Grobner Bases Writeup

Riyaz Ahuja, Dennis Chen, Rohan Jain

November 29, 2025

Table of Contents

1 The Basic Theory

1.1 Introduction

Fix your field k and consider the ring $R = k[x_1, x_2, ..., x_n]$. Remember that by Hilbert Basissatz, any ideal in this ring is finitely generated. That is, for an ideal I, we can express it as

$$I = (f_1, ..., f_n) = \{a_1 f_1 + \cdots + a_n f_n : a_i \in R\}.$$

As an example, let $f = xy$ and $g = xy - z$ in $k[x, y, z]$ and define $I = (f, g)$. Someone may ask where $z \in I$ or not, and we can respond by saying

$$z = f - g$$

and therefore $z \in I$. But what an expression like z^2? Is that in I as well? This makes us define our problem.

Ideal Membership Problem: Given an ideal $I = (f_1, ..., f_n) \subset R$ and a polynomial $f \in R$, is $f \in I$? If it is, what's the linear combination of f_i that is equal to f?

1.2 Preliminary Definitions

Theorem 1.1 (Monomial ordering): Let $\alpha = [\alpha_1, \alpha_2, ..., \alpha_n]$ be a multi-index, meaning

$$x^\alpha = x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}.$$

There is a total order $\prec$ on R satisfying:
1) $x^\alpha \prec x^\beta \implies x^{\alpha+\gamma} \prec x^{\beta+\gamma}$ for multi-indices α, β, γ.
2) $1 \prec x^\alpha$ for all $\alpha \in \mathbb{N}^n \setminus \{0\}$.

The previous theorem induces a "degree lexicographic order". In simple terms, if we have $x > y > z$ lexicographically, suppose

$$f = x^3 + z^7 + x^2y + yz^2 + y^2z + y + x.$$

Writing out their multi-indices, we get

$$(3,0,0), (0,0,7), (2,1,0), (0,1,2), (0,2,1), (0,1,0), (1,0,0).$$

We want to sort these by index from left to right, meaning the right order is

$$(3,0,0), (2,1,0), (1,0,0), (0,2,1), (0,1,2), (0,1,0), (0,0,7).$$

So we get that f should be written as

$$f = x^3 + x^2y + x + y^2z + yz^2 + y + z^6.$$

Definition 1.2: Fix a monomial order on $k[x_1, ..., x_n]$ and let $f \in k[x_1, ..., x_n]$ written as

$$f = c_1 X^{\alpha_1} + \cdots + c_r X^{\alpha_r}$$

where each α_i is a multiindex such that $X^{\alpha_1} > \cdots > X^{\alpha_r}$ with respect to our monomial ordering. We define:

- $\mathrm{LM}(f) = X^{\alpha_1}$ (the leading monomial)
- $\mathrm{LC}(f) = c_1$ (the leading coefficient)
- $\mathrm{LT}(f) = c_1 X^{\alpha_1} = \mathrm{LC}(f) \cdot \mathrm{LM}(f)$ (the leading term)

Example 1.3: Let $f = 42x^3 + 5y^2 + z$. Then,

$$\mathrm{LM}(f) = x^3, \quad \mathrm{LC}(f) = 42, \quad \mathrm{LT}(f) = 42x^3.$$

With these definitions, we are now equipped to tackle the idea of polynomial reduction.

Definition 1.4: Given $f, g, h \in R$ with $g \neq 0$, we can say f reduces to h modulo g if $\mathrm{LM}(g)$ divides a non-zero term X of f and $h = f - \frac{X}{\mathrm{LT}(g)} \cdot g$.

Example 1.5:

- xyz reduces to y^2 modulo $xz - y$ because

$$xyz - y \cdot (xz - y) = y^2.$$

- $x^2z + 3y^2$ reduces to $-x^3 + 3y^2 - 7xy$ modulo $x^2 + xz + 7y$ because

$$x^2z + 3y^2 - x \cdot (x^2 + xz + 7y) = -x^3 + 3y^2 - 7xy.$$

We can then extend this definition to motivate the idea of a Grobner Basis.

Definition 1.6: Given $f, h \in R$ and a set $G = \{g_1, ..., g_n\} \subset R$ of nonzero polynomials, we can say f reduces to h modulo G if there exists a sequence of indices $i_1, ..., i_\ell \in \{1, ..., n\}$ and polynomials $h_1, ...h_{\ell-1}$ such that f reduces to h_1 modulo g_{i_1}, h_1 reduces to h_2 modulo g_{i-2}, ..., $h_{\ell-1}$ reduces to h modulo g_{i_ℓ}.

Definition 1.7: A polynomial f is called reduced with respect to G if it cannot be reduced modulo G. That is, no term of f is divisible by $\mathrm{LM}(g_i)$ for any i.

With this definition, note that such a decomposition G will not always be unique. Regardless, this gives way to the definition of a Grobner basis.

Definition 1.8: A set $G = \{g_1, ..., g_n\}$ of non-zero polynomials is a Grobner basis for the ideal $I = (f_1, ..., f_n)$ if for all non-zero $f \in I$, we have that $\mathrm{LM}(g_i) \mid \mathrm{LM}(f)$ for some $g_i \in G$.

1.3 Gaussian Elimination v2

The motivation for Grobner bases comes from wanting to solve systems of polynomials efficiently. Consider the example below

$$\begin{cases} f := xy^2 + 4 = 0 \\ g := x^2y - 8 = 0 \end{cases}.$$

We want to "eliminate" the first term as we did in classic Gaussian Elimination. This introduces the idea of an S-polynomial, denoted $S(f, g)$. In this case, we get

$$S(f, g) = xf - yg = 4x + 8y.$$

By solving and checking with our equations, we get $(x, y) = (-1, 2)$. With this motivation, we define an S-polynomial formally.

Defintion: Given non-zero $f, g \in R$, the S-polynomial is defined as

$$S(f, g) = \frac{L}{\mathrm{LT}(f)} f - \frac{L}{\mathrm{LT}(g)} g$$

where $L = \mathrm{lcm}(\mathrm{LM}(f), \mathrm{LM}(g))$.

1.4 Finding the Grobner Basis

We introduce Buchberger's Algorithm. Let $F = \{f_1, ..., f_m\}$ be a set of polynomials.

1) $G := F$. Construct an initial set of pairs to examine:

$$P := \{(f, g) \mid f, g \in G, f \neq g\}.$$

2) While P is non-empty,
   a) Select an remove a pair $(f, g) \in P$.
   b) Compute $L := \mathrm{lcm}(\mathrm{LM}(f), \mathrm{LM}(g))$.
   c) Compute $S(f, g) = \frac{L}{\mathrm{LT}(f)} f - \frac{L}{\mathrm{LT}(g)} g$.
   d) Reduce $S(f, g)$ with respect to G with the following reduction process:
      - While there is a nonzero term T in $S(f, g)$ for which there exists an $h \in G$ with $\mathrm{LM}(h) \mid T$, write $T = cX$ (with X monomial and c coefficient) and replace

$$S(f, g) := S(f, g) - \frac{c}{\mathrm{LC}(h)} \cdot \frac{X}{\mathrm{LM}(h)} h.$$

      Denote the fully reduced polynomial as S'.

e) If S' is nonzero, add it to G. And for every h in G, add the pair (S', h) to P.

3) When no new S-polynomials reduce to a nonzero remainder, (i.e. when P is empty), the current set G is the Grobner basis we are looking for.

1.4.1 Example of Buchberger's Algorithm

Let $f_1 = x^2 - y$ and $f_2 = xy - 1$. Our goal is to compute the Grobner basis for the ideal $I = \langle f_1, f_2 \rangle$.

We start by computing the S polynomial

$$S(f_1, f_2) = yf_1 - xf_2 = x - y^2.$$

Since $x - y^2$ cannot be reduced by f_1 or f_2, we add it to our basis:

$$f_3 := x - y^2.$$

So now we have $G = \{f_1 = x^2 - y, f_2 = xy - 1, f_3 = x - y^2\}$. Now we want to calculate $S(f_1, f_3)$ and $S(f_2, f_3)$. Firstly,

$$S(f_1, f_3) = f_1 - xf_3 = xy^2 - y.$$

However, we can see that $S(f_1, f_3) = yf_2$, so we don't add it.

$$S(f_2, f_3) = f_2 - yf_3 = y^3 - 1.$$

If we take f_3 and replace $x = y^2$ into this polynomial, we get that

$$y^3 - 1 = xy - 1 = f_2.$$

As such, we don't want to add this polynomial to our basis either.

As we have considered every S polynomial of every pair of polynomials in our basis, we are done and we have that our Grobner basis is

$$G = \{f_1 = x^2 - y, f_2 = xy - 1, f_3 = x - y^2\}.$$

1.5 Unique Representatives

Theorem 1.9: A basis $\{g_1, ..., g_n\}$ of I is a Grobner basis iff every element of $A(X) = k[\boldsymbol{x}]/I$ has exactly one representative with none of its terms divisible by any $\mathrm{LM}(g_i)$.

As a note, even though 0 is divisible by every polynomial, we treat it as having no terms, so the condition is vacuously true for 0.

Proof of Theorem 1.9: Follows from the definition of a Grobner Basis.

- Grobner Basis $\implies$ Unique Representative: For the sake of contradiction suppose some polynomial has two representatives r_1 and r_2. But then $r_1 - r_2 \in I$, and the leading term from $r_1 + (r_2 - r_1)$ comes from I.
- Unique Representative $\implies$ Grobner Basis: The unique representative of 0 is 0.

□

To compute the representative, just use the division algorithm the exact same way as computing the Grobner basis.

2 Applications of Grobner Bases

2.1 Ideal Membership Problem

A big use of Grobner Bases for mathematicians is the *ideal membership problem.*

Solution: Compute a Grobner Basis for I and the representative of f. If it is 0, then $f \in I$; otherwise, we know for sure that $f \notin I$. □

2.2 Radicals

Suppose $I = \langle f_1, ..., f_n \rangle$ and our Grobner Basis is G.

We overload notation a little and define $\text{LM}(I)$ to be the ideal of I generated by the leading monomials $\text{LM}(f)$ for all $f \in I$.

1) Radical Membership Problem: Recall that $f \in \sqrt{I} \iff 1 \in \langle f_1, ..., f_n, 1 - yf \rangle$.
2) Is I radical: It is a fact that $\text{LM}(G)$ generates $\text{LM}(I)$ and G being square free implies I is radical (since G generates I).

2.3 Computing the Hilbert Polynomial

We can also use Grobner Bases to compute the Hilbert Polynomial of an ideal.

Theorem 2.1: Precisely, suppose that I is a homogeneous ideal; then

$$\chi_I = \chi_{\text{LM}(I)},$$

where χ_I denotes the Hilbert Polynomial of the ideal I.

The proof requires the notion of flatness, which we will not develop in this paper.

Why is this useful? It turns out that understanding the Hilbert Polynomial of an ideal generated by monomials is much easier than understanding the Hilbert Polynomial of a general ideal.

To compute $h_{\text{LM}}(I)$, all we need to do is find the number of monomials of degree d not divisible by any of the generators of $\text{LM}(I)$. (More precisely, these quantities are the same.)

2.4 Elimination

We consider the following exercise as motivation.

Exercise 2.2 (4.15 from Chris' notes, also H2 3.8): Consider the projection of the twisted cubic (i.e. the Veronese embedding $\mathbb{P}^1 \ni [x : y] \mapsto [x^3 : x^2y : xy^2 : y^3] \in \mathbb{P}^3$) from (i) the point $[1 : 0 : 0 : 1]$ and from (ii) the point $[0 : 1 : 0 : 0]$. In each case, show the image is an irreducible curve in $\mathbb{P}^2$, and find the defining equation.

Solution of Exercise 2.2: We present a solution just to part (i); the second part can be done similarly. Here is a sketch:

- Take the projection $[a : b : c : d] \mapsto [b : c : a - d]$. The image has parametrization $[x, y] \mapsto [x^2y : xy^2 : x^3 - y^3]$.
- A point $[a : b : c]$ is in the image if some $[x : y : a : b : c]$ is in the ideal

$$I := \langle a - x^2y, b - xy^2, c - (x^3 - y^3)\rangle.$$

- Eliminate x and y from the ideal to get a single equation in terms of a, b, and c. (This is what the Elimination and Extension Theorems will show us how to do.)
- If we really wanted to we could use M2 to check irreducibility, but that's kind of silly in this case: the image of a dominant rational map is irreducible.

□

What is the point? We no longer have to make ad-hoc arguments that the image is the vanishing ideal of some polynomial; we (or M2) can mindlessly perform some calculations.

Here is our goal. We would like to write I in the form

$$\langle f, g_1, ..., g_n\rangle$$

where f depends entirely on a, b, and c, and the g_i yield solutions x and y after we plug in a, b, and c which satisfy f.

In order to do this, we will need the Elimination and Extension Theorems.

Theorem 2.3 (Elimination Theorem): For $I \subseteq k[\boldsymbol{x}]$ with Grobner basis G (with respect to lexicographic ordering $x_n \prec ... \prec x_1$),

$$G_\ell := G \cap k[x_{\ell+1}, ..., x_n]$$

is a Grobner basis of

$$I_\ell := I \cap k[x_{\ell+1}, ..., x_n].$$

Proof of Theorem 2.3: It suffices to show that $\mathrm{LM}(I_\ell) = \mathrm{LM}(G_\ell)$.

It is obvious that $\mathrm{LM}(G_\ell) \subseteq \mathrm{LM}(I_\ell)$ as $G_\ell \subseteq I_\ell$.

To show that $\mathrm{LM}(I_\ell) \subseteq \mathrm{LM}(G_\ell)$, note that for every $f \in I_\ell \subseteq I$, we may write $f = \sum_{i=1}^{n} h_i g_i$ for $h_i \in k[\boldsymbol{x}]$ such that $h_i = 0 \implies \mathrm{LM}(g_i) \mid \mathrm{LM}(f)$.

Now we claim that $h_i = 0 \implies g_i \in G_\ell$. Since f does not involve any of the terms $x_1, ..., x_\ell$, nor can $\mathrm{LM}(g_i)$, which implies that none of the terms of g_i do.

Thus we conclude that $\mathrm{LM}(g_i)$ generates every $f \in \mathrm{LM}(I_\ell)$, as desired. And the way we have picked the h_i guarantee that G_ℓ is in fact a Grobner Basis of $\mathrm{LM}(G_\ell)$. □

This allows us to recover a partial solution in terms of a, b, and c. However, to confirm that the curve we derive is correct, we need to show that we may extend this partial solution to a complete solution in terms of a, b, c, x and y. We show that we may do this in general via Theorem 2.4.

Theorem 2.4 (Extension Theorem): Suppose k is algebraically closed field and $(a_2, ..., a_n) \in V(I_1)$. If I contains a polynomial f such that there exist polynomials $p \in k[x_2, ..., x_n]$ and $q \in k[x_1, ..., x_n]$ satisfying

$$f = p(x_2, ..., x_n)x_1^N + q(x_1, x_2, ..., x_n)$$

such that the degree of q in x_1 is less than N and

$$p(x_2, ..., x_n) \neq 0,$$

then there is some a_1 such that $(a_1, ..., a_n) \in V(I)$.

A proof of Theorem 2.4 may be found starting from Slide 22 of https://dacox.people.amherst.edu/lectures/gb1.handout.pdf.

2.4.1 Concluding with M2

We will discuss M2 more fully later. But if you are curious, the following M2 code allows us to compute a Grobner Basis and outputs it.[1]

```
R = QQ[x,y,a,b,c, MonomialOrder => Lex];
I = ideal(a - x^2*y, b - x*y^2, c - (x^3 - y^3));
G = groebnerBasis I;
G
```

After running this code, we get a basis including the element

$$a^3 - abc - b^3.$$

And thus our variety is $V(a^3 - abc - b^3) \subseteq \mathbb{P}^2$.

3 Applications

3.1 Graph Coloring

[1] You may try it yourself at https://www.unimelb-macaulay2.cloud.edu.au/#home.

3.1.1 Representing 3-coloring with polynomials

Consider a graph Γ. Let V_Γ be the vertices of Γ, which we'll label as $1, 2, ..., n$. Then let E_Γ be the edges (bidirectional). We want to think about how we can represent the three-coloring problem in terms of ideals. So consider working in the field $\mathbb{F}_3$ (integers mod 3). Now consider all the requirements we need for a valid 3-coloring.

- Each vertex has to be associate with one color.
- Adjacent edges cannot have the same color.

Since we are working in a field with three elements, we can arbitrarily set our colors to be in the set $\{-1, 0, 1\}$. Additionally, we will work in the polynomial ring $\mathbb{F}_3[x_1, ..., x_n]$ where each variable will be associated with the color of an edge.

Idea Consider the polynomial $x_i^3 - x_i$. The zeroes of this polynomials can be given by the set $\{-1, 0, 1\}$. As such, this may be a constraint regarding each vertex having a color.

Theorem 3.1: The zeroes of the ideal $I = \langle x_1^3 - x_1, x_2^3 - x_2, ..., x_n^3 - x_n \rangle$ allow for each vertex to be one of the three colors $\{-1, 0, 1\}$.

Proof of Theorem 3.1: Let $I = \langle x_1^3 - x_1, x_2^3 - x_2, ..., x_n^3 - x_n \rangle$. Then we know the zero locus is defined as

$$V(I) = \{(v_1, v_2, ..., v_n) \mid v_i^3 - v_i = 0 \text{ for } i = 1, 2, ..., n\}.$$

Then if we plug $-1, 0, 1$ for v_i, this will be in the zero locus. As such, each vertex can be colored with any of $\{-1, 0, 1\}$. □

Now we need a way to encode the adjacency constraint. That is, we need a polynomial in two variables such that we get zero if and only if we plug in two different values from $\{-1, 0, 1\}$. So consider

$$f(x_i, x_j) = x_i^2 + x_i x_j + x_j^2 - 1.$$

We will call this the adjacency polynomial.

Theorem 3.2: For every $v_i, v_j \in V_\Gamma$, v_i does not share a color with v_j if and only if the adjacency polynomial is zero based on the coloring.

Proof of Theorem 3.2: ($\Rightarrow$) Given v_i, v_j with different colors, we just need to show that for any choice of x_i and x_j such that $x_i \neq x_j$, the adjacency polynomial will turn out to be 0. So, we just check

$$(x_i, x_j) \in \{(-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0)\}.$$

Because of symmetry, we only have to check three cases:

$$f(-1,0) = (-1)^2 + (-1)(0) + (0)^2 - 1 = 0$$
$$f(-1,1) = (-1)^2 + (-1)(1) + (1)^2 - 1 = 0$$
$$f(0,1) = (0)^2 + (0)(1) + (1)^2 - 1 = 0.$$

So we can conclude that if two vertices are different colors, their adjacency polynomial is 0.

($\Leftarrow$) Suppose $f(x_i, x_j)x_i^2 + x_i x_j + x_j^2 - 1 = 0$. We want to show that $x_i \neq x_j$. But suppose they were equal. Then we would get

$$x_i^2 + x_i x_i + x_i^2 - 1 = 0$$
$$\Rightarrow 3x_i^2 = 1.$$

This polynomial has no solutions in $\mathbb{F}_3$. Therefore, we can conclude that if the adjacency polynomial is 0, then v_i and v_j are colored differently. □

This yields a polynomial representation of all colorings of the graph. Computing the Grobner basis of this ideal will just give us polynomial equations for all possible colorings without restriction on adjacency. However, the number of possible colorings grows approximately exponentially in the number of vertices. We can reduce the amount of computation necessary picking to adjacent nodes and assigning their colors arbitrarily. Suppose we construct our graph such that x_1 and x_2 are always adjacent and we choose to set $x_1 = -1$ and $x_2 = 1$. (Note that if we cannot have x_1 and x_2 be adjacent, then that means there are no edges in the graph and any choice of colors is valid). Then our new ideal becomes

$$I = \langle x_i^3 - x_i \mid i = 1,...,n, x_1 + 1, x_2 - 1, f(x_i, x_j) \mid (i,j) \in E_\Gamma \rangle.$$

This ideal has a Grobner basis that is much easier to compute. However, now we need to be able to distinguish between bases that are associated with valid colorings and those that cannot be associated with valid colorings.

Theorem 3.3: Consider the coloring ideal for a graph Γ:

$$I_\Gamma = \langle x_1 + 1, x_2 - 1, f(x_i, x_j) \mid (i,j) \in E_\Gamma \rangle.$$

The Grobner basis of I_Γ contains 1 as an element if and only if there is no 3-coloring of Γ.

Notably, this ideal does not contain the polynomials we first talked about that "constrain" the x_i to values in $\{-1, 0, 1\}$.

Proof of Theorem 3.3: ($\Rightarrow$) Assume that the Grobner basis of I_Γ contains 1. The zeroes of the generators of I_Γ correspond to colorings of Γ, which implies $1 = 0$ for some coloring. But this is a contradiction, so there is no valid 3-coloring.

($\Leftarrow$) Suppose Γ is 3-colorable. Then each generator of I_Γ has an assignment of variables such that they are equal to 0. As such, 1 will not be in the Grobner basis as all the polynomials will be equal to 0 and $1 \neq 0$. □

Now we prove that we don't need the original set of polynomials to constrain the values to $\{-1, 0, 1\}$.

Over any extension of the field $\mathbb{F}_3$, for each fixed $a \in \{-1, 0, 1\} \subset \mathbb{F}^3$, the quadratic adjacency polynomial in the other variable

$$f(a, x) = a^2 + ax + x^2 - 1$$

can be factored as

$$f(a, x) = (x - (a + 1))(x - (a - 1)).$$

This is because one can check that

$$a^2 + ax + x^2 - 1 = x^2 + ax + (a^2 - 1) = (x - (a + 1))(x - (a - 1)).$$

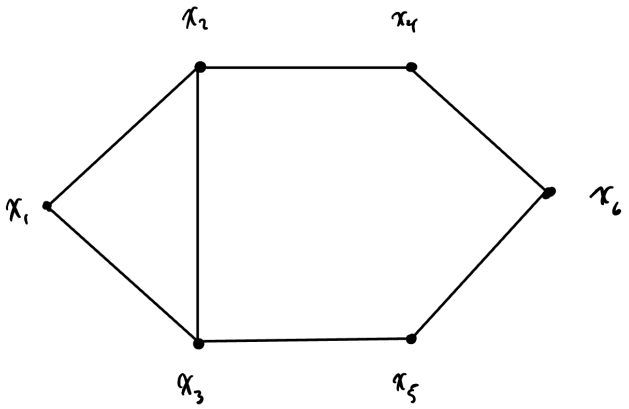
To show this,

$$\begin{aligned}(x - (a + 1))(x - (a - 1)) &= x^2 - ((a + 1) + (a - 1))x + (a + 1)(a - 1) \\ &= x^2 - 2ax + (a^2 - 1) \\ &= x^2 + ax + (a^2 - 1).\end{aligned}$$

This completes the proof. ■

3.1.2 Actually computing a 3-coloring

Consider the graph below.



Consider all the relevant polynomials:

$$\begin{aligned} &x_1^2 + x_1x_2 + x_2^2 - 1, \quad x_1^2 + x_1x_3 + x_3^2 - 1, \\ &x_2^2 + x_2x_3 + x_3^2 - 1, \quad x_2^2 + x_2x_4 + x_4^2 - 1, \\ &x_3^2 + x_3x_5 + x_5^2 - 1, \quad x_4^2 + x_4x_6 + x_6^2 - 1, \\ &x_5^2 + x_5x_6 + x_6^2 - 1 \end{aligned}$$

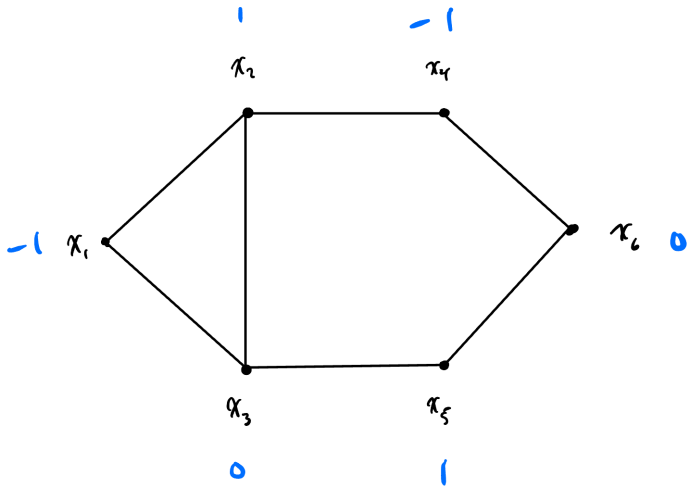
Now if we include $x_1 + 1$ and $x_2 - 1$, we have the polynomials to our coloring ideal for this Γ. Now if we compute the Grobner basis (using a software), we get

$$\begin{aligned} G(I_\Gamma) = \{&x_1 + 1, x_2 - 1, x_3, x_5x_6 + x_6, x_4x_6 + x_6^2 - x_4 - 1, x_5^2 - 1, \\ &x - 4x_5 - x_6^4 + x_4 + x_5 + x_6 + 1, x_4^2 + x_4, x_6^3 - x_6\}. \end{aligned}$$

This gives us a multitude of possible assignments, one of which is

$$x_1 = -1, x_2 = 1, x_3 = 0, x_4 = -1, x_5 = 1, x_6 = 0.$$

Now if we look below, we'll see that this is indeed a valid 3-coloring.



3.2 Chicken McNugget

The Oakland McDonald's sells Chicken McNuggets in sizes of 4, 6, 10, and 20. However, suppose when someone buys a 20-piece, they get lazy and only put 19. I'm wondering if I can buy 849 pieces because I love the class 21-849 so much. If I can do this, I also want to know how I can do it in the least number of boxes.

Idea: Consider the ideal

$$I = \langle x_4 - z^4, x_6 - z^6, x_{10} - z^{10}, x_{19} - z^{19} \rangle \subseteq \mathbb{Q}[z, x_4, x_6, x_{10}, x_{19}].$$

Note that

$$x_4^a x_6^b x_{10}^c x_{19}^d = z^{849}$$

as an element of $A(X)$ precisely when $4a + 6b + 10c + 19d = 849$.

3.2.1 Using the representative

We want our representative (in the sense of Theorem 1.9) of z^{849} to satisfy two properties:

1) If possible (i.e. if there is a member of the equivalence class satisfying this property) we do not want any term of our representative to be divisible by z.
2) The degree of the representative is minimal.

In other words, our goals are, *in this order*,

1) Minimize the degree of z in the representative.
2) Minimize the degree of the representative.

It can be seen that any representative of z^{849} satisfying these conditions must be a monomial. Here is an outline of the proof:

1) We convert all other variables to z by substituting x_4 with z^4, etc. This allows us to write any member of $A(X)$ as a polynomial in $k[z]$. (Note that $A(X) \simeq k[z]$.) **However, we do not yet combine previously distinct terms that have been converted into the same power of z.**
2) For any two converted terms, the degree of z cannot be the same, else we violate our uniqueness of representatives. So there are no terms to combine, for if there were, each term would be some scalar multiple of z^{849}.

3.2.2 Forcing the representative

Computing the Grobner Basis does not require us to use the lexiographic ordering on monomials! We only need the ordering to respect divisibility.

There exists an ordering such that

- if $\alpha_z < \beta_z$ then $\alpha \prec \beta$,
- and if $\alpha_z = \beta_z$ and $\deg \alpha < \deg \beta$, then $\alpha < \beta$.

Such an ordering suffices.

Why? Because Theorem 1.9 implies the representative of a polynomial in $A(X)$ which has none of its terms divisible by $\mathrm{LM}(g_i)$ is unique, which means that it must be the representative with the minimal leading monomial. (Any representative with a smaller leading monomial would also satisfy the conditions of Theorem 1.9, contradiction.)

4 Computation and Implementation

As we have already seen, Gröbner bases have many widely used applications, and are additionally highly computational. However, when analyzing their complexity, most Gröbner basis algorithms have extremely poor theoretical worst-case complexity. Despite this, advances in implementation (driven by the many possible applications of Gröbner bases) has led to highly-optimized algorithms that have good average-case performance in a practical sense.

4.1 Macaulay2

Macaulay2 is a free computer algebra system for commutative algebra and algebraic geometry that is designed to provide algebraic algorithms with fast and efficient implementations. It is designed to be useful for mathematicians, with core functionality including:

- arithmetic on rings, modules, and matrices
- Grobner basis algorithms and monomial orderings
- primary decomposition of ideals
- and many more, with thousands of user-made libraries to extend this functionality.

With this powerful tool, there has been many (2916+) published works that heavily rely on, or are motivated by, Macaulay2.

4.1.1 Gröbner Base Computation

As much of M2′s functionality and utility comes from its efficient handling of Grobner basis computation, the natural question is how M2 is able to optimize the Buchberger algorithm to such a degree. Towards that end, we analyze the two algorithms – the standard Buchberger vs M2′s optimized F4 algorithm – to understand where the optimization occurs.

Buchberger

Algorithm 1: Reduction

```
function BUCHBERGER(F)
    ▷ F = ⟨f_1, ..., f_k⟩ is a list of generators for our ideal
    G ← F
    k ← |G|
    P ← {(i, j) | 1 ≤ i < j ≤ k}
    while |P| > 0 do
        (i, j) ← P[0]
        P ← P[1 :]
        r ← S(G[i], G[j])%G
        if r ≠ 0 then
            G ← G ∪ {r}
            k ← k + 1
            P ← P ∪ {(i, k) | 1 ≤ i < k}
        end
    end
    return G
end
```

With this basic algorithm, we have that each S-polynomial is reduced completely before proceeding, and namely we proceed squentially through the pairs of generators one at a time, with the reductions therefore also happening one at a time, which prevents any reuse of intermediate calculations. Moreover, note that since we perform one polynomial divison (slow!) per generator pair, we get that the cost quickly increases when polynomials get more complex. Thus, we have a worst-case double exponential time complexity in the number of variables, and in practice, it is sensitive to pair selection order and reduction criteria for efficiency, but is overall quite inefficient.

As such, Faugere addressed some of these issues by allowing for efficient reuse of computation, parallelizable operations, and a more robust reduction and selection system via his F4 algorithm, which is what is used by Macaulay2.

F4

Helper functions:

Algorithm 2: Reduction

function REDUCTION(P', G)
  ▷ P' is a set of pairs, G is the current basis. Output G' a new basis
  $L \leftarrow$ **preprocess**(P', G)
  $M \leftarrow$ **insertMatrix**(L)
  ▷ Note **insertMatrix**(L) inserts the polynomials in L into a matrix
  $M' \leftarrow$ **rref**(M)
  $L' \leftarrow$ **extractPolynomials**(M')
  ▷ Note **extractPolynomials** is the inverse of **insertMatrix**
  $G' \leftarrow \{f \in L' \mid \forall g \in L, \mathrm{LM}(f) \neq \mathrm{LM}(g)\}$
  return G'
end

Algorithm 3: Preprocess

function PREPROCESS(P', G)
  ▷ P' is a set of pairs, G is the current basis. Output L a set of polynomials

  $P \leftarrow \left\{ \frac{\mathrm{lcm}(\mathrm{LM}(G_i), \mathrm{LM}(G_j))}{\mathrm{LT}(G_i)} * G_i \mid (i,j) \in P' \right\}$

  $Q \leftarrow \left\{ \frac{\mathrm{lcm}(\mathrm{LM}(G_i), \mathrm{LM}(G_j))}{\mathrm{LT}(G_j)} * G_j \mid (i,j) \in P' \right\}$

  $L \leftarrow P \cup Q$
  $D \leftarrow \{\mathrm{LM}(f) \mid f \in L\}$
  while $D \neq \mathrm{Mon}(L)$ **do**
    $m \leftarrow \max(\mathrm{Mon}(L) - D)$
    $D \leftarrow D \cup m$
    if $\exists g \in G, \mathrm{LM}(g) \mid m$ **then**
      $L \leftarrow L \cup \{m - \mathrm{LM}(g) \cdot g\}$
  end
  return L
end

F4

Algorithm 4: F4

function F4(F)
  ▷ $F = \langle f_1, ..., f_k \rangle$ is a list of generators for our ideal
  $G \leftarrow F$
  $k \leftarrow |G|$
  $P \leftarrow \{(i,j) \mid 1 \leq i < j \leq k\}$
  while $|P| > 0$ **do**
    $P' \leftarrow$ **select**(P)

```
        ▷ select chooses a nonempty subset of P
        P ← P − P′
        G′ ← reduction(P′, G)
        for g ∈ G′ do
            G ← G ∪ {g}
            k ← k + 1
            P ← P ∪ {(i, k) | 1 ≤ i < k}
        end
    end
    return G
end
```

Looking first at the helper functions, when we preprocess a set of pairs and current basis, we are essentially performing a batched version of the reduction used in Buchberger's algorithm, in that we output a set of polynomials whose leading-term structure encodes exactly the multiples of current basis elements needed to perform a batched reduction of all S-pairs in the inputted set of pairs. With respect to cost, we enumerate all monomials occurring in the current L, which potentially grows when compared with D, which has worst-case potential for a combinatorial blowup in cost. In practice, however, this is not the computational bottleneck.

Similarly, when we run reduction, we apply a sparse variant of Gaussian elimination to compute all of the polynomial reductions corresponding to all of the current S-pairs in one step. This is done by insertion into a matrix, row-reduction, and extraction back into a set of polynomials, in which the underlying sparse linear algebra is far faster in practice than repeated polynomial division—but matrix size (rows $\sim |L|$, cols $\approx$ # distinct monomials) is highly proportional to the performance. There are highly optimized sparse-matrix libraries inside Macaulay2 to speed this process up, though this is by far the computational bottleneck, so F4 is optimized to perform as little of these operations as possible.

And lastly, for the core F4 algorithm, we use these helper functions to reduce a whole block of S-pairs together rather than one at a time. Moreover, we note that if our pair selection function only chooses singleton pairs, we actually reduce to the original Buchberger algorithm. The core improvements against the Buchberger algorithm occur from the preprocessing of common divisibility, as this allows for our reduction matrix to contain no redundant or unnecessary rows or columns, and then the sparse Gaussian elimination allows for highly parallelized reduction. Then, pairing this with an optimized pair selection heuristic, we get a highly efficient in practice (though not necessarily asymptotically better) algorithm for use in Macaulay2.

These Grobner bases can be calculated over a wide class of base rings and ideals. Namely, Grobner bases can be calculated for ideals and submodules over the integers, fields, polynomial rings over such rings (w/wo skew commutative multiplication), quotients of such rings, Weyl algebras, and any ring in the "closure" of the previously mentioned (i.e. a polynomial ring of a quotient etc.)

Then, given a Grobner basis, Macaulay2 is then able to perform many computations, such as checking for ideal membership by reducing a ring element modulo the basis, dimension,

degree, Hilbert series, Hilbert polynomial, radical and primary decomposition, elimination, projections, and countless other applications.

For examples, demos, and documentation on utilizing M2, see:

- https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/___Groebner__Basis.html
- https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/___Gr%C3%B6bner_spbases.html
- https://macaulay2.com/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/_simple_sp__Groebner_spbasis_spcomputations_spover_spvarious_springs.html
- https://www.unimelb-macaulay2.cloud.edu.au/#tutorial-groebner-4

4.2 Interactive Theorem Proving

4.2.1 Introduction

Interactive theorem provers are software that allow users to interactively work with the computer to construct *formal* mathematical proofs. Namely, these systems are generally functional programming languages that utilize the Curry-Howard correspondence to specify and form formal proofs, in that proofs are represented as types. Such systems are quickly rising in popularity for their perfect garuntees in verifiabile correctness, with ITPs such as Lean, Rocq, Isabelle, etc. all being utilized in research and industrial work in pure math, ML, physics, programming language theory, and many more fields.

Here, the language kernel itself verifies proofs (constructed as terms inhabiting a goal type) by type inference and typechecking, a strategy that is incredibly generalizable in ITP's like Lean, that rely on a complex dependent type theory, making it incredibly expressive and powerful across application domains. Moreover, this efficient kernel and verifiability provides a strong reward signal for ML applications, with automated neural theorem provers such as AlphaProof, STP, and others being able to (formally) prove difficult IMO, Putnam, and assist in research-level problems.

Additionally, the verifiable correctness allows for a low-trust collaborative system for mathematical research to thrive, with thousands of contributors – with mathematical backgrounds ranging from high-school students to fields medalists – collaborating on projects such as:

- Mathlib, a comprehensive and research-level library of mathematical results in Lean. [Mathlib]
- Characterization of Equational Magmas (*Heavily* relied on Grobner Bases!) [Tao]
- Polynomial Friedman-Ruzsa Conjecture [Tao]
- Proof of the Kepler Conjecture [Hales]
- Perfectoid Spaces Formalization [Buzzard]

And many others.

4.2.2 Dependent Type Theory

In more detail, dependent type theory – as formalized in the "Calculus of Inductive Constructions" – serves as the underlying logic and type system for Lean and several other modern proof assistants. Unlike simple type theories where types classify terms but cannot depend on them, dependent types allow a type to be parameterized by values. This enrichment yields two fundamental type constructors:

- Π-types (dependent functions): Given a collection of types $B : A \to \text{Type}$, the dependent function type $\Pi(a : A), Ba$ generalizes function types such that its inhabitants are functions that map $a : A$ to elements of Ba. This corresponds to universal quantification.
- Σ-types (dependent pairs): Given $B : A \to \text{Type}$, the dependent pair $\Sigma(a : A), Ba$ consists of pairs (a, b) where $a : A$ and $b : Ba$. It corresponds to existential quantification.

Moreover, CIC extends this core DTT with:

- Inductive Types: Transparent definitions of a type constructuve from constants and functions that create terms of that type – which is perhaps self-referential, modulo structural recursion. Namely, each inductive type I has a recursor $I.\text{rec}$ or eliminator $I.\text{elim}$ that supports pattern-matching on constructors and ensures total, terminating definitions.
- Universe Hierarchy: To avoid Girard's paradox (type theory version of Russell's paradox), CIC contains a (countable) sequence of universes $\text{Type } 0 : \text{Type } 1 : \text{Type } 2 : \ldots : \text{Type } *$. Types inhabit these universes, and coercions ($\text{Type}.\{u\} \subseteq \text{Type}.\{v\}$ for $u \leq v$) preserve consistency while enabling universe polymorphism.

Thus, with this rich type theory, we can – via the previously mentioned Curry–Howard Correspondence – encode propositions as encoded as types, and proofs as terms inhabiting a type. Thus, proving a theorem P means constructing a term of type P. Proof objects can be inspected, composed, and even executed as programs (for computationally relevant proofs).

Under this typechecking mechanism, the Lean kernel performs definitional equality checks by normalizing terms to head-normal form and simply checking equality of the normal forms. This guarantees both consistency (no proof of $\bot$) and decidability of type-checking. Moreover, the strong normalization property ensures every well-typed term reduces to a normal form, which is critical for implementing automation and automated proof search systems.

At a higher level, in ITP, DTT and CIC allows for incredibly expressive specifications, generating certified code, metaprogramming and imperitive programming with monads, and modularity coming from universe polymorphism and inductive types. As such, interactive theorem provers are empowered to richly specify complex, formal mathematics and logic, and are ripe to automate the process with both human-driven proof discovery and machine-checked verification.

Note that under this theory, we can define structures that are both computable and noncomputable, meaning that we can define complex objects and properties as programs

which cannot be computed. This is valuable for theorem proving, but provides a challenging problem when building CAS systems in ITPs. For example, the real numbers are implemented as Cauchy completions in Lean, which is incredibly valuable for proving properties about the real numbers, but makes it impossible to directly compute results on real numbers. This is the most difficult challenge when hoping to implement interfaces for external computational systems in Lean.

4.2.3 Theorem Proving in Lean4

In practice, theorem proving in Lean is not handled by directly constructing the proof term inhabiting the goal type, but rather by using advanced human-made tactics that procedurally and interactively construct the proof term in a human-parseable manner.

4.3 LeanM2

4.3.1 Motivation

Now with a introductory background in interactive theorem proving like Lean, Grobner basis computation systems such as Macaulay2, and the utility of Grobner bases in general, we can note the strengths shortcomings of each system.

Namely, interactive theorem provers have the unique power to be able to state and rigorously prove theorem statements, rather than simply just give an answer like a calculator or M2 would do. This, combined with the large community results in incredibly sophisticated, research-level problems being proven end-to-end in Lean. However, this comes with the caveat that actually forming these proofs are incredibly difficult, and moreover, Lean is not good for performing calculations, as it aims to verify the validity of a result rather than compute the result itself.

On the other hand, CAS systems like Macaulay2 have the opposite situation, in that they excel at computation and solving problems, but have no guarantees of correctness or more advanced specification abilities to work on complex problems at the research level – all of that must be handled on paper and pencil by humans.

As such, we propose *LeanM2*, which aims to upgrade Macaulay2 to form formal proofs for use in Lean, bridging this aforementioned gap to offset the drawbacks of both systems by combining them as much as possible.

4.3.2 Implementation

Code for the LeanM2 implementation is released publically at https://github.com/riyazahuja/lean-m2.

We now outline the basic idea behind this implemention. Namely, we aim to simply convert types and expressions in Lean to M2 syntax, and then send this command to M2, get our response, and convert back with M2, providing a proof witness to construct the certificate. For the purposes of this demo, we restrict to ideal membership questions, as there is a strong Lean API for applying these explicit proof witnesses and constructing our certificates.

Namely, in order to achieve each of these steps, we perform the following:

1) Convert current Lean hypotheses + goals into M2 command
    a) Parse proof state in `TacticM` monad and extract relevant structures
    b) Synthesize metavariables and types into computable structures
    c) Combine new structures into M2 command
2) Receive M2 response
    a) parse messy M2 response into proof witness
3) Convert response into proof certificate and Lean syntax
    a) Build Mathlib API for GB proof certification
    b) Create parser for M2 outputs into syntactic, computable structures
    c) Reinstance structures as `Lean.Expr` and parse into valid proof
    d) Create and apply tactics to automatically use certificates to close goals.

With this in mind, we define the `M2Type` class to represent the mappings from the semantic, and possibly nonconstructive, Lean types into the syntactic M2 representations. Here, `M2Type` instances the semantic (often noncomputable) meaning of Lean code to the corresponding syntactic (computable) type. This is done by constructing a computable subtype of the Lean type with explicitly constructed partial isomorphisms between the types, with formal proofs of invertibility. Moreover, this instance encapsulates `Repr`, `UnRepr` for easy conversion to/from M2 and parsing and serialization to/from strings.

We implement support for the (inductive closure of the) following types and subtypes:

- $\mathbb{Z}$: Integers
- $\mathbb{Q}$: Rationals
- $\mathbb{R}$: Cauchy Completion $\mapsto$ rationals + trancendental fns
- $\mathbb{C}$: See above.
- $GF(p^n)$: Galois (finite) field, implemented as splitting fields (choice-dependent!) $\mapsto$ Conway w/ proof of algebraic equivalence between the two.
- Polynomial rings
- Quotient rings over finitely generated ideals

There is a large amount of complexity associated with constructing the subtypes and their respective bindings, as well as implementing parsing, and the inductive closure of these types. However, that discussion is outside the scope of this course and it is left to the curious reader to read from the posted codebase.

Then, with this semantic/syntactic duality between M2 types and Lean types, we construct the M2 command as follows:

Command:

```
R=QQ[x0, x1]
f=((x0)^2 + (x1)^2)
I=ideal(x0,x1)
G=gb(I,ChangeMatrix=>true)
f % G
(getChangeMatrix G)*(f// groebnerBasis I)
```

Response:

```
i1 : R=QQ[x0, x1]
o1 = R
o1 : PolynomialRing

i2 : f=((x0)^2 + (x1)^2)
       2     2
o2 = x0  + x1
o2 : R

i3 : I=ideal(x0,x1)
o3 = ideal (x0, x1)
o3 : Ideal of R

i4 : G=gb(I,ChangeMatrix=>true)
o4 = GroebnerBasis[status: done; S-pairs
encountered up to degree 0]
o4 : GroebnerBasis

i5 : f % G
o5 = 0
o5 : R

i6 : (getChangeMatrix G)*(f//
groebnerBasis I)
o6 = {1} | x0 |
     {1} | x1 |
              2       1
o6 : Matrix R  <--- R
```

This system is extensible in that we can freely modify the command to unlock additional functionality of M2, as the difficult part of building this general interface is the coercion of types between the two softwares, which we have already implemented. Therefore, the extension of our methodology is only bottlenecked by the Lean and Mathlib API, and the ability to state and construct formal proof certificates.

4.3.3 Results

With this implemention plan in mind, we implement the first type-consistent computer algebra system in Lean for Macaulay2. As a first work in this field, we focus on Macaulay2′s Grobner-basis and ideal membership computation capabilities for their relatively simple proof witnesses, as we can thereby improve the power of Macaulay2 to be verified and proof-forming within Lean4.

We see that as our system is entirely locally hosted, and relies only on Macaulay2′s efficient computation system, and supports many complex types and subtypes in Lean4 beyond rational polynomial rings, from basic number rings to nested polynomial quotient rings over noncomputable dependent types, significantly outperforming the state-of-the-art `polyrith` tactic.

We publically release the code on github for this project and hope to extend this work to more types and functionality in Macaulay2.

4.3.4 Future Work

As previously mentioned, we hope in the future to port over more of M2 functionality, as the primary bottleneck is the Lean/Mathlib API on Grobner bases, as that is needed for constructing proof certificates. Additionally, we will also pursue more support for more datatypes in Macaulay2 and improved tactic UI.