SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Gröbner Bases and Elimination in Macaulay 2

av

Christian Eriksson

2026 - No K30

Gröbner Bases and Elimination in Macaulay 2

Christian Eriksson

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Sofia Tirabassi

2026

Abstract

This thesis covers fundamental concepts in algebraic geometry, polynomial elimination and Gröbner bases. It accomplishes this by introducing affine spaces and varieties to allow a better understanding of what algebraic geometry is, and then pivots to exploring Gröbner bases. This is done with the final goal of constructing an algorithm that can compute these bases deterministically. At first, monomial ideals are considered but the scope is widened to polynomial ideals. A verification of membership to Gröbner bases, the Buchberger criterion, is proven. The Buchberger algorithm, or the construction of Gröbner basis, is implemented in the computer algebra system Macaulay2 using the criterion. Univariate division, multivariate division, and solutions to systems of polynomial equations were also implemented algorithmically. The algorithms were verified programmatically with unit tests. The thesis provides an introduction to the construction and implementation of algorithms as well as a solid foundation for future expansion into algebraic geometry or construction of algorithms.

Abstract

Den här kandidatuppsatsen fokuserar på grundläggande koncept inom algebraisk geometri, polynomial eliminering samt Gröbnerbaser. I uppsatsen introduceras affina rum och varieteter för att främja förståelsen av algebraisk geometri, och därefter skiftar fokus till utforskandet av Gröbnerbasen. Syftet med det här är att konstruera en algoritm som deterministiskt kan beräkna en Gröberbas. För att uppnå detta diskuteras inledningsvis monomialideal, vilket sedan vidgas till att innefatta även polynomideal och dess användbara egenskaper. Därefter bevisas grundkärnan till Gröbnerbas-algoritmen, Buchberger-kriteriet. Detta kriterium används för att programmera Buchbergers algoritm, det vill säga skapandet av en Gröbnerbas, i datoralgebrapråket Macaulay2. Envariabel och flervariabel polynomdivision samt lösningar till polynoma system implementeras också i Macaulay2. Resultaten verifieras med enhetstester. Sammantaget fungerar uppsatsen som en introduktion till konstruktionen och implementationen av algoritmer samt utgör en grund för framtida utforskande av algebraisk geometri och algoritmkonstruktion.

Contents

1 Introduction **8**

2 The Algebraic Abstraction of Geometry **9**
2.1 Polynomials . . . 9
2.2 Affine Spaces and Varieties . . . 10
2.2.1 Affine Spaces . . . 10
2.2.2 Polynomials in Affine Spaces . . . 12
2.2.3 Affine Varieties . . . 12

3 Algebra Theory **14**
3.1 Rings and Ideals . . . 14
3.1.1 Monomial Ideals . . . 14
3.1.2 Dickson's Lemma . . . 15
3.2 Hilbert Bases Theorem . . . 16
3.3 Gröbner Bases . . . 17
3.3.1 Properties Of Gröbner Bases . . . 18
3.3.2 Buchberger's Criterion . . . 18

4 Algorithms in Macaulay2 **22**
4.1 Division Algorithms . . . 22
4.1.1 Univariate Division Algorithm . . . 22
4.1.2 Multivariate Division Algorithm . . . 23
4.2 The Buchberger Algorithm . . . 24
4.3 Solving Systems of Polynomials . . . 25
4.3.1 Book Examples of Polynomial Solutions . . . 26
4.3.2 Solving a System of Polynomial Equations . . . 27

5 Discussion **28**
5.1 Mathematical Proof and Knowledge . . . 28
5.2 Algorithm Implementation . . . 28

A Macaulay2 source files **31**
A.1 Algorithms . . . 31
A.2 Examples . . . 40

References **45**

1 Introduction

This bachelor's thesis is centered around algorithms, an intersection between theoretical mathematics and practical computer science. Algorithms stand with one foot in each domain, and both are equally important. In order to begin instructing a computer on how to solve a problem, a clear idea of the problem must be established. Additionally, a concrete way to solve the problem must also exist. This is the theoretical aspect of algorithmic construction and is agnostic of any computer implementation. Proving that a result is unique, or verifying that an algorithm terminates are critical aspects of success. The theoretical alone is insufficient and the implementation is equally necessary. These algorithms are created to solve practical problems, so one must be able to provide inputs to some mechanism and recieve outputs.

The gap between the theoretical and practical is never easy to bridge, they both present unique challenges. The main algorithm being studied in this thesis is the construction of Gröbner bases. These bases are computationally useful and can be applied to optimizing the movement of robotic arms [LM21]. As such, finding an algorithm that always yields a Gröbner basis and terminates after a finite number of steps is of interest. The steps taken to prove the algorithm theoretically were inspired by [DAC10]. After an understanding of the theory was eastablished, the computer algebra system (CAS) Macaulay2 [GS] was leveraged to implement the algorithm.

Division and polynomial elimination algorithms were tackled through a different lens. Instead of standing in the middle of mathematics and computer science, a step towards the computer was taken. This thesis explores the process of receiving a proven algorithm and implementing it. The theory behind division algorithms is well known, and implementing them provided sufficient resistance when acclimating to a new computer language. When writing the division algorithms, the challenges were syntactical or a general lack of knowledge surrounding how a CAS works. The algorithms themselves are straightforward from a mathematical implementation view, but they were a necessary step to get accustomed with Macaulay2. The implementation of the polynomial elimination was a step back towards the mathematical domain. Without solid mathematical knowledge surrounding ideals it would have been difficult to translate the pseudocode presented ([LM21]) into a working algorithm.

2 The Algebraic Abstraction of Geometry

Algebra can be seen as the mathematical expression of the human need for structure and abstraction. Through algebra, humans try to capture more of the mathematical, and natural, domain under their pen. The endless hunger for knowledge can express itself in various ways. When using algebra as a tool, creating abstract models that capture generic structure is favored. However, these abstract models can be difficult to understand. The ability to navigate between levels of abstraction is a vital tool for any mathematician interested in applying algebraic methods.

This thesis assumes a degree of familiarity with algebraic structures, but the uninitiated peruser is kindly directed towards the digestible "Introduction to Modern Algebra" [Wei70] with guiding exercises, or alternatively, the more rigorous "Abstract Algebra" [DF03] for a deeper understanding.

This section will primarily focus on defining geometric concepts that can be viewed through an algebraic lens. An algorithm to solve the end concept, affine varieties is introduced.

2.1 Polynomials

In order to discuss mathematical concepts, a compact object is required. This is done by compressing information into a single mathematical expression. In order to discuss polynomial rings, we begin by defining a relatively uncompressed case, monomials. Then, we consider a collection of monomials, a polynomial. Finally, we can consider the compact object, polynomial rings.

Definition 2.1. A **monomial** in $x_1, \ldots, x_n$ is a product of the form

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

where all $\alpha = \alpha_1, \ldots, \alpha_n$ are nonnegative integers.

As the root prefix "mono" suggests, there is only one of these terms x^α. They are independent and self-contained. However, as we are creating the tools to generalize and deal with more abstract concepts, we also want to include the ability to capture any linear combination of these monomials.

Definition 2.2. A **polynomial** f in $x_1, \ldots, x_n$ with coefficients, $a_1, \ldots, a_n$, in a field k is a finite linear combination of monomials.

$$f = \sum_{\alpha} a_\alpha x^\alpha, \quad a_\alpha \in k$$

With the polynomial definitions at hand, we can construct the final abstraction for our mathematics involving polynomials, the polynomial ring. This is defined as $k[x_1, \ldots, x_n]$. When leveraging this powerful abstraction, we capture n dimensions of variables on any field k. This achieves the goal of compact analysis.

2.2 Affine Spaces and Varieties

For the purpose of this thesis, geometry will be restricted to how points, lines, planes (and higher dimensional objects) interact with space. To build a bridge between algebra and geometry, there must be an established way to discuss geometry algebraically. This is where the concept of affine spaces comes in.

2.2.1 Affine Spaces

Let us take a moment to reflect around how one might represent a geometric object. Consider a line. One could use the well known two dimensional cartesian plane, plot the points $(x_1, y_1), (x_2, y_2)$ and then pull a pen between the two points. Here, a line will be created, but it is dependent on the following: a coordinate system and a restriction of dimensions so we can physically represent it. These restrictions are undesirable when studying a line algebraically, so we must circumvent these restrictions. The solution is affine spaces. Affine spaces are coordinate-agnostic spaces. This eliminates the need for a coordinate system, which was a prerequisite, when drawing the lines. The affine space is also easily extended beyond the physical.

Definition 2.3. An *affine space* is either the empty set, or a triple $\langle E, \vec{E}, + \rangle$. E is a nonempty set of points while $\vec{E}$ is a vector space (of translations). The action $+$ is an operation on both E and $\vec{E}$ defined as follows: $+ : E \times \vec{E} \to E$. The following axioms are assumed for affine space:

1. $a + 0 = a$ for every $a \in E$.
2. $(a + u) + v = a + (u + v)$ for every $a \in E$ and every $u, v \in \vec{E}$.
3. For any two points $a, b \in E$, there is a unique $u \in \vec{E}$ such that $a + u = b$

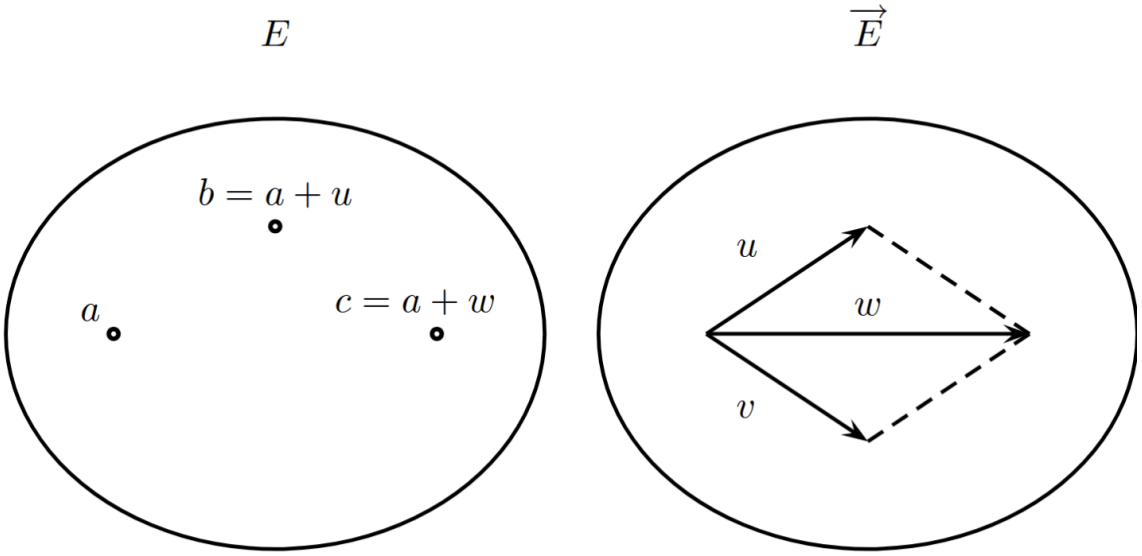


Figure 1: Intuitive understanding of affine space [Gal25]

A quick note of importance on each axiom. If you start at point A and do nothing, you should still be at A. Axiom two deals with the associative property. The ability to rearrange calculations simplifies any algebra involved. The last axiom, that of uniqueness, is computationally useful. Assuming that a unique element in $\vec{E}$ always exists that can connect two points a and b is useful when solving these problems on a higher level of abstraction, especially when a computer that cannot handle errors is involved.

The intuition behind the coordinate-agnostic nature should be clear when considering figure 1. Here a separation between the vectors and the points *to which the vector is applied to* should be noted. These two can be defined independently and therefore there is no system tying them together.An example is provided for further clarity.

Example 2.4. Given a field k and a positive integer n, define the n-dimensional *affine space* over k as:

$$k^n := \langle \left\{ [\hat{x}_1, \ldots, \hat{x}_n]^T : \hat{x}_i \in k, i = 1, \ldots, n \right\}, \vec{k^n}, + \rangle$$

Where $\vec{k^n}$ is the vector space of translations, and $+$ denotes the action of shifting a point by a vector via component-wise addition.

To further our understanding of the space, we can use the concrete example where the field in question is $\mathbb{R}$. Then, we get the familiar space $\mathbb{R}^n$. So the affine

space is more abstract than the one we are used to dealing with from real world applications where we are restricted.

2.2.2 Polynomials in Affine Spaces

With a better understanding of affine spaces, we will consider how polynomials relate to this space. Using the definition 2.2 we define the polynomial *function* $p \in k[x_1, \ldots, x_n]$ as follows:

$$p : k^n \to k$$

The function maps the multidimensional field onto the single dimension. This p takes every $x_1, \ldots, x_n$ and replaces it with $a_1, \ldots, a_n \in k[x_1, \ldots, x_n]$. This is done easily by substituting every $x_i = a_i$ in the polynomial p. Since all the subbed in a's are elements $k[x_1, \ldots, x_n]$, then $p(a_1, \ldots, a_n)$ is also in $k[x_1, \ldots, x_n]$.

2.2.3 Affine Varieties

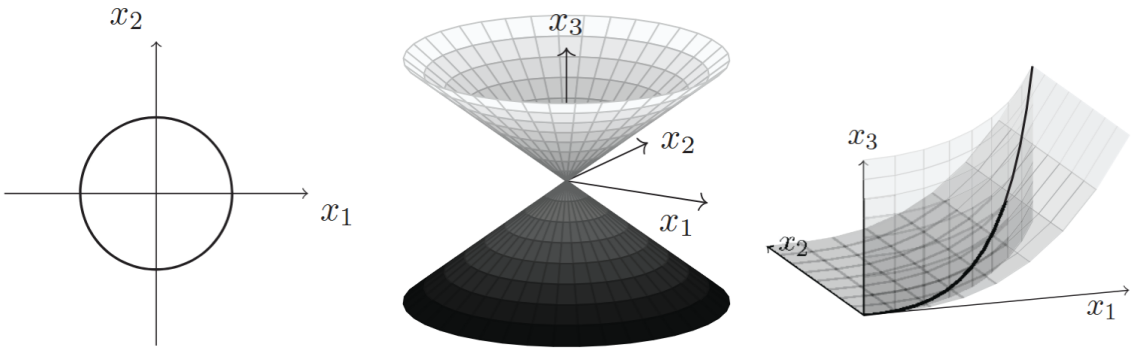
With an understanding of how the space we will operate upon will work and how the polynomials can be evaluated on that space, the first algebraic structure of geometry will be discussed: affine varieties.

Definition 2.5. Let k be a field, and let $f_1, \ldots, f_s$ be polynomials in $k[x_1, \ldots, x_n]$. Then, set

$$\boldsymbol{V}(f_1, \ldots, f_s) = \{(a_1, \ldots, a_n) \in k^n |\ \ f_i(a_1, \ldots, a_n) = 0\ \ \forall 1 \leq i \leq s\}$$

We call $\boldsymbol{V}(f_1, \ldots, f_s)$ the **affine variety** defined by $f_1, \ldots, f_s$.

In other words, the affine variety is the set of $(a_1, \ldots, a_n)$ that solve all the equations defined in the variety simultaneously. Or, it is the intersection of all the polynomial roots. Even though $\boldsymbol{V}$ might simply look like a graphed equation, figure 2a only provides the *base* for intuition. While it may look like a regular circular graph, with familiar x and y coordinates, the keen observer will note that we are dealing with two dimensions, x_1, x_2 and that the graph is in fact the where this polynomial studied is equal to 0. If we extract the polynomial from the variety and set it to 0, we end up with $x_1^2 + x_2^2 = 1$. Figures 2b and 2c show the increasing complexity in three dimensions. Graphing this equation makes the roots of the polynomial self-evident, but graphing it is neither necessary nor desirable. It is only



(a) $\mathbf{V}(x_1^2 + x_2^2 - 1)$. (b) $\mathbf{V}(x_1^2 + x_2^2 - x_3^2)$. (c) $\mathbf{V}(x_1^2 - x_2, x_1^3 - x_3)$.

Figure 2: Examples of affine spaces [LM21]

a tool to inform intuition. This is especially true as we are interested in n dimensions and graphing will become next to impossible.

3 Algebra Theory

With affine varieties under our belt, there is an obvious follow up to this knowledge, especially for those interested in the real application of this mathematics. Namely, how do we solve these varieties? Luckily, the problem of asking for the affine variety points can be reduced to solving a system of polynomial equations. This is easier said than done, and computational complexity arises when considering n dimensions. Most real world problems will involve many dimensions, and being able to solve these problems quickly and with confidence is of importance. To do so, utilizing a computer algorithm would be a solution.

Every computer language has important data structures that determine how the language is shaped. For Macaulay 2, ideals are one of these types. They are important to algebra, and native support for manipulating ideals exists. However, to efficiently handle ideals, there is a problem that needs to be solved with great confidence. Namely: how do we go between abstract entities such as ideals and something that a computer can interact with?

3.1 Rings and Ideals

The ideal description problem formulates the bridge between the abstract and machine world. It does this by concretizing the abstract concept of an ideal to a tangible set of generating polynomials that can be interacted with and evaluated. In order to bridge this gap, finite polynomial ideals have to be found. Starting by analyzing monomial ideals in $k[x_1, \ldots, x_n]$ is the first step.

3.1.1 Monomial Ideals

Much like when introducing affine varieties, starting with a single monomial instead of the combination of them reduces complexity. So, when studying the ideals, we start by considering a less complex problem.

Definition 3.1. An ideal $I \subseteq k[x_1, \ldots, x_n]$ is a monomial ideal that is determined by a set $A \subset \mathbb{Z}^n$. All polynomials in the ideal can be written as a linear combination of $\sum_{\alpha \in A} h_\alpha x^\alpha$ where h_α lies in $k[x_1, \ldots, x_n]$.

Monomial ideals can largely be thought of as their exponents. This is their main benefit and why they are important when dealing with computer algebra systems.

We will see this property in the coming proofs. As an example, consider the following lemma:

Lemma 3.2. *Let $I = \langle x^\alpha | \alpha \in A \rangle$ be a monomial ideal. Then a monomial x^β lies in I if and only if x^β is divisible by x^α for some $\alpha \in A$.*

Proof. $\Leftarrow$ This can be proven by considering the exponents. Starting off with showing x^β to x^α, the exponents can be leveraged. If β is a multiple of α then x^β is in the ideal by definition since x^α generates the ideal.

$\Rightarrow$ Using the definition of the monomial ideal, we can write x^β as the polynomial sum of generators of the monomial ideal and then write h_i as the sum of generators again.

$$x^\beta = \sum_{i=1}^{s} h_i x^{\alpha(i)} = \sum_{i=1}^{s} \left(\sum_j c_{i,j} x^{\beta(i,j)} \right) x^{\alpha(i)} = \sum_{i,j} c_{i,j} x^{\beta(i,j)} x^{\alpha(i)}$$

Here we can see that every term on the right hand side is divisible by x^α, so then x^β on the left hand side must also be so. This proves that we can follow our intuition regarding the exponents and their divisibility.

□

3.1.2 Dickson's Lemma

This lemma demonstrates the usefulness of monomial ideals. The ability to reduce ideals to finite generators is a step towards closing the gap between the machine and abstract.

Theorem 3.3. *Let $I = \langle x^\alpha | \alpha \in A \rangle$ be a monomial ideal. Then I can be finitely written by $I = \langle x^{\alpha(1)}, ..., x^{\alpha(s)} \rangle$, where all α lie in A.*

Proof. Consider an inductive approach. Starting off with $n = 1$ for our $k[x_1, \ldots, x_n]$, it is easy to see that the smallest element of $A \subseteq \mathbb{Z}_{\geq 0}$ generates the remaining elements. Whith $n > 1$, we start our induction. Consider the nth case. We have the monomial ideal, call it J, from the $n-1$ case. This is finite by our induction, so when our A goes from a subset of $\mathbb{Z}_{\geq 0}^{n-1}$ to $\mathbb{Z}_{\geq 0}^{n}$, we can generate slices of these J ideals with the new $x_n = y$ variable. We take some sufficiently large $m \geq 0$ and group the J_l ideals by their corresponding y value up to $m-1$. In other words, let the ideal J_l be generated by x^β where $x^\beta y^l \in I$. These J_l slices form a topological map of the ideal I. Thus, I is also finite. □

After a finite ideal is generated, the monomial ideal membership problem is solved. This is since we can take a polynomial and check if the remainder from division by the finite monomial ideal is 0. If it is, it belongs to the monomial ideal since one of the members will generate it.

3.2 Hilbert Bases Theorem

In this section, the result that monomial ideals have finite generators is extended to polynomial ideals. The basic idea is that by rearranging the ideal to a monomial ideal using the leading term, we can apply Dickson's lemma. The first step towards bridging the gap between polynomial and monomial ideals is by crafting an ideal of interest. Namely, the ideal generated by the leading term of a polynomial ideal.

Definition 3.4. Let $I \subseteq k[x_1, \ldots, x_n]$ be an ideal other than $\{0\}$ and fix a monomial ordering. Then

1. $LT(I) = \{cx^\alpha \mid \text{there exists } f \in I \backslash \{0\} \text{ with } LT(f) = cx^\alpha\}$.
2. $\langle LT(I) \rangle$ is the ideal generated by the elements of $LT(I)$.

$LT(I)$ is a large set, but it is a bridge between monomial ideals and polynomial ideals. After all, the difference between a monomial and the lead term of the polynomial are the other monomials in the polynomial. If we can capture them by constructing an ideal that generates everything by using the leading term, we can create a superset to operate upon. $\langle LT(I) \rangle$ is that set.

Proposition 3.5. *Let $I \subseteq k[x_1, \ldots, x_n]$ be an ideal different from the $\{0\}$.*

1. *$\langle LT(I) \rangle$ is a monomial ideal.*
2. *There are $g_1, \ldots, g_t \in I$ such that $\langle LT(I) \rangle = \langle LT(g_1), \ldots, LT(g_t) \rangle$.*

Proof. Since we have used the leading term to construct an ideal, the difference between the monomial ideal and $\langle LT(I) \rangle$ is a nonzero constant. Consider a single $g_i = g$, and break the lead term down to the lead coefficient and lead monomial. $LT(g) = LC(g) \cdot LM(g)$ means that $LT(g)$ is a field scalar of $LM(g)$, which means that $LT(g) \in \langle LM(g) \rangle$. We can multiply by the inverse to show that $LT(g) \cdot LC^{-1}(g) = LM(g)$. But $LC^{-1}(g)$ is still in $k[x_1, \ldots, x_n]$, so the same logic applies and they are equal to each other. So, $\langle LT(I) \rangle = \langle LM(I) \rangle$. The same logic is used for the second item. Using Dickson's Lemma tells us that monomial ideals are finitely

generated. Since $\langle LT(I)\rangle$ only differs from a monomial ideal by a nonzero constant, Dickson's lemma can be applied here as well and I is finitely generated. □

With all this information accumulated, we can formally prove that every polynomial ideal has a finite generating set. This theorem is known as the Hilber Basis Theorem.

Theorem 3.6 (Hilber Bases Theorem)**.** *Every ideal $I \in k[x_1, \ldots, x_n]$ has a finite generating set. In other words, $I = \langle g_1, \ldots, g_t\rangle$ for some $g_1, \ldots, g_t \in I$.*

This proof is outside of the scope, but is given in chapter 2 of Cox [DAC10] as Theorem 4 on page 77.

3.3 Gröbner Bases

The generating set $\langle LT(I)\rangle$ was introduced in the last section. Dealing with monomials reduces the complexity drastically as opposed to polynomials, so if we can use the $\langle LT(I)\rangle$ instead of the polynomial one, it would simplify calculations. Recall that the Hilbert basis theorem stated that every polynomial ideal has a finite generating set. However, it is not unique. We will start with an ideal that is of special interest to the world of polynomial calculation.

Definition 3.7. Choose a monomial ordering on the polynomial ring $k[x_1, \ldots, x_n]$. A finite nonzero subset, $G = \{g_1, \ldots, g_t\}$ of an ideal, I, *which* is a subset of *polyRing* is a **Gröbner basis** if

$$\langle LT(g_1), ..., LT(g_t)\rangle = \langle LT(I)\rangle \tag{1}$$

The construction is purposeful; we can operate on a object of immense size, $\langle LT(I)\rangle$, while restricting ourselves to a finite number of generators. This makes the theoretical exploration less complex due to the flexibility of $\langle LT(I)\rangle$). It also ensures that giving instruction to a computer becomes possible as we have eliminated the need to find an infinite ideal. Instead, we find the finite generating set.

The next question to answer is how to construct Gröbner bases to operate upon. Since these are useful for computational purposes, an algorithm for how to compute them will be discussed and proven. Before that, it is important to have an understanding of some key properties we can leverage from this basis.

3.3.1 Properties Of Gröbner Bases

The order of the dividing polynomials affects the outcome when performing multivariate division. This is problematic when establishing deterministic outcomes. By fixing a lexicographical order, Gröbner bases guaurantee a unique remainder.

Proposition 3.8. *Let $I \subseteq k[x_1, \ldots, x_n]$ be an ideal and let $G = \{g_1, \ldots, g_t\}$ be a Gröbner basis for I. Then given $f \in k[x_1, \ldots, x_n]$, there is a unique $r \in k[x_1, \ldots, x_n]$ with the two following properties:*

1. *No term of r is divisible by any of $LT(g_1), ..., LT(g_t)$.*

2. *There is a $g \in I$ such that $f = g + r$.*

Proof. Consider the division algorithm $f = q_1 g_1 \cdots + q_n g_n + r$. Every $LT(g_t)$ is already divided out. In other words, if r had been divisible by the lead term there would have been a corresponding q_t to engulf it. That proves (1). (2) is solved by considering the fact that we are operating with Gröbner bases; we must leverage our ability to show that $\langle LT(I) \rangle = \langle LT(g_1), \ldots, LT(g_t) \rangle$. To prove uniqueness, assume that the remainder is non-unique and use lemma 3.2 to disprove that assumption. In other words: assume that $f = g + r = g' + r'$ follows our proposition. Rearranging, $r - r' = g - g' \in I$. Now, using the special property of the basis, $LT(r - r') \in \langle LT(I) \rangle = \langle LT(g_1), \ldots, LT(g_t) \rangle$. Since $LT(r - r')$ is on $LT(g_i)$, then it is also divisible by that same $LT(g_i)$. Using the first property, $r - r'$ cannot be divided by $LT(g_i)$, so in order for the division to succeed, $r - r' = 0$. This contradicts the assumption that r was non-unique. □

Another important property is how easily one can verify if a function is in an ideal. The membership problem is solved by dividing f by G and checking if the remainder is zero. If it is, then the function belongs to the ideal. This membership checks concrete nature is useful for computational applications.

3.3.2 Buchberger's Criterion

The following definitions and lemmas are preparatory for algorithmically finding Gröbner bases. The algorithm and my implementation using Macaulay2 can be found in section 4.2.

Definition 3.9. We will write $\overline{f}^F$ for the remainder on division of f by the ordered s-tuple $F = (f_1, \ldots, f_s)$. If F is a Gröbner basis for $\langle f_1, \ldots, f_s \rangle$, then we can regard F as a set.

Now that we have a concise way to refer to the remainder of a division by two arbitrary parameters, our dialog around the subject will be less cluttered. A deterministic approach to cancelling leading terms would help in the construction of our algorithm. Below is a definition that introduces the tool we will use to craft the algorithms kernel.

Definition 3.10. Let $f, g \in k[x_1, \ldots, x_n]$ be nonzero polynomials.

1. If multideg$(f) = \alpha$ and multideg$(g) = \beta$, then let $\gamma = (\gamma_1, \ldots, \gamma_n)$, where $\gamma_i = max(\alpha_i, \beta_i)$ for each i. We call x^γ the least common multiple of $LM(f)$ and $LM(g)$, written $x^\gamma = \text{lcm}(LM(f), LM(g))$.

2. The S-polynomial of f and g is the combination

$$S(f, g) = \frac{x^\gamma}{LT(f)} \cdot f - \frac{x^\gamma}{LT(g)} \cdot g \tag{2}$$

Let us explore the definition provided by the book [DAC10]. Our numerators share the extracted least common multiple. The other interesting part is that the incoming parameterized functions, f, g, are being divided by their own leading term. It seems as though the least common multiple ensures that a cancellation occurs. In other words, this S-polynomial could be an engine for reducing the degree of polynomials. The next lemma showcases an exciting part of mathematics. Namely, validating intuition.

Lemma 3.11. *Suppose we have a sum $\sum_{i=1}^s p_i$ where multideg$(p_i) = \delta \in \mathbb{Z}_n^{\geq 0}$ for all i. If multdeg$(\sum_{i=1}^s p_i) < \delta$, then $\sum_{i=1}^s p_i$ is a linear combination with coefficients in k, of the S-polynomials $S(p_j, p_l)$ for $i \leq j, l \leq s$. Furthermore, each $S(p_j, p_l)$ has multidegree $< \delta$.*

Cox and his book [DAC10] turns our attention to an interesting idea. The assumption made in the lemma relies heavily on the properties of the s-polynomial. The written proof exists in the book in chapter 2.6 as Lemma 5. The intuition to extract from the proof can be described as follows.

Consider an addition of polynomials, all of them equal in multidegree, summing to a lesser multidegree. If you can find an example of that, this lemma ensures that you can rewrite that addition of polynomials as an addition of s-polynomials. Using this fact, the criteria for admission into the Gröbner basis can be created.

Theorem 3.12 (Buchberger's Criterion)**.** *Let I be a polynomial ideal. Then a basis $G = g_1, \ldots, g_t$ of I is a Gröbner basis of I if and only if the remainder of the S-polynomial constructed using g_i, g_j $(i \neq j)$ by G is zero. Or in other words, if: $\overline{S(g_i, g_j)}^G = 0$ for all $(g_i, g_j) \in G$ where $(i \neq j)$.*

Here, the idea behind finding Gröbner bases is shown. All one has to do to find a Gröbner basis is to simply check all the g_ts against each other using the constructed S-polynomial. If a nonzero remainder is found, a new g_{t+1} is added to the basis being generated. If the remainder is zero, then we are positive that we have a Gröbner basis.

Proof. $\Rightarrow$ If G is a Gröbner basis, then we are trying to prove the remainder $\overline{S(g_i, g_j)}^G$ is 0. Leveraging the problem of membership, we can recall that the remainder would be 0 for all members of the Gröbner basis. Since all of the S-polynomials are on I, the remainder is 0.

$\Leftarrow$ We start by letting $f \in I$ be nonzero. Prove that

$$LT(f) \in \langle LT(g_1), \ldots, LT(g_n) \rangle$$

This is done by considering the following functions:

$$f = \sum_{i=1}^{t} h_i g_i, h_i \in k[x_1, \ldots, x_n] \tag{3}$$

$$\text{multdeg}(f) \leq \max\Big\{\text{multdeg}(h_i g_i) \;\Big|\; h_i g_i \neq 0\Big\} \tag{4}$$

$$\delta = \max_{1 \leq i \leq t}\Big\{\text{multdeg}(h_i g_i)\Big\} \tag{5}$$

By equations 4 and 5 we can arrive at the fact $\text{multdeg}(f) \leq \delta$. In addition to this, we will also minimize the δ. Since we have that $\text{multdeg}(f) \leq \delta = \min \text{multdeg}(f)$, there needs to be two separate cases.

1. $\text{multdeg}(f) = \delta$. Then, $LT(f)$ is divisible by $LT(g_i)$ since the Gröbner base produces all leading terms in the ideal. In other words:

$$LT(f) \in \langle LT(g_1), \ldots, LT(g_n) \rangle$$

2. multdeg$(f) < \delta$.

Proving multdeg$(f) < \delta$ requires rigor. Let us first state an assumption which we will operate upon: The arrangement of $f$provides a minimal δ which cannot be reduced further. If that assumption can be broken by finding a smaller δ, then we have proven that δ will always be the multdeg$(f) = \delta$ case.

We do this by considering an f with a minimum δ. Split the sum into its two components and then extract the leading term for each iteration of i.

$$\begin{aligned} f &= \sum_{\text{multdeg}(h_i g_i)=\delta} h_i g_i + \sum_{\text{multdeg}(h_i g_i)<\delta} h_i g_i \\ &= \sum_{\text{multdeg}(h_i g_i)=\delta} LT(h_i) g_i + \sum_{\text{multdeg}(h_i g_i)=\delta} (h_i - LT(h_i)) g_i + \sum_{\text{multdeg}(h_i g_i)<\delta} h_i g_i \end{aligned} \tag{6}$$

Clearly the sum of functions with multidegree less than δ cannot sum to something greater than δ. Equally so, the degree of the sum of functions where a leading term has been removed will be lesser than the multidegree of the functions to begin with. So the second and third sums of the expanded equation 6 will be less than δ. So, we can focus our attention solely on the monomials of the sum that includes the leading term. If we can prove that the monomials in the sum of the leading terms is less than δ, then the claim of δ minimal property is contradicted. This is where our well studied S-polynomial is used.

Since the sum of all the sums in equation 6 is less than δ, our leading term sum is forced to be less than δ. But this was the exact definition from lemma 3.11. So we can rewrite the sum of leading terms as a combination of S-polynomials. But the consequence of using S-polynomials and lemma 3.11 was precisely that the summands have a lesser multidegree. Then, we have shown that f can be rewritten as a sum of polynomials, all with multidegree less than that minimal δ. Thus, we have contradicted the initial claim.

□

This criterion is the primary tool for the algorithm that is going to be constructed and programmed. By having a concrete way of checking if a function is part of a Gröbner basis, an algorithm can be constructed.

4 Algorithms in Macaulay2

Before this thesis, the computer language Macaulay2 was unknown to me; familiarization was necessary. One of the books I was reading ([LM21]) provided an introductory section to Macaulay2, so to understand the language, I read and recreated given examples in Macaulay2. These can be found in the appendix A.2 or at the GitHub link [Eri26]. These examples were solid introductions to important concepts such as how to create an ideal or Gröbner bases while also providing a base for how write code in the language. After getting a sense for syntax and structure, the next step was to implement an algorithm.

4.1 Division Algorithms

Division algorithms have been of interest for mathematicians for a long time. Basic arithmetic with additions, subtraction, and multiplication can be constrained to $\mathbb{Z}$, the integers, while the introduction of division necessitates an entirely different way of interacting with numbers, $\mathbb{Q}$, the rationals. Thus, a mathematical programming language must be equipped with ways to handle division. Macaulay2 has various ways of dividing, and offers built-in support for polynomial division. However, the division algorithm seemed like a fitting first task.

4.1.1 Univariate Division Algorithm

Algorithm 1 Univariate Division Algorithm

Require: Functions p, g
Ensure: Quotient q and remainder r

1: $r \leftarrow p$
2: $q \leftarrow 0$
3: **while** $r \neq 0$ AND $\deg(r) < \deg(g)$ **do**
4: $\quad t \leftarrow \mathrm{LT}(r)/\mathrm{LT}(g)$
5: $\quad q \leftarrow q + t$
6: $\quad r \leftarrow r - gt$
7: **end while**
8: **return** (q, r)

The first algorithm I tried writing was the univariate polynomial division, see code in listing 1. The algorithm is straight forward. Start by extracting the leading term of the numerator and divide it by the denominator. Add the quotient to the

result and subtract the quotient times the denominator from the numerator. Repeat until the termination condition is hit: either the numerator is zero, and there is nothing to divide, or the degree of the denominator exceeds that of the numerator, and division is not possible. Theoretically and practically, it was simple.

4.1.2 Multivariate Division Algorithm

The idea behind the multivariate division algorithm is similar to the univariate one. Divide a polynomial p by an array of polynomials $g_1, \ldots, g_s$. If the leading term of p cannot be divided by the leading term of g_1, move on to the next until a g_i that can divide p is found. If no leading term of $g_1, \ldots, g_s$ can divide the leading term of p, then the leading term p is subtracted from itself and the algorithm marches to the next polynomial.

Algorithm 2 Multivariate Division Algorithm

Require: Functions $p, g_1, \ldots, g_n$
Ensure: Quotients q_s and remainder r

```
1:  r ← 0
2:  f ← p
3:  q_s ←
4:  while f ≠ 0 do
5:      i ← 0
6:      divisionOccurred ← false
7:      while i ≤ n AND divisionOccurred = false do
8:          if LM(f) | LM(g_i) = 0 then
9:              t ← f/g_i
10:             q_i ← q_i + t
11:             f ← f − g_i t
12:             divisionOccurred = true
13:         else
14:             i ← i + 1
15:         end if
16:     end while
17:     if divisionOccurred = false then
18:         r ← r + LT(f)
19:         f ← f − LT(f)
20:     end if
21: end while
22: return (q_s, r)
```

A challenge presented itself when writing this algorithm. Division of more than

one variable returned a fractions of rings. This was problematic as the return (q_s, r) need to be members of the same ring as p. To solve this, a function that decomposed the polynomials into lists of monomials, coefficients, and exponents was created. The function would generate a list of the exponents taking the ring elements into account. So, as an example, if the ring is $Q[x, y, z]$, the polynomial $x^2 * z$ would be decomposed into $\{2, 0, 1\}$. By creating these lists and subtracting the exponents from each other, division occurred. Transforming this list into a polynomial element using the generators of the ring would require raise the elements to the exponent in the listed order. However, generators of rings, and the rings themselves are different in Macaulay2. When reapplying the exponential list to the generators, any reference to the ring itself is lost. The list of generators is simply a list, decoupled from the ring reference.

Instead of recreating a way to divide monomials or using fractions of rings, Macaulay2 supports division that ensures the result remains in the ring. Thus, the algorithm was ready to be implemented. By first checking if the modulo between the lead monomial of p and g_i would be 0, we can ensure that division within the ring would succeed.

4.2 The Buchberger Algorithm

Buchbergers Algorithm is the algorithm that finds a complete Gröbner basis. The Buchberger criterion so carefully proven in the previous sections of this thesis is fully leveraged here. Due to this, the algorithm itself is easy to implement as all the work behind the algorithm is in proving that the criterion is sufficient to generate the Gröbner basis. By pairing all the functions $p_1, \ldots, p_n$, calculating the S-polynomial for each, and performing the subsequent multivariate polynomial division one can successfully find a Gröbner basis.

Proof. There are two parts to the proof. First, we must prove that the generated set, G, actually is a Gröbner basis, and second we must prove that the algorithm terminates. We can prove that G is a Gröbner basis by first noting that that $G \subseteq I$ at every point of the algorithm. Since we are solely using functions that generate I, any remainder from division will be on I as well. So by extending G with the remainder, $G \subseteq I$ holds. The fact that the ideal G is a Gröbner basis comes from theorem 3.12. The algorithm terminates when $\overline{S(g_i, g_j)}^{\hat{G}} = 0$ for all pairs which was the condition for the basis to be a Gröbner basis.

Algorithm 3 Buchberger Algorithm

Require: Functions $P = p_1, \ldots, p_n$ that generate I.
Ensure: A Gröbner basis $G = g_1, \ldots, g_s$ for I, with $P \subseteq G$

$G \leftarrow P$
$\hat{G} \leftarrow \{\}$
while $G \neq \hat{G}$ **do**
    $\hat{G} \leftarrow G$
    for every pair $\{g_i, g_j\}$, $g_i \neq g_j$ in $\hat{G}$ **do**
        $r \leftarrow \overline{S(g_i, g_j)}^{\hat{G}}$
        if $r \neq 0$ **then**
            $G \leftarrow G \cup \{r\}$
        end if
    end for
end while
return G

The remaining part is to prove that the algorithm terminates. Consider the remainder,r, that is generated from the division of members on G. From lemma 3.2, we can note that the $LT(r)$ is not contained in $\hat{G}$, but is added to G. Thus, we can state that the $\langle LT(\hat{G})\rangle \subseteq \langle LT(G)\rangle$ since $LT(r)$ is solely in $LT(G)$. Since we are dealing with an ascending chain of ideals in $k[x_1, \ldots, x_n]$, the ascending chain condition can be used (see chapter 2.5 theorem 7 in [LM21]). By the ascending chain condition, there exists an $N \geq 1$ such that $I_N = I_{N+1} = I_{N+2} = \cdots$. Thus, after N steps of the algorithm, further steps will not add any r, and the algorithm will terminate. □

This algorithm was actually implemented *before* I properly understood Gröbner bases and S-polynomials. This is a fascinating aspect of the separation between proving algorithms and their implementations. Since mathematicians take great care in proving that the algorithm is correct, the implementee can trust the algorithm's correctness and tell the machine what to do. In a way, the person instructing the machine how to inform the tasks are they themselves a machine to the mathematician.

4.3 Solving Systems of Polynomials

The polynomial solution was by far the most involved algorithm to construct from a Macaulay2 standpoint. At this point, I was feeling much more comfortable with

the language and started by following examples from the book [LM21], found under listing 5 and 6.

4.3.1 Book Examples of Polynomial Solutions

To start off, a monovariable polynomial is considered. The quotient ring is generated, and the monomial basis is found. The implementation of this algorithm was done with as many built in functions that Macaulay2 has as possible, so most of the complexity is hidden. One important note is that the "basis" function of Macaulay2 generates elements outside the ring, the "lift" function is used to bring them back into the ring. Consider the fact that $\frac{4}{2}$ can be "lifted" from $\mathbb{Q}$ to $\mathbb{Z}$. After this, we find the Q matrix associated with Z, or M_z^Q. The first step is to find the quotient of the ring by the ideal and then find the basis of the quotient. Then, loop over the basis and find the modulo of the basis times the polynomial by the ideal. The remainder of this operation will contain coefficients that correspond to the basis. These coefficients are extracted and put into a matrix.

Algorithm 4 SolveQMatrix

Require: A ring,R, a zero-dimensional ideal, I, and the polynomial to evaluate p .
Ensure: The matrix, M_p^Q.

$q \leftarrow R/I$
$b_1, \ldots, b_n \leftarrow$ basis q
rows $\leftarrow \{\}$
for b_i in $b_1, \ldots, b_n$ **do**
    $r \leftarrow (p \cdot b_i) \mod I$
    rows $\leftarrow$ rows $\cup$ { coefficients(r) from $b_1, \ldots, b_n$}
end for
$M \leftarrow matrix$(rows)
return M

The second book example, (listing 6), considered the polyvariable case. At this point I realized that a dedicated function to solving M^Q would be necessary. The logic used in listing 5 was refactored to its own function. After testing to ensure that they returned the same result, implementing the second example was easier. Solving the M^Q became a function call, send the ring, ideal, and the monomial of interest and recieve the matrix as a result. Solving the eigenvalues uses the provided functions, so finding the solution was no problem.

4.3.2 Solving a System of Polynomial Equations

Algorithm 5 Solution of a system of polynomial equations

Require: A non-zero, zero-dimensional ideal, $I = \langle p_1, \ldots, p_s \rangle$ in $\mathbb{R}[x_1, \ldots, x_n]$
Ensure: The points in $\mathbf{V}_{\mathbb{C}}(I)$
1: $q \leftarrow \mathbb{R}[x_1, \ldots, x_n]/I$
2: $b_1, \ldots, b_l \leftarrow$ basis q
3: eigenList $\leftarrow \{\}$
4: **for** $i = 1$ to n **do**
5: $\quad Q \leftarrow$ solveQMatrix(R, I, x_i)
6: $\quad$ eigenList $\leftarrow$ eigenList $\cup$ eigenvalues of Q.
7: **end for**
8: Compute the sets:
9: $X_{\mathbb{R}} = \{\hat{x} \in \mathbb{R}^n : \hat{x}_i \in \text{eigenList}_i, i = 1, \ldots, n \text{ and } p_j(\hat{x}) = 0, j = 1, \ldots, s\}$
$X_{\mathbb{C}} = \{\hat{x} \in \mathbb{C}^n : \hat{x}_i \in \text{eigenList}_i, i = 1, \ldots, n \text{ and } p_j(\hat{x}) = 0, j = 1, \ldots, s\}$
10: **return** $\mathbf{V}_{\mathbb{C}}(I) = X_{\mathbb{C}}$ and $\mathbf{V}_{\mathbb{R}}(I) = X_{\mathbb{R}}$

The book [LM21] gives an algorithm to solve a variety. Starting off, for a ring $R[x_1, \ldots, x_n]$, all the $M_{x_i}^Q$ had to be found.

This, along with finding eigenvalues were part of previous excercises and therefore already solved. However, when calculating the sets or constructing the candidate points to plug back into the initial functions difficulties arose. When the function evaluates the candidate points over $p_1, \ldots, p_s$, to see which yield 0, a slight problem occurs. The exact cause is unknown, but since the ring is in $\mathbb{R}$, the floating point errors occur and slight variations can cause duplication. In order to solve this issue, there is additional code that filters almost duplicate solutions. This filter works by ensuring that the sum of the distance between all the points is lesser than a defined epsilon.

5 Discussion

There are two distinct parts of this thesis to consider: the construction of an algorithm and the execution of an algorithm. They were not weighed equally. This section considers the imbalance between the two and what they offered as part of a thesis surrounding mathematics.

5.1 Mathematical Proof and Knowledge

Building an algorithm requires intimate knowledge of underlying mathematical concepts as well as a familiarity with proof construction. While background knowledge was gathered for affine varieties, and more generally algebraic geometry, these were less significant to the scope of the thesis. Instead, these concepts justified the exploration of Gröbner bases. Even then, theory surrounding the *usage* of Gröbner bases was limited. Instead, a majority of the theoretical knowledge and proofs found in the thesis concern Gröbner bases, specifically the construction of the Buchberger criterion. The criterion is the motor behind the Buchberger algorithm, used to construct Gröbner bases, and provided insight into how algorithms are created.

Singling in on the Buchberger criterion left other parts of the mathematical domain unexplored, especially considering Gröbner bases' usefulness. The elimination properties of the bases and any application to algebraic geometry or robotics were left unexplored. Both the topics are interesting in their own regard.

5.2 Algorithm Implementation

The traditional mathematical method of proving theorems was less important in the creation of this thesis. Instead the primary focus was directed towards the implementation of algorithms. While a part of algorithms is transforming an idea to something concrete, another is to implement the algorithm itself. For a bachelor's thesis, the implementation was appropriate. Familiarizing oneself with the process of algorithm construction was of importance, but exploring the implementation was more fruitful from a learning aspect. Abstract algebra can be confusing, and being forced to instruct a machine to execute the algorithm increases understanding of the underlying mathematics. Learning a computer language where ideals are an integral reason to its creation provides deeper insight into ideals themselves. To use ideals and rings, one must have a working knowledge of them.

Through implementation, the balancing act between the amount of work that is necessary to create these algorithms and keeping the implementation simple can be appreciated. The Buchberger algorithm is reduced to concrete steps, such as finding S-polynomials 3.10, but proving that the Buchberger criterion is sufficient for finding a Gröbner basis. In contrast, the algorithm that finds the solution to polynomial systems (listing 4) provides guidelines and is flexible in its interpretation; it shifts more responsibility to the implementee.

All in all, insight to both sides of algorithm construction was gained. Both sides provide challenges in their own right and exploring these challenges while deepening my algebraic knowledge was great

A Macaulay2 source files

A link to all the source files can be found at the github link [Eri26] under the m2 folder.

A.1 Algorithms

Listing 1: A Univariate Division Algorithm

```
univariatePolynomialDivision = (p,g) -> (
    q := 0;
    r := p;
    while (r != 0_R and degree(g) < degree(r)) do (
        t := leadTerm(r) / leadTerm(g);
        q = q+t;
        r = r-t*g;
        );
    (q,r)
    )

runUnivariateTest = () -> (
    R = QQ[x];
    p := x^6 - 1;
    g := x^4 - 1;

    (resultQ, resultR) := univariatePolynomialDivision(p,g);

    print(resultQ == x^2);
    print(resultR == x^2 - 1);
)
```

The multivariate division algorithm is loaded into other scripts under the alias algorithm123.m2. This is due to the fact that the pseudocode found as algorithm 1.2.3 [LM21]. LeadingTermDivision is a failed attempted at dividing the leading terms, but is kept to display progress.

Listing 2: The Multivariate Division Algorithm.

```
multivariatePolynomialDivision = (p, gs) -> (
    r := 0_(ring p);
    f := p;
    qs := for i from 0 to #gs-1 list 0_(ring p); --generating
        dummy variable for result
```

```
5
6       while f != 0_(ring p) do (
7           i := 0;
8           divisionOccured := false;
9           --print "Outer  loop";
10
11          while (i < #gs and divisionOccured == false) do (
12              --print "Inner loop";
13              lmf := leadMonomial f;
14              lmg := leadMonomial (gs#i);
15
16              if (lmf % lmg == 0) then (
17                  monomialQuotient := lmf // lmg;
18
19                  coefficientQuotient := leadCoefficient f /
                        leadCoefficient gs#i;
20
21                  t := coefficientQuotient * monomialQuotient;
22
23                  qs = replace(i, qs#i + t, qs);
24                  f = f - t*gs#i;
25
26                  divisionOccured = true;
27                  --print "we were divisible";
28              )
29              else (
30                  i = i+1;
31
32                  --print "not divisible, function further along";
33              )
34          );
35
36          if divisionOccured == false then (
37              r = r + leadTerm(f);
38              f = f- leadTerm(f);
39              --print "division did not occur, subtracting leading
                    term and continuing";
40          );
41      );
42      return (qs, r)
43  )
44
```

```
45 -- This below code is useless, it was an attempt to get around the
       fact that i couldnt devide with the leadTerm.
46 -- However, I needed to seperate out the divisions as well as use
       "//" instead of "/"
47 leadingTermDevision = (p,g) -> (
48     lmP := leadMonomial(p);
49     lmG := leadMonomial(g);
50     lcP := leadCoefficient(p);
51     lcG := leadCoefficient(g);
52     exponentsP := flatten exponents lmP;
53     exponentsG := flatten exponents lmG;
54
55     divisible := all(#exponentsP, i -> exponentsP#i >=
           exponentsG#i); --checking that all exponents are greater or
           equal
56     result := 0_(ring p);
57
58     if divisible then(
59         coefficientResult := lcP / lcG;
60
61         exponentsQ := for i from 0 to (#exponentsP-1) list
               (exponentsP#i - exponentsG#i); --doing the division
62
63         ringVariables := toList gens ring p;
64
65         Q := product(#ringVariables, i -> ringVariables#i ^
               (exponentsQ#i)); --generating the the polynomial
66
67         result = coefficientResult * Q;
68      );
69
70     return (divisible, result);
71  )
72
73  runMultimultivariateTest = () -> (
74      R = QQ[x,y,MonomialOrder=>Lex];
75      p := -x^2*y-x^2+x*y+y^3;
76      g1 := x*y-x+y-1;
77      g2 := x+y+1;
78
79      (resultG1, resultRemainder1) :=
           multivariatePolynomialDivision(p,{g1,g2});
```

```
80      (resultG2, resultRemainder2) :=
            multivariatePolynomialDivision(p,{g2,g1});
81
82      print "-----------{g1,g2}------------";
83      print resultG1;
84      print resultRemainder1;
85      print (resultG1#0 == -x+4);
86      print (resultG1#1 == -2*x+5);
87      print (resultRemainder1 == y^3-9*y-1);
88
89      print "-----------{g2,g1}------------";
90      print resultG2;
91      print resultRemainder2;
92      print (resultG2#1 == 0);
93      print (resultG2#0 == -x*y-x+y^2+3*y+1);
94      print (resultRemainder2 == -4*y^2-4*y-1);
95  )
```

The Buchberger algorithm is used to find Gröbner bases. The function FindXDelta was later replaced with the native least common denominator function to reduce clutter and error.

Listing 3: Buchbergers Algorithm.

```
1   load "algorithm123.m2";
2
3   buchberger = ps -> (
4       g := ps;
5       gHat := {0_(ring  ps#0)};
6
7       while g != gHat do (
8           gHat = g;
9
10          -- generating all the possible pairs by looping through i
                and then from j -> end
11          polynomialPairs := flatten for i from 0 to #gHat-1 list
                for j from i+1 to #gHat-1 list (gHat#i, gHat#j);
12
13          i := 0;
14          while i < #polynomialPairs do (
15              p := (polynomialPairs#i)#0;
16              q := (polynomialPairs#i)#1;
17
```

```
18              xDelta := lcm(leadMonomial(p), leadMonomial(q));
19
20              s := xDelta // leadTerm(p) * p - xDelta // leadTerm(q)
                    * q;
21
22              (qs, r) = multivariatePolynomialDivision(s, gHat);
23
24              if r != 0_(ring ps#0) then (
25                  g = append(gHat,r);
26                  );
27
28              i = i+1;
29              );
30          );
31      print g;
32      return g;
33      )
34
35  testAlgorithm = () -> (
36      R = QQ[x,y, MonomialOrder => GLex];
37
38      p1 := x^2 + y;
39      p2 := x^4 + 2*x^2*y + y^2 + 3;
40
41      myBasis := gb ideal (buchberger({p1, p2}));
42      correctBasis := gb ideal(p1, p2);
43      areTheyEqual := myBasis == correctBasis;
44
45      print "My basis:";
46      print myBasis;
47
48      print "Built-in basis:";
49      print gens correctBasis;
50
51      print "Are they equal?";
52      print areTheyEqual;
53      )
54
55  --again, this is not needed and M2 implements LCM.
56  findXDelta = (p,q) -> (
57      alpha := flatten exponents leadMonomial(p);
58      gamma := flatten exponents leadMonomial(q);
```

```
59
60      delta := for i from 0 to (#alpha-1) list
            (max(alpha#i,gamma#i)); --getting the max of each variables
            exponent
61
62      ringVariables := toList gens ring p;
63      xDelta := product(#ringVariables, i -> ringVariables#i ^
            (delta#i)); -- reconstructing the variable
64
65      return xDelta
66      )
67
68  testGamma = () -> (
69      R := RR[x,y, MonomialOrder => GLex];
70      p := x^3*y^2-x^2*y^3+x;
71      q := 3*x^4*y +y^2;
72
73      xDelta := findXDelta(p,q);
74      xDeltaShouldBe := x^4*y^2;
75
76      print "-----------gamma results------------";
77      print xDelta;
78      print xDeltaShouldBe;
79      print("the result is " | toString(xDelta == xDeltaShouldBe))
80      )
```

Refered to as algorithm 1.6.1 as the pseudocode us found under the same name in [LM21]. The refactored QMatrix function can be found here, but was initially created using the preparatory examples listing 5 and listing 5.

Listing 4: Solution to Systes of Polynomial Equations

```
1  solveSystemOfPolynomialEquations = (theRing, theIdeal) -> (
2      varsToLoopOver := gens theRing;
3
4     -- get the list of eigenvalues
5      eigenList := for var in varsToLoopOver list (
6          qMatrix := solveQMatrix(theRing, theIdeal, var);
7
8           --ensure that the eigenvector can be solved and save it
                to the list
9           toList eigenvalues(sub(qMatrix,CC))
10         );
```

```
11
12      -- use fold to go through all elements in the eigenList
            outwards
13      candidatePoints := fold((L1, L2) -> (
14              --use table to create a table of all the pairs and
                    flatten down
15              flatten table(L1, L2, (a, b) -> (
16                      --if we are past first instance, we append
                            instead of create a list
17                      if instance(a, List) then a | {b} else {a, b}
18                      )
19                  )
20              ),
21          eigenList);
22
23      epsilon := 1e-6;
24      functionList := flatten entries gens theIdeal;
25
26      validSolutions := select(candidatePoints, pt -> (
27              -- ensure that x_n maps to the the nth point
28              subMap = apply(#varsToLoopOver, i -> varsToLoopOver#i
                    => pt#i);
29              -- sub in the values
30              all(functionList, p -> abs(sub(p, subMap)) < epsilon)
31              ));
32
33      -- something is wrong with my solveQMatrix which results in
            cluttered returns.
34      -- so for now just to check if it works, we remove the
            duplicates
35      uniqueSolutions := {};
36      scan(validSolutions, v -> (
37              if not any(uniqueSolutions, u -> (
38                      dist := sqrt(sum(0..#v-1, i -> abs(v#i -
                            u#i)^2));
39                      dist < epsilon
40                      )) then uniqueSolutions =
                            append(uniqueSolutions, v);
41              ));
42
43      uniqueSolutions
44      )
```

```
45
46
47  testAlgorithm2D = () -> (
48      R := CC[x, y];
49      I := ideal(x^2 + y^2 - 5, x*y - 2);
50
51      myResults := solveSystemOfPolynomialEquations(R, I);
52      solution := {{-2, -1}, {-1, -2}, {2, 1}, {1, 2}};
53
54      print ("The 2D test output: " | toString  myResults);
55
56      print ("Verified solution: " | toString solution);
57  )
58
59  testAlgorithm4D = () -> (
60      R4 := CC[x1, x2, x3, x4];
61
62      p1 := x1^2 + x2^2 - 2;
63      p2 := x1 - x2;
64      p3 := x3^2 + x4^2 - 13;
65      p4 := x3*x4 - 6;
66      I4 := ideal(p1, p2, p3, p4);
67
68      results4D := solveSystemOfPolynomialEquations(R4, I4);
69
70      if #results4D == 8 then (
71          print "SUCCESS: Found all 8 points in 4D space.";
72      ) else (
73          print ("FAILURE: Expected 8 points, but found " |
                #results4D);
74      );
75  )
76
77  solveQMatrix = (theRing, theIdeal, thePolynomial) ->(
78      q := theRing / theIdeal;
79
80      basisList := flatten entries basis q;
81      polyBasis := for b in basisList list lift(b,theRing); --basis
            to R
82
83      rows := for b in polyBasis list (
84          currentPoly := thePolynomial * b;
```

```
85          r := currentPoly % theIdeal;
86
87          coeffsMatrix := last coefficients(r, Monomials =>
                polyBasis);
88          flatten entries coeffsMatrix
89          );
90
91      return transpose matrix rows;
92      )
93
94  print "-------2D CASE-------";
95  testAlgorithm2D();
96  print "------4D CASE---------";
97  testAlgorithm4D();
98  print "-------DONE-------";
```

A.2 Examples

Listing 5: Example 1.6.12 [LM21]

```
runExample = () -> (
    R = RR[z];

    g := z^3-3*z^2+2*z;

    I = ideal g;

    q := R / I;

    basisList := flatten entries basis q;
    polyBasis := for b in basisList list lift(b, R); -- ensuring
        that the actualy polynomials exists in R and not only in q

    rows := for b in  basisList list (
        currentPoly = z *lift(b,R);

        r := currentPoly % I;

        --taking #1 here since coefficients returns a bundle of
            want the coefficients, no the monomials
        coeffsMatrix := (coefficients(r, Monomials =>
            polyBasis))#1;
        flatten entries coeffsMatrix
        );

    zmatrix := matrix rows;

    finalZMatrix := zmatrix^2+zmatrix+id_(target zmatrix);
    return finalZMatrix;
)


 runExampleWithFunction = () -> (
    R = RR[z];

    g := z^3-3*z^2+2*z;

    I = ideal g;

```

```
    load "algorithm161.m2";

    zmatrix := solveQMatrix(R, I, z);

    finalZMatrix := zmatrix^2+zmatrix+id_(target zmatrix);
    return finalZMatrix;
)

testFunction = () -> (
    preBuiltMatrix := runExample();
    matrixFromFunction := runExampleWithFunction();

    print "Matrix from the example";
    print preBuiltMatrix;

    print "Matrix using refactored function";
    print matrixFromFunction;
 )
```

Listing 6: Example 1.6.13 [LM21]

```
runExample = () -> (
    load "algorithm161.m2";

    R = RR[x,y];
    f := (y^2+x-3, x^2-4*x*y-y^2-10*y+15);
    I := ideal f;

    regMatrix := solveQMatrix(R,I,1);
    xMatrix := solveQMatrix(R,I,x);
    xyMatrix := solveQMatrix(R,I,x*y);
    yMatrix := solveQMatrix(R,I,y);

    --only care about the x,y individual ones
    xEigens := (eigenvectors sub(xMatrix,CC))#0;
    yEigens := (eigenvectors sub(yMatrix,CC))#0;

    varietyList := for i from 0 to  #xEigens-1 list {xEigens#i,
       yEigens#i};
    print varietyList;
    )
```

Listing 7: Macaulay2 code from 2.1.1 [LM21]

```
R = QQ[x_1..x_4, MonomialOrder => Lex]

p1 = x_1-x_2**2+x_3

p2 = x_1**2+x_2*x_3**2+x_4

I = ideal(p1,p2)

G = gens gb I

polyToDivide = x_1**2+x_2**2+x_3**2+x_4**2-1

rem = polyToDivide%I

print("the rem is: ")
print(rem)
```

Listing 8: Macaulay2 code from 2.2.1 [LM21]

```
R = QQ[x_1,x_2,x_3]

p = (1/2)+x_1^2*x_2*x_3^4+(6/5)*x_1*x_3+2

-- output is a polynomial in R
```

Listing 9: Macaulay2 code from 2.2.2 [LM21]

```
R = frac(QQ[a]/ideal(a^2-2))[x_1,x_2,x_3]

substitute(a^2, R)
-- output is that a^2 = 2

p = x_1 + x_2 + a*x_3
```

Listing 10: Macaulay2 code from 2.3.1 [LM21]

```
R = QQ[z]

p = z^4+3*z^3-3*z^2+3*z-4

pc = -2*z^4+3*z^3-6*z^2+5*z-4

g = z^2+1
```

```
8
9  (q,r)= quotientRemainder(p,g)
10
11 (qc, rc) = quotientRemainder(pc,g)
```

Listing 11: Macaulay2 code from 2.3.2 [LM21]

```
1  R=QQ[c_1,c_2,c_3,c_4, MonomialOrder=>Lex]
2  M = matrix{{1,-2,0,-1},{4,0,7,3}, {7,-6,7,0}}
3  C = matrix{{c_1},{c_2},{c_3},{c_4}}
4  I = ideal(M*C)
5  GI = value toString gens gb I
6  (Ce,Me) = value toString coefficients(GI,
       Monomials=>{c_1,c_2,c_3,c_4})
7
8  Mr = value toString transpose Me
9
10 --powerful way to solve the row reduced form
11 inverse(substitute(submatrix(Mr, {0,1}),QQ))*Mr
```

Listing 12: documentationConditionals.m2 [GS]

```
1  isEven = i -> i % 2 == 0
2
3  oddOrEven = i -> (
4      if isEven i then "even"
5      else "odd")
```

References

[DAC10] Donal B O'Shea David A Cox, John Little. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer Publishing Company, third edition, November 2010.

[DF03] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Wiley, Hoboken, NJ, USA, 3 edition, 2003.

[Eri26] Christian Eriksson. M2 algorithms. Available at https://github.com/Skalmanen/MM6010, 2026.

[Gal25] Jean H. Gallier. Basics of affine geometry. In Jean H. Gallier, editor, *Geometric Methods and Applications for Computer Science and Engineering*, pages 7–63. Springer, 2025. Chapter 2 from online course materials. URL: https://www.cis.upenn.edu/~cis6100/geombchap2.pdf.

[GS] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at https://macaulay2.com/.

[LM21] Antonio Tornambè Laura Menini, Corrado Possieri. *Algebraic Geometry for Robotics and Control Theory*. WORLD SCIENTIFIC (EUROPE), 2021. arXiv:https://www.worldscientific.com/doi/pdf/10.1142/q0308.

[Wei70] Louis M. Weiner. *Introduction to Modern Algebra*. Harcourt, Brace & World, New York, NY, USA, 1970.