();
  schrans_troost(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Schrans-Troost '\n"); benchmark();
  f633(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'F633 '\n"); benchmark();
  cyclic_n(6); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Cyclic (n=6) '\n"); benchmark();
  cyclic_n(7); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Cyclic (n=7) '\n"); benchmark();
  //bench_n();
}

proc benchmarkSlow() {
  f744(); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'F744 '\n"); validate();
  cyclic_n(8); printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\nFor 'Cyclic (n=8) '\n"); validate();
}

// //////////////////////////////////////////////////////////////////

}
proc benchmark (int mode) {
    option (noredefine);
    ideal j;
    printf("Running\nF5\nF5\nF5\nF5\nF5\nF5\nF5\n−−−−−−−−−−−−−");
    j = basis(i);
    printf("Running\nF5C\nF5C\nF5C\nF5C\nF5C\nF5C\nF5C\n−−−−−−−−−−−−−−");
    j = basis_c(i);
    printf("Running\nG1\nG1\nG1\nG1\nG1\nG1\nG1\n−−−−−−−−−−−−−");
    j = calcBasis(i, mode);
    // validate();
}

proc g1benchmark (int mode) {
    option (noredefine);
    ideal j;
    printf("Running\nG1\nG1\nG1\nG1\nG1\nG1\nG1\n−−−−−−−−−−−−−");
    j = calcBasis(i, mode);
}

proc f5cbenchmark () {
    option (noredefine);
    ideal j;
    printf("Running\nF5C\nF5C\nF5C\nF5C\nF5C\nF5C\nF5C\n−−−−−−−−−−−−−−");
    j = basis_c(i);
    printf("Running\nF5\nF5\nF5\nF5\nF5\nF5\nF5\n−−−−−−−−−−−−−");
    j = basis(i);
}

Examples from Eder and Perry

proc katsura5 () {

    ring R = 7583,(x,y,z,t,u,v,h),dp;

    poly p1= 2*x^2 + 2*y^2 + 2*z^2 + 2*t^2 + 2*u^2 + v^2 - vh;
    poly p2= xy + yz + 2*zt + 2*tu + 2*uv + uh;
    poly p3= 2*xz + 2*y^t + 2*zu + u^2 + 2*tv - th;
    poly p4= 2*xt + 2*yu + 2*tu + 2*zv - zh;
    poly p5= t^2 + 2*xy + 2*yv + 2*zv - yh;
    poly p6= 2*x + 2*y + 2*z + 2*t + 2*u + v - h;

    ideal i= p1,p2,p3,p4,p5,p6;
    export (R);
    export (i);

    return ();
}
proc lexKatsura5()
{
    ring R = 7583,(x,y,z,t,u,v,h),lp;
    poly p1= 2*x2 + 2*y2 + 2*z2 + 2*t2 + 2*u2 + v2 - vh;
    poly p2= xy + yz + 2*zt + 2*tu + 2*uv + uh;
    poly p3= 2*xz + 2*yt + 2*zu + u2 + 2*tv - th;
    poly p4= 2*xt + 2*yu + 2*tu + 2*zv - zh;
    poly p5= t2 + 2*xv + 2*yv + 2*zv - yh;
    poly p6= 2*x + 2*y + 2*z + 2*t + 2*u + v - h;
    ideal i= p1 , p2 , p3 , p4 , p5 , p6 , p7;
    export(R);
    export(i);
    return();
}

proc katsura6()
{
    ring R = 7583,(x(1..7),h),dp;
    poly p1= x(1) + 2*x(2) + 2*x(3) + 2*x(4) + 2*x(5) + 2*x(6) +
        2*x(7) - h;
    poly p2= 2*x(3)*x(4) + 2*x(2)*x(5) + 2*x(1)*x(6) + 2*x(2)*x(7) - x(6)*h;
    poly p3= x(3)^2 + 2*x(2)*x(4) + 2*x(1)*x(5) + 2*x(2)*x(6) +
        2*x(3)*x(7) - x(5)*h;
    poly p4= 2*x(2)*x(3) + 2*x(1)*x(4) + 2*x(2)*x(5) + 2*x(3)*x(6) +
        2*x(4)*x(7) - x(4)*h;
    poly p5= x(2)^2 + 2*x(1)*x(3) + 2*x(2)*x(4) + 2*x(3)*x(5) +
        2*x(4)*x(6) + 2*x(5)*x(7) - x(3)*h;
    poly p6= 2*x(1)*x(2) + 2*x(2)*x(3) + 2*x(3)*x(4) + 2*x(4)*x(5) +
        2*x(5)*x(6) + 2*x(6)*x(7) - x(2)*h;
    poly p7= x(1)^2 + 2*x(2)^2 + 2*x(3)^2 + 2*x(4)^2 + 2*x(5)^2 +
        2*x(6)^2 + 2*x(7)^2 - x(1)*h;
    ideal i= p1 , p2 , p3 , p4 , p5 , p6 , p7;
    export(R);
    export(i);
    return();
}
proc katsura7()
{
    ring R = 7583,(x(1..8),h),dp;
    poly p1= x(1)^2 + 2*x(2)^2 + 2*x(3)^2 + 2*x(4)^2 + 2*x(5)^2 + 2*x(6)^2 + 2*x(7)^2 + 2*x(8)^2 - x(1)*h;
    poly p2= 2*x(1)*x(2) + 2*x(2)*x(3) + 2*x(3)*x(4) + 2*x(4)*x(5) + 2*x(5)*x(6) + 2*x(6)*x(7) + 2*x(7)*x(8) - x(2)*h;
    poly p3= x(2)^2 + 2*x(1)*x(3) + 2*x(2)*x(4) + 2*x(3)*x(5) + 2*x(4)*x(6) + 2*x(5)*x(7) + 2*x(6)*x(8) - x(3)*h;
    poly p4= 2*x(2)*x(3) + 2*x(1)*x(4) + 2*x(2)*x(5) + 2*x(3)*x(6) + 2*x(4)*x(7) + 2*x(5)*x(8) - x(4)*h;
    poly p5= x(3)^2 + 2*x(2)*x(4) + 2*x(1)*x(5) + 2*x(2)*x(6) + 2*x(3)*x(7) + 2*x(4)*x(8) - x(5)*h;
    poly p6= 2*x(3)*x(4) + 2*x(2)*x(5) + 2*x(1)*x(6) + 2*x(2)*x(7) + 2*x(3)*x(8) - x(6)*h;
    poly p7= x(4)^2 + 2*x(3)*x(5) + 2*x(2)*x(6) + 2*x(1)*x(7) + 2*x(2)*x(8) - x(7)*h;
    poly p8= x(5)^2 + 2*x(4)*x(5) + 2*x(3)*x(6) + 2*x(4)*x(7) + 2*x(5)*x(8) - h;
    list iList= p1,p2,p3,p4,p5,p6,p7,p8;
    ideal i= p1,p2,p3,p4,p5,p6,p7,p8;
    export(R);
    export(i);
    export(iList);
}

proc katsura8()
{
    ring R = 7583,(x(1..9),h),dp;
    poly p1= x(1)^2 + 2*x(2)^2 + 2*x(3)^2 + 2*x(4)^2 + 2*x(5)^2 + 2*x(6)^2 + 2*x(7)^2 + 2*x(8)^2 + 2*x(9)^2 - x(1)*h;
    poly p2= 2*x(1)*x(2) + 2*x(2)*x(3) + 2*x(3)*x(4) + 2*x(4)*x(5) + 2*x(5)*x(6) + 2*x(6)*x(7) + 2*x(7)*x(8) + 2*x(8)*x(9) - x(2)*h;
    poly p3= x(2)^2 + 2*x(1)*x(3) + 2*x(2)*x(4) + 2*x(3)*x(5) + 2*x(4)*x(6) + 2*x(5)*x(7) + 2*x(6)*x(8) + 2*x(7)*x(9) - x(3)*h;
    poly p4= 2*x(2)*x(3) + 2*x(1)*x(4) + 2*x(2)*x(5) + 2*x(3)*x(6) + 2*x(4)*x(7) + 2*x(5)*x(8) + 2*x(6)*x(9) - x(4)*h;

polynomial p5 = \((x(3)^2 + 2*x(2)*x(4) + 2*x(1)*x(5) + 2*x(2)*x(6) + 2*x(3)*x(7) + 2*x(4)*x(8) + 2*x(5)*x(9) - x(5))*h;
polynomial p6 = 2*x(3)*x(4) + 2*x(2)*x(5) + 2*x(1)*x(6) + 2*x(2)*x(7) + 2*x(3)*x(8) + 2*x(4)*x(9) - x(6)*h;
polynomial p7 = (x(4)^2 + 2*x(3)*x(5) + 2*x(2)*x(6) + 2*x(1)*x(7) + 2*x(2)*x(8) + 2*x(3)*x(9) - x(7))*h;
polynomial p8 = 2*x(4)*x(5) + 2*x(3)*x(6) + 2*x(2)*x(7) + 2*x(1)*x(8) + 2*x(2)*x(9) - x(8)*h;
polynomial p9 = x(1) + 2*x(2) + 2*x(3) + 2*x(4) + 2*x(5) + 2*x(6) + 2*x(7) + 2*x(8) + 2*x(9) - h;

ideal i = p1, p2, p3, p4, p5, p6, p7, p8, p9;
list iList = p1, p2, p3, p4, p5, p6, p7, p8, p9;

export(R);
export(i);
export(iList);

}

proc schrans_troost() {

ring R = 7583,(x(1..8), h),dp;
// assuming the first + in p4 belongs there
polynomial p1 = 8*x(1)^2 + 8*x(1)*x(2) + 8*x(1)*x(3) - 8*x(2)*x(3) + 2*x(1)*x(4) + 2*x(1)*x(5) + 2*x(1)*x(6) - 2*x(5)*x(6) + 2*x(1)*x(7) - 2*x(4)*x(7) - x(1)*h;
polynomial p2 = 8*x(1)*x(2) + 8*x(2)^2 - 8*x(1)*x(3) + 8*x(2)*x(3) + 2*x(2)*x(4) + 2*x(2)*x(5) + 2*x(2)*x(6) - 2*x(4)*x(6) + 2*x(2)*x(7) - 2*x(5)*x(7) - x(2)*h;
polynomial p3 = -8*x(1)*x(2) + 8*x(1)*x(3) + 8*x(2)*x(3) + 8*x(3)^2 + 2*x(3)*x(4) + 2*x(3)*x(5) + 2*x(4)*x(5) + 2*x(3)*x(6) + 2*x(3)*x(7) - 2*x(6)*x(7) - x(3)*h;
polynomial p4 = 2*x(1)*x(4) + 2*x(2)*x(4) + 2*x(3)*x(4) + 8*x(4)^2 - 2*x(3)*x(5) + 8*x(4)*x(5) - 2*x(2)*x(6) + 2*x(4)*x(6) + 2*x(1)*x(7) + 2*x(4)*x(7) + 6*x(4)*x(8) - 6*x(5)*x(8) - x(4)*h;
polynomial p5 = -2*x(1)*x(4) - 2*x(2)*x(5) - 2*x(3)*x(6) + 2*x(1)*x(7) + 2*x(2)*x(7) + 2*x(3)*x(7) + 2*x(4)*x(7) + 2*x(5)*x(7) + 8*x(6)*x(7) + 8*x(7)^2 - 6*x(6)*x(8) + 6*x(7)*x(8) - x(7)*h;
polynomial p6 = -2*x(2)*x(4) - 2*x(1)*x(5) + 2*x(1)*x(6) + 2*x(2)*x(6) + 2*x(3)*x(6) + 2*x(4)*x(6) + 2*x(5)*x(6) + 8*x(6)^2 - 2*x(3)*x(7) + 8*x(6)*x(7) + 6*x(6)*x(8) - 6*x(7)*x(8) - x(6)*h;
polynomial p7 = -2*x(3)*x(4) + 2*x(1)*x(5) + 2*x(2)*x(5) + 2*x(3)*x(5) + 8*x(4)*x(5) + 8*x(5)^2 - 2*x(1)*x(6) + 2*x(5)*x(6) - 2*x(2)*x(7) + 2*x(5)*x(7) - 6*x(4)*x(8) + 6*x(5)*x(8) - x(5)*h;
polynomial p8 = -6*x(4)*x(5) - 6*x(6)*x(7) + 6*x(4)*x(8) + 6*x(5)*x(8) + 6*x(6)*x(8) + 8*x(8)^2 - x(8)*h;

ideal i = p1, p2, p3, p4, p5, p6, p7, p8;
list iList = p1, p2, p3, p4, p5, p6, p7, p8;

175
proc f633()
{
    ring R = 7583,(U(6),U(5),U(4),U(3),u(6),u(5),u(4),u(3),u(2),h),dp;
    poly p1= 2*u(6) + 2*u(5) + 2*u(4) + 2*u(3) + 2*u(2) + h;
    poly p2= 2*U(6) + 2*U(5) + 2*U(4) + 2*U(3) + 2*U(2) + h;
    poly p3= 4*U(5)*u(6) + 4*U(4)*u(6) + 4*U(3)*u(6) + 4*U(2)*u(6) - 4*U(6)*u(5) + 4*U(4)*u(5) + 4*U(3)*u(5) + 4*U(2)*u(5) - 4*U(6)*u(4) + 4*U(4)*u(4) + 4*U(3)*u(4) + 4*U(2)*u(4) - 4*U(6)*u(3) + 4*U(5)*u(3) - 4*U(4)*u(3) + 4*U(2)*u(3) - 4*U(6)*u(2) + 4*U(5)*u(2) - 4*U(4)*u(2) - 4*U(3)*u(2) + 2*u(6)*h + 2*u(5)*h + 2*u(4)*h + 2*u(3)*h + 2*u(2)*h + h^2;
    poly p4= -4*U(5)*u(6) - 4*U(4)*u(6) - 4*U(3)*u(6) - 4*U(2)*u(6) + 4*U(6)*u(5) - 4*U(4)*u(5) - 4*U(3)*u(5) - 4*U(2)*u(5) + 4*U(6)*u(4) + 4*U(5)*u(4) - 4*U(3)*u(4) - 4*U(2)*u(4) + 4*U(6)*u(3) + 4*U(5)*u(3) + 4*U(4)*u(3) - 4*U(2)*u(3) + 4*U(6)*u(2) + 4*U(5)*u(2) + 4*U(4)*u(2) + 4*U(3)*u(2) + 2*u(6)*h + 2*u(5)*h + 2*u(4)*h + 2*u(3)*h + 2*u(2)*h + h^2;
    poly p5= U(2)*u(2) - h^2;
    poly p6= U(3)*u(3) - h^2;
    poly p7= U(4)*u(4) - h^2;
    poly p8= U(5)*u(5) - h^2;
    poly p9= U(6)*u(6) - h^2;

    ideal i= p1,p2,p3,p4,p5,p6,p7,p8,p9;
    list iList= p1,p2,p3,p4,p5,p6,p7,p8,p9;

    export(R);
    export(i);
    export(iList);
}

proc f744()
{
    ring R = 7583,(U(7..2),u(7..2),h),dp;
    poly p1= 2*u(7) + 2*u(6) + 2*u(5) + 2*u(4) + 2*u(3) + 2*u(2) + h;
    poly p2= 2*U(7) + 2*U(6) + 2*U(5) + 2*U(4) + 2*U(3) + 2*U(2) + h;
    poly p3= 8*U(6)*u(7) + 8*U(5)*u(7) + 8*U(4)*u(7) + 8*U(3)*u(7) + 8*U(2)*u(7) + 8*U(6)*u(6) + 8*U(5)*u(6) + 8*U(4)*u(6) +
\[
8 \ast U(3) \ast u(6) + 8 \ast U(2) \ast u(6) + 8 \ast U(5) \ast u(5) + 8 \ast U(4) \ast u(5) + \\
8 \ast U(2) \ast u(5) + 8 \ast U(3) \ast u(5) + 8 \ast U(2) \ast u(2) - 17 \ast h^2; \\
poly p_4 = 16 \ast U(5) \ast U(3) \ast u(4) + 16 \ast U(5) \ast U(2) \ast u(4) + 16 \ast U(5) \ast U(2) \ast u(3) \\
+ 16 \ast U(4) \ast U(2) \ast u(3) + 8 \ast U(5) \ast u(4) \ast h + 8 \ast U(5) \ast u(3) \ast h \\
+ 8 \ast U(4) \ast u(3) \ast h + 8 \ast U(5) \ast u(2) \ast h + 8 \ast U(4) \ast u(2) \ast h + 8 \ast U(3) \ast u(2) \ast h \\
+ 18 \ast U(5) \ast h \ast 2 + 18 \ast U(4) \ast h \ast 2 + 18 \ast U(3) \ast h \ast 2 + 18 \ast U(2) \ast h \ast 2 + 11 \ast h \ast 3; \\
poly p_5 = 16 \ast U(4) \ast U(5) \ast u(3) + 16 \ast U(4) \ast U(5) \ast u(2) + 16 \ast U(3) \ast u(5) \ast u(2) \\
+ 16 \ast U(3) \ast u(4) \ast u(2) + 8 \ast U(4) \ast u(5) \ast h + 8 \ast U(3) \ast u(5) \ast h \\
+ 8 \ast U(2) \ast u(3) \ast h + 8 \ast U(3) \ast u(4) \ast h + 8 \ast U(2) \ast u(4) \ast h + 8 \ast U(2) \ast u(3) \ast h \\
+ 18 \ast u(5) \ast h \ast 2 + 18 \ast u(4) \ast h \ast 2 + 18 \ast u(3) \ast h \ast 2 + 18 \ast u(2) \ast h \ast 2 + 11 \ast h \ast 3; \\
poly p_6 = U(2) \ast u(2) - h \ast 2; \\
poly p_7 = U(3) \ast u(3) - h \ast 2; \\
poly p_8 = U(4) \ast u(4) - h \ast 2; \\
poly p_9 = U(5) \ast u(5) - h \ast 2; \\
poly p_{10} = U(6) \ast u(6) - h \ast 2; \\
poly p_{11} = U(7) \ast u(7) - h \ast 2; \\
\]

ideal i = p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}; \\
list iList = p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}; \\

export(R); \\
export(i); \\
export(iList); 

proc find_counterexample() { 
  list p = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 39, 41, 43, 47, 49; 
  list n = 1, 2, 3, 4, 5, 6; 
  int i,j; 
  for(i = 1; i <= size(n); i++) { 
    for(j = 1; j <= size(p); j++) { 
      cyclic_np(n[i], p[j]); 
      if(!validate()) { 
        printf("Found a counterexample, n=%s, p=%s", n[i], p[j]); 
        return(); 
      } 
    } 
  } 
}

proc cyclic_n(int n) { cyclic_np(n,7583); } 
proc cyclic_np(int n, int p) 
{ 
  ring R = p,(x(1..(n)),h),dp; // default characteristic is 7583 
  int 1, j, k, ctr; 
  ideal i, base_set; 
  poly facs, addem; 
  i = 0; 
}
for (ctr = 1; ctr <= n; ctr++) {
    base_set[ctr] = x(ctr);
}
for (l = 1; l < n; l++) {
    addem = 0;
    for (j = 1; j <= n; j++) {
        facs = 1;
        for (k = j; k <= l + j - 1; k++) {
            if (k <= n) {
                facs = facs * x(k);
            } else {
                facs = facs * x(k - n);
            }
        }
        addem = addem + facs;
    }
    /* l[i] = addem; */
    i = i + addem;
}

facs = 1;
for (l = 1; l <= n; l++) {
    facs = facs * x(1);
} /* l[n] = facs - 1; */
i = i + (facs - h`n);
export(R);
export(i);
list IPList;
for (k = 1; k <= size(i); k++) {
    IPList = insert(IPList, i[k], size(IPList));
}
export(IList);

// //////////////////////////////////////////////////////////////////
// //////////////////////////////////////////////////////////////////
proc validate6() {
    for (int k = 1; k <= 5; k++) {
        validate(k);
    }
} proc validate(int mode) {
    validate_option(1, mode);
}
proc validate_option(int print_results, int mode) {
    ideal gIdeal = calcBasis(i, mode);
    ideal jIdeal = groebner(i);
    int answer = validate2ideals(gIdeal, jIdeal);
    if (print_results) {
        if (answer) {
            // printf(">>>>>>>Everything matches!!!!!");
        } else {
            printf("<<<<<<<MISMATCH");
        }
    }
} // return(answer);

178
proc validate2ideals(ideal i1, ideal i2) {
    option(redSB);
    int i, flag = 0, 1;
    if(size(i1) != size(i2)) {
        printf("basis size mismatch: %s vs %s", size(i1), size(i2));
        flag = 0;
    }
    for(i = 1; i <= size(i1); i++) {
        if(i1[i] != i2[i]) {
            printf("basis mismatch at %s", i);
            return(0);
        }
    }
    return(flag);
}

Listing 28: Examples used in Singular
Bibliography


