Universidad Nacional de Córdoba

Facultad de Matemática, Astronomía, Física y Computación

Implementación de bases de Gröbner no conmutativas en C++ con un poquito de paralelismo

por

Iván Ariel Renison

Presentado ante la FACULTAD DE MATEMÁTICA, ASTRONOMÍA, FÍSICA Y COMPUTACIÓN como parte de los requerimientos para la obtención del grado de Licenciado en Ciencias de la Computación de la

UNIVERSIDAD NACIONAL DE CÓRDOBA

Marzo, 2025

Directores:

Cristian Vay

Miguel Maria Pagano

Resumen

Esta tesis presenta una implementación en C++ para el cálculo de las llamadas bases de Gröbner no conmutativas. Ya existían previamente algunas implementaciones de dicho cálculo, pero la que se presenta en esta tesis es la primera en C++ y además incluye un poquito de paralelismo.

En los primeros capítulos se proporciona una explicación matemática completa del tema, para que la tesis sea autocontenida y en los restantes se explica la librería de C++ donde se implementó el cálculo (junto con las estructuras y algoritmos auxiliares necesarios) y se la compara con una de las implementaciones anteriores.

Abstract

This thesis presents an implementation in C++ for the computation of the so called noncommutative Gröbner bases. Some implementations of this computation already exist previously, but the one presented in this thesis is the first one in C++ and it also includes a little bit of parallelism.

In the first chapters a complete mathematical explanation of the subject is provided, so that the thesis is self contained, and in the remaining chapters the C++ library where the calculation was implemented (together with the necessary structures and auxiliary algorithms) is explained and compared with one of the previous implementations.

Clasificación (MSC2010)

68W30: Computación simbólica y computación algebraica

Palabras clave

- Polinomios no conmutativos
- Bases de Gröbner no conmutativas
- Algoritmo de Buchberger no conmutativo
- Algoritmo F4 no conmutativo
- Sistemas de reescritura
- Implementación

Reconocimientos

A continuación una lista de quienes ayudaron en algo con este trabajo:

- Mis directores Cristian Vay y Miguel Maria Pagano ayudaron un montón, tanto en la parte de aprender los temas, como en la parte de hacer el código y en la parte de escribir.
- Lucía Martinez Gavier leyó la tesis e hizo bastantes sugerencias.
- Álvaro Roy Schachner me ayudó con cosas de la computadora Atom y además leyó la tesis e hizo bastantes sugerencias.
- Felipe Clariá Dambolena leyó la tesis e hizo bastantes sugerencias.
- Agustín Garcia Iglesias como miembro del tribunal leyó la tesis e hizo algunas sugerencias.
- Ana María Cingolani leyó el resumen y la introducción e hizo algunas sugerencias.

Además se usaron recursos computacionales de la computadora Atom de FAMAF.

Índice general

Capítulo 1
Introducción

Usted posiblemente haya escuchado hablar de los polinomios de varias variables, esos como $5 + 3y - 2xy + x^3y^5$, y que todo el conjunto de los polinomios sobre el cuerpo K y con variables en el alfabeto X se denota $K[X]$. En esos polinomios el producto entre las variables conmuta; por ejemplo, el polinomio xy es igual al polinomio yx. Sin embargo, se pueden considerar otros polinomios en los que el producto entre las variables no conmuta, así que por ejemplo, el polinomio xy es distinto del polinomio yx; pero el producto de las variables con los coeficientes sí sigue conmutando, por ejemplo el polinomio $x3y$ es igual al polinomio $3xy$. Estos nuevos polinomios, llamados polinomios no conmutativos, cuando son sobre el cuerpo K y con variables en el alfabeto X se denotan $K\langle X\rangle$, y consisten en combinaciones lineales de palabras del alfabeto X con coeficientes en K. Algunos ejemplos de polinomios no conmutativos en $\mathbb{Q}\langle x, y, z\rangle$ son los siguientes:

$$\begin{aligned} f_0 &= x, \\ f_1 &= xy + yz, \\ f_2 &= 3xyy - 2xzxy + \frac{4}{3}yzzx. \end{aligned}$$

Notar que $f_1 \neq xy + zy$ ya que el producto es no conmutativo.

Un problema de decisión que existe sobre los polinomios no conmutativos es: dado un conjunto G de polinomios no conmutativos y un polinomio no conmutativo f, decidir si f se puede escribir como combinación lineal de elementos de G con coeficientes en $K\langle X\rangle$. O escrito formalmente:

Problema 1. *Dados $G \subseteq K\langle X\rangle$ y $f \in K\langle X\rangle$, determinar si vale que*

$$\exists n \in \mathbb{N} \cup \{0\}, g_1, ..., g_n \in G, f_1, ..., f_n, f'_1, ..., f'_n \in K\langle X\rangle : f = \sum_{i=1}^{n} f_i g_i f'_i.$$

Acá los f_i y f'_i se llaman cofactores. Por ejemplo con $K = \mathbb{Q}$ y $X = \{x, t, z\}$, si:

$$\begin{aligned} g_0 &= xy + yz, \\ g_1 &= yx + zx, \\ G &= \{g_0, g_1\}, \\ f_0 &= xyx - yyx, \\ f_1 &= xyz + zyx. \end{aligned}$$

Tenemos que para f_0 la respuesta al Problema 1 es afirmativa porque

$$\begin{aligned} g_0 x - y g_1 &= (xy + yz)x - y(yx + zx) \\ &= xyx + yzx - yyx - yzx \\ &= xyx - yyx \\ &= f_0 \end{aligned}$$

Pero para f_1 es negativa porque no hay ninguna forma de escribirlo como combinación lineal de g_0 y g_1 con coeficientes en $\mathbb{Q}\langle x, y, z\rangle$. Más adelante se explicará el código que se hizo que permite a veces responder al Problema 1 y en particular determinar que para f_1 la respuesta es negativa.

El Problema 1 no es decidible, lo que significa que no hay un algoritmo que decide si vale la expresión o no. Sin embargo sí es semi-decidible, lo que significa que hay un algoritmo que en los casos en los que sí vale la expresión termina y en los casos en los que no vale la expresión puede no terminar.

Este problema no es decidible porque el llamado problema de la palabra se reduce a este. En [WWP] se puede ver qué es el problema de la palabra y en la Sección 1.3 de [Mor94] cómo es la reducción. En esta tesis no se abordará el tema.

Para trabajar en algoritmos que aborden el Problema 1 se considera el conjunto denotado como (G) de todos los polinomios no conmutativos que satisfacen ese $\exists$, de forma que el Problema 1 se convierte en determinar si $f \in (G)$. Para responder a esto se usa el siguiente resultado: si $G = \{g_1, \ldots, g_n\} \subseteq K\langle X\rangle$, $f \in K\langle X\rangle$, a y b son polinomios que consisten en una sola palabra y $c \in K$, entonces

$$f \in (G) \Leftrightarrow f + cag_ib \in (G).$$

Teniendo en cuenta este resultado se intenta cancelar los términos de f sumando distintos cag_ib, y si en algún momento se llega a 0 entonces vale que $f \in (G)$ porque el 0 es la sumatoria vacía y porque todas las sumas mantienen la pertenencia a (G).

El problema es que a veces puede pasar que ya no queden más términos para cancelar y no se haya llegado a 0 pero aun así valga que $f \in (G)$. Para solucionar este problema lo que se hace es agregar elementos a G para obtener lo que se llama una base de Gröbner de (G), con la cual siempre que el polinomio está en (G) se puede llegar a 0 cancelando términos. El único problema de esto es que la base de Gröbner no siempre es finita, lo que refleja la no decidibilidad del problema. De cualquier manera, en los casos en los que es infinita, es computacionalmente enumerable, y el poder enumerarla también puede ser útil para abordar el Problema 1.

Para calcular o enumerar las bases de Gröbner hay dos algoritmos destacados, llamados el algoritmo de Buchberger y el algoritmo F4. La diferencia entre los dos algoritmos es que Buchberger va calculando la base de Gröbner de a un elemento por vez mientras que F4 usa herramientas de álgebra lineal para calcular de a muchos elementos al mismo tiempo.

Si la explicación sobre los algoritmos no se entendió no importa porque en el Capítulo 2 se explica bien.

Estos algoritmos, al igual que mucha de la teoría, están inspirados en el caso análogo para polinomios conmutativos y muchos de los nombres de las cosas se usaron primero para el caso conmutativo y después se usó el mismo nombre para el caso no conmutativo. Por ejemplo, "el algoritmo de Buchberger" en otros contextos puede referirse al algoritmo para el caso conmutativo. En esta tesis siempre se estará hablando del caso no conmutativo. Para una explicación del caso conmutativo ver [CLO15].

Existían previamente varias implementaciones del algoritmo de Buchberger en el caso no conmutativo, por ejemplo [CKG24; Dec+24; NCA]. El algoritmo F4 para el caso no conmutativo fue primero adaptado por Xiu Xingqiang en [Xiu12] y está implementado en el sistema algebraico computacional Magma. Luego, Clemens Hofstadler hizo en [Hof20] su propia implementación en Python con SageMath, que es otro sistema algebraico computacional que al mismo tiempo es librería de Python. De acuerdo a mi conocimiento esas son las únicas dos implementaciones de F4 para el caso no conmutativo.

El objetivo de este trabajo fue implementar en C++ ambos algoritmos y si es posible con un poco de paralelismo. La implementación en C++ se logró, aunque para el algoritmo F4 hay una optimización que ayuda bastante que quedó pendiente. El paralelismo también se logró, pero solo para una parte y para la parte de la optimización que quedó pendiente tampoco se hizo nada de paralelismo. Por eso es que tiene un "poquito" de paralelismo.

Por la optimización que quedó pendiente en los casos muy grandes la implementación no le logra ganar a la implementación en Python con SageMath. Esto no significa que todo el trabajo no haya servido para nada, porque se podría continuar con el desarrollo y lograr mejorar suficientemente la implementación para igualar o llegar a ser mejor que la de Python con SageMath.

Alguien podría preguntar para qué sirve intentar resolver el Problema 1 y calcular bases de Gröbner no conmutativas. En el caso conmutativo hay un montón de aplicaciones prácticas

importantes, por ejemplo en robótica porque el caso conmutativo tiene mucho significado geométrico; en cambio en el caso no conmutativo las aplicaciones son más limitadas y específicas.

Algo para lo que las bases de Gröbner no conmutativas son muy útiles es para trabajar con álgebras presentadas por generadores y relaciones. En este contexto, los generadores son las variables y las relaciones están dadas por los polinomios generadores del ideal. Un problema que las bases de Gröbner ayudan a resolver es el cálculo de la dimensión de este tipo de álgebras. Por ejemplo, algunos investigadores de FAMAF (incluido Cristian Vay, el director de este trabajo) se enfrentan a este problema trabajando con (las que llaman) álgebras de Nichols, que están explicadas en [And17]. Usaremos algunos ejemplos de estas álgebras en el Capítulo 6.

Como ya se dijo, el problema de la palabra se puede reducir al Problema 1, así que los algoritmos eficientes para bases de Gröbner no conmutativas también pueden ser útiles para el problema de la palabra.

También hay algunas aplicaciones de bases de Gröbner no conmutativas y conmutativas en criptografía. En [AK06] se habla de eso.

Esta tesis está organizada de la siguiente forma. En el Capítulo 2 se explican los contenidos matemáticos mínimos necesarios para que la exposición sea autocontenida. En el Capítulo 3 se explica cómo usar la librería que se desarrolló. En el Capítulo 4 se describe cómo están hechas las implementaciones. En el Capítulo 5 se describe brevemente qué tests se hicieron para eliminar los errores y darle robustez a la librería. En el Capítulo 6 se muestran algunos resultados de benchmarks que se hicieron, comparando cosas dentro de la librería y comparando con la implementación de Python con SageMath. Por último en el Capítulo 7 se destacan algunos puntos positivos del trabajo y se mencionan algunos trabajos futuros que se podrían hacer.

Capítulo 2
Preliminares

En este capítulo se desarrolla toda la parte matemática, basada principalmente en la exposición de [Hof20]. Primero se explican sistemas de reescritura, después se introduce el álgebra libre, se prueba que es un anillo, se dan algunas definiciones y propiedades de anillos y se definen las bases de Gröbner. Por último, se explican y dan seudocódigos para los algoritmos de Buchberger y F4, que son los dos algoritmos importantes para calcular bases de Gröbner no conmutativas.

El capítulo incluye demostraciones de muchos de los resultados expuestos, pero leer las demostraciones no es necesario para entender la tesis.

2.1 Sistemas de reescritura

Sistemas de reescritura trata básicamente sobre el estudio de relaciones entre objetos con la idea de usar las relaciones para convertir un objeto en otro. En general usaremos relaciones denotadas con algún símbolo con forma de flecha porque se tiene la idea de que se *usa* la relación para convertir el objeto de la base de la flecha en el objeto de la punta de la flecha. Como recordatorio, una relación entre A y B es un subconjunto de $A \times B$ y una relación sobre A es un subconjunto de A^2.

Para toda esta sección fijemos un conjunto A. Primero recordamos algunas operaciones muy comunes sobre relaciones.

Definición 2.1. *Dadas relaciones $\to$ y $\rightsquigarrow$ sobre A se define:*

- $\to \circ \rightsquigarrow = \{(x,z) \in A^2 : \exists y \in A : x \to y \wedge y \rightsquigarrow z\}$.
- $\to^0 = \{(x,x) : x \in A\}$.
- $\to^{i+1} = \to^i \circ \to$.
- $\to^* = \bigcup_{i=0}^{\infty} \to^i$.
- $\leftarrow = \{(y,x) \in A^2 : (x,y) \in \to\}$.
- $\leftrightarrow = \to \cup \leftarrow$.

Con esta definición se define automáticamente también $\leftrightarrow^*$ que se llama la clausura reflexotransitiva de $\to$.

Con $\leftarrow$ y $\leftrightarrow$ podría haber problema para usarlas con una relación denotada por un símbolo que no tenga forma de flecha, pero en esta tesis las relaciones que se usan siempre se denotan con un símbolo con forma de flecha.

Para el resto de la sección fijemos una relación $\to$ sobre A. Tenemos los siguientes dos lemas.

Lema 2.2. *Sean $a, b \in A$. Entonces*

$$a \to^* b \Leftrightarrow \exists n \in \mathbb{N} \cup \{0\}, x_0, ..., x_n \in A : x_0 = a \wedge x_n = b \wedge \forall\, i \in \{1, ..., n\} : x_{i-1} \to x_i.$$

□

Lema 2.3. *$\leftrightarrow^*$ es la mínima relación de equivalencia que contiene a $\to$.* □

Como querremos hablar de usar $\to$ para transformar un objeto en otro, es útil la próxima definición que permite hablar de una forma normal de un objeto como un elemento al que se llega aplicando $\to$ hasta que no se pueda más.

Definición 2.4. *Dados $a, b \in A$ se define:*

- *a está en forma normal $\Leftrightarrow \nexists x \in A : a \to x$.*
- *b es forma normal de $a \Leftrightarrow a \to^* b \wedge b$ está en forma normal.*
- *a tiene forma normal $\Leftrightarrow \exists x \in A : x$ es forma normal de a.*
- *$a \downarrow b \Leftrightarrow \exists x \in A : a \to^* x \wedge b \to^* x$.*

Con $\downarrow$ pasa lo mismo que con $\leftarrow$ y $\leftrightarrow$ de que podría ser problemático usarla para una relación denotada con un símbolo que no tenga forma de flecha, pero en esta tesis eso no pasa.

También necesitamos definir las siguientes propiedades de las relaciones.

Definición 2.5. *Se define que*

- *$\to$ es confluente $\Leftrightarrow \forall x, y, z \in A : x \to^* y \wedge x \to^* z \Rightarrow y \downarrow z$.*
- *$\to$ es Church-Rosser $\Leftrightarrow \forall x, y \in A : x \leftrightarrow^* y \Leftrightarrow x \downarrow y$.*
- *$\to$ es normalizante $\Leftrightarrow \forall x \in A : x$ tiene forma normal.*
- *$\to$ es terminante $\Leftrightarrow \nexists X \in A^{\mathbb{N}} : \forall i \in \mathbb{N} : X_i \to X_{i+1}$.*

Las siguientes propiedades referidas a relaciones serán de utilidad.

Teorema 2.6. *$\to$ es terminante $\Rightarrow$ $\to$ es normalizante.* □

Teorema 2.7. *$\to$ es confluente $\Leftrightarrow$ $\to$ es Church-Rosser.* □

Si $\to$ es confluente y normalizante entonces todos los elementos tienen una única forma normal, y si además es terminante tenemos la siguiente observación.

Observación 2.8. *Si $\to$ es confluente y terminante entonces calcular la forma normal de a y de b permite decidir si $a \leftrightarrow^* b$.* □

Con esto ya está todo lo necesario sobre sistemas de re-escritura. Para aprender más sobre el tema se puede leer [BN99].

2.2 Álgebra libre

El álgebra libre es básicamente el conjunto de los polinomios no conmutativos con sus operaciones. En los polinomios no conmutativos los monomios son palabras y la única operación interna de los monomios (entre monomios y que devuelve monomios) es la multiplicación, que equivale a la concatenación de palabras.

Definición 2.9. *Sea X un alfabeto finito. Se define la estructura $(\langle X \rangle, \cdot)$ de la siguiente manera:*

- *$\langle X \rangle$ es el conjunto de palabras finitas sobre X.*
- *$\cdot : \langle X \rangle^2 \to \langle X \rangle$ es la concatenación.*

Al par $(\langle X \rangle, \cdot)$ se lo llama el monoide libre sobre X, a los elementos de $\langle X \rangle$ se los llama monomios libres sobre X y a $\cdot$ el producto de $\langle X \rangle$.

Si $X = \{x_1, \ldots, x_n\}$ escribimos $\langle x_1, \ldots, x_n \rangle$ en lugar de $\langle X \rangle$.

Por ejemplo, si $X = \{x, y, z\}$, algunos monomios son:

$$\begin{aligned} m_0 &= xyyzy, \\ m_1 &= yyz, \\ m_2 &= yzy, \\ m_3 &= \varepsilon, \\ m_1 \cdot m_2 &= yzzyzy. \end{aligned}$$

El ε de m_3 es la palabra vacía. Notar que $m_1 \neq m_2$ ya que el producto es no conmutativo.

Para todo el resto de la tesis fijemos un alfabeto finito X.

En estos monomios, al igual que en los conmutativos, se puede hablar de que un monomio divida a otro, pero acá que un monomio divida a otro es equivalente a que sea una sub-palabra.

Definición 2.10. *Sean $m, m' \in \langle X \rangle$. Se define que m divida a m', denotado como $m|m'$ de la siguiente forma:*

$$m|m' \Leftrightarrow \exists a, b \in \langle X \rangle : m' = amb.$$

En el ejemplo de antes tenemos que $m_1|m_0$ ya que $m_0 = am_1b$.

Hablar del resultado de la división es un poco más complicado acá porque tendría que haber dos resultados, el a y el b de la definición, así que en ningún momento hablaremos de dividir un monomio en otro ni escribiremos divisiones entre monomios.

Más adelante será necesario tener un orden entre los elementos de $\langle X \rangle$. Se podría fijar uno concreto definiendo directamente $\leq$ para $\langle X \rangle$, pero es mejor enunciar las mínimas propiedades necesarias y trabajar con cualquier orden que las satisfaga. Eso se logra con la siguiente definición.

Definición 2.11. *Sea $\leq$ un orden total sobre $\langle X \rangle$. Se define que $\leq$ es un buen orden monomial si y solo si:*

(1) $\forall m, m', a, b \in \langle X \rangle : m \leq m' \Rightarrow amb \leq am'b$.

(2) $\forall S \subseteq \langle X \rangle : S \neq \emptyset \Rightarrow S$ *tiene mínimo elemento con respecto a* $\leq$.

Un ejemplo de orden que cumple con esta definición es el siguiente.

Definición 2.12. *Fijemos $X = \{x_1, ..., x_n\}$ y un orden total sobre X: $x_1 \leq ... \leq x_n$, el cual se extiende (como es usual) de forma lexicográfica a $\langle X \rangle$. El orden lexicográfico por grado $\leq_{deglex}$ sobre $\langle X \rangle$ se define así:*

$$m \leq_{deglex} m' \Leftrightarrow |m| < |m'| \vee (|m| = |m'| \wedge m \leq m').$$

O sea, el orden lexicográfico por grado ordena primero por cardinalidad, también llamado grado, y desempata con el orden lexicográfico. Por ejemplo, tenemos que $bc \leq_{deglex} abb$, $aabbc \leq_{deglex} abbcc$ y $\varepsilon \leq_{deglex} a$. Se puede probar fácilmente que este orden es un buen orden monomial.

A partir de ahora fijamos un buen orden monomial $\leq$ y usaremos $<$, $\geq$ y $>$ como se usan habitualmente.

La siguiente propiedad es consecuencia directa de la definición de buen orden monomial.

Teorema 2.13. *La relación $\leq$ no tiene sucesiones estrictamente decrecientes infinitas.* □

Ahora pasemos a hablar de sumar monomios entre sí para tener polinomios no conmutativos.

Definición 2.14. *Sea R un anillo conmutativo. Se define la R-álgebra libre sobre X como el conjunto*

$$R\langle X\rangle = \{\sum_{i=1}^{n} c_i m_i : c_1, \dots, c_n \in R, m_1, \dots, m_n \in \langle X\rangle\}$$

con las siguientes operaciones: la suma de los elementos de $R\langle X\rangle$ como definido unir las $\sum$, el producto por escalares definido como

$$c(\sum_{i=1}^{n} c_i m_i) = \sum_{i=1}^{n} c c_i m_i,$$

y el producto entre elementos de $R\langle X\rangle$ definido como

$$(\sum_{i=1}^{n} c_i m_i) \cdot (\sum_{i=1}^{m} c'_i m'_i) = \sum_{i=1}^{n}\sum_{j=1}^{m} c_i c'_j m_i m'_j.$$

A los elementos de $R\langle X\rangle$ se los llama polinomios no conmutativos. Cuando tengamos una lista de variables $x_1, \dots, x_n$ escribimos $R\langle x_1, \dots, x_n\rangle$ en lugar de $R\langle\{x_1, \dots, x_n\}\rangle$.

Recordemos los ejemplos de polinomios no conmutativos en $\mathbb{Q}\langle x, y, z\rangle$ dados en la introducción:

$$\begin{aligned} f_0 &= x, \\ f_1 &= xy + zy, \\ f_2 &= 3xyy - 2xzxy + \frac{4}{3}yzzx. \end{aligned}$$

Recordar que $f_1 \neq xy + xz$ ya que el producto es no conmutativo.

Sobre los polinomios no conmutativos se hacen las siguientes definiciones.

Definición 2.15. *Sean R un anillo conmutativo, $f \in R\langle X\rangle$, $c_1, \dots, c_n \in R - \{0\}$, $m_1, \dots, m_n, m \in \langle X\rangle$, $f = \sum_{i=1}^{n} c_i m_i$ y $\leq$ un buen orden monomial. Se definen:*

- *el coeficiente de m en f es* $f_m = \begin{cases} c_0 & \text{si } m = m_0 \\ \vdots & \\ c_n & \text{si } m = m_n \\ 0 & \text{en otro caso} \end{cases}$.
- *el soporte de f es el conjunto* $\operatorname{sop}(f) = \{m_1, \dots, m_n\}$.
- *el monomio principal de f es* $\operatorname{lm}_{\leq}(f) = \max_{\leq}(\operatorname{sop}(f))$.
- *el coeficiente principal de f es* $\operatorname{lc}_{\leq}(f) = f_{\operatorname{lm}_{\leq}(f)}$.
- *el término principal de f es* $\operatorname{lt}_{\leq}(f) = \operatorname{lc}_{\leq}(f) \cdot \operatorname{lm}_{\leq}(f)$.
- $\operatorname{tail}_{\leq}(f) = f - \operatorname{lt}_{\leq}(f)$.

Los nombres lm, lc, lt y tail vienen del inglés leading monomial, leading coefficient, leading term y tail respectivamente.

Siguiendo con el ejemplo tenemos:

$$\begin{aligned} f_{2_{xyy}} &= 3, \\ \operatorname{sop}(f_2) &= \{xyy, xxxy, yxxx\}, \\ \operatorname{lm}_{\leq_{deglex}}(f_2) &= yxxx, \\ \operatorname{lc}_{\leq_{deglex}}(f_2) &= 4, \\ \operatorname{lt}_{\leq_{deglex}}(f_2) &= 4yxxx, \\ \operatorname{tail}_{\leq_{deglex}}(f_2) &= 3xyy - 2xxxy. \end{aligned}$$

Sobre estas definiciones valen muchas propiedades fáciles de probar, como por ejemplo $\mathrm{lm}_{\leq}(f)\,\mathrm{lm}_{\leq}(f') = \mathrm{lm}_{\leq}(ff')$, que durante el resto de la tesis usaremos mucho, pero no las probamos una por una porque sería tedioso y aburrido.

Más adelante será necesario comparar no solo monomios sino también polinomios, así que el orden $\leq$ se extiende a $K\langle X\rangle$ así:

Definición 2.16. *Sean R un anillo conmutativo y $f, g \in R\langle X\rangle$. Se define que $f < g$ si y solo si vale alguna de las siguientes:*

(1) $f = 0 \wedge g \neq 0$.

(2) $\mathrm{lm}_{\leq}(f) < \mathrm{lm}_{\leq}(g)$.

(3) $\mathrm{lm}_{\leq}(f) = \mathrm{lm}_{\leq}(g) \wedge \mathrm{tail}_{\leq}(f) < \mathrm{tail}_{\leq}(g)$.

Y como es usual $f \leq g$ si y solo si $f < g \vee f = g$.

Dicho en palabras, el orden en los polinomios es orden lexicográfico con el polinomio visto como una lista de monomios, sin coeficientes, ordenada de mayor a menor. Por ejemplo, tenemos estas desigualdades en $K\langle X\rangle$:

$$x < xy,$$
$$yzz + z < yzz + xyy,$$
$$xz < xz + xx.$$

Si bien a este $<$ lo estamos llamando (y lo continuaremos llamando) orden, en realidad no es un orden sino un preorden porque cuando solo cambian los coeficientes entre un polinomio y otro, ninguno de los polinomios es menor que el otro.

Teorema 2.17. *Sea R un anillo conmutativo. Entonces*

la relación $<$ en $R\langle X\rangle$ es un preorden parcial.

□

Además, el polinomio 0 es el mínimo.

Lema 2.18. *0 es el mínimo de $<$.* □

En este orden, al igual que para los monomios, vale que no hay sucesiones estrictamente decrecientes infinitas.

Lema 2.19. *Sea R un anillo conmutativo. Entonces*

la relación $\leq$ en $R\langle X\rangle$ no tiene sucesiones estrictamente decrecientes infinitas.

Demostración: Supongamos que existen sucesiones estrictamente decrecientes infinitas. Tomemos una sucesión estrictamente decreciente infinita P, o sea que valga $P_1 > P_2 > P_3 > \ldots$, que minimice $\mathrm{lm}_{\leq}(P_1)$. Tomar este mínimo es posible por el Teorema 2.13 que dice que no hay sucesiones estrictamente decrecientes infinitas en los monomios.

Notemos que:

Afirmación 2.19.1. $\forall\, i \in \mathbb{N} : \mathrm{lm}_{\leq}(P_i) = \mathrm{lm}_{\leq}(P_1)$.

En efecto, no puede ser $\mathrm{lm}_{\leq}(P_i) < \mathrm{lm}_{\leq}(P_1)$ porque entonces $P_i, P_{i+1}, \ldots$ sería una sucesión estrictamente decreciente infinita que rompería la minimalidad de $\mathrm{lm}_{\leq}(P_1)$ y no puede ser $\mathrm{lm}_{\leq}(P_i) > \mathrm{lm}_{\leq}(P_1)$ porque eso implicaría $P_i > P_1$ y eso contradice que P sea decreciente.

Afirmación 2.19.2. $\mathrm{tail}_{\leq}(P_1), \mathrm{tail}_{\leq}(P_2), \ldots$ es una sucesión estrictamente decreciente infinita.

Esto vale porque al aplicar la definición del orden polinomial sobre $P_1 > P_2 > P_3 > ...$ y usar la Afirmación 2.19.1 queda $\text{tail}_{\leq}(P_1) > \text{tail}_{\leq}(P_2) > \text{tail}_{\leq}(P_3) > ...$

Como por el Lema 2.18 0 es un mínimo:

Afirmación 2.19.3. $P_1 \neq 0$.

Sin embargo, la Afirmación 2.19.2 contradice la minimalidad de $\text{lm}_{\leq}(P_1)$ ya que por la Afirmación 2.19.3 vale que $\text{lm}_{\leq}(\text{tail}_{\leq}(P_1)) < \text{lm}_{\leq}(P_1)$.

□

La estructura del álgebra libre es un anillo, lo cual es muy útil por muchas definiciones y teoremas que ya existen sobre los anillos.

Teorema 2.20. *Sea R un anillo conmutativo. Entonces*

$$(R\langle X\rangle, +, \cdot) \text{ es un anillo.}$$

□

Las definiciones y teoremas sobre anillos necesarios están a continuación.

Definición 2.21. *Sean R un anillo e $I \subseteq R$. Se define que I es un ideal de R si y solo si:*

(1) $I \neq \emptyset$.

(2) $\forall\, a, b \in I : a + b \in I$.

(3) $\forall\, a \in I, r, r' \in R : rar' \in I$.

Los ideales son básicamente conjuntos cerrados por la suma y por el producto por cualquier elemento del anillo. Si se tiene un conjunto que no es un ideal se puede agregar todo lo mínimo necesario para convertirlo en un ideal. La siguiente definición define eso y el siguiente teorema dice que efectivamente se obtiene un ideal.

Definición 2.22. *Sean R un anillo y $G \subseteq R$. Se define el ideal generado por G, denotado (G), como*

$$(G) = \{\sum_{i=1}^{n} c_i g_i c_i' : n \in \mathbb{N} \cup \{0\}, g_1, ..., g_n \in G, c_1, ..., c_n, c_1', ..., c_n' \in R\}.$$

A los c_i, c_i' se los llama cofactores.

Teorema 2.23. *Sean R un anillo y $G \subseteq R$. Entonces*

$$(G) \text{ es un ideal de } R.$$

□

Sobre los ideales generados por un conjunto valen los siguientes dos lemas básicos.

Lema 2.24. *Sean R un anillo, $G \subseteq R$ y $a \in (G)$. Entonces*

$$(G) = (G \cup \{a\}).$$

□

Lema 2.25. *Sean R un anillo y $G, G' \subseteq R$. Entonces*

$$G \subseteq (G') \wedge G' \subseteq (G) \Leftrightarrow (G) = (G').$$

□

Cada ideal define una clase de equivalencia en el anillo, la cual se llama congruencia módulo el ideal.

Definición 2.26. *Sean R un anillo e $I \subseteq R$. Se define la relación $\equiv_I$ en R, llamada congruencia módulo I, así:*

$$a \equiv_I b \Leftrightarrow a - b \in I.$$

Teorema 2.27. *Sean R un anillo e $I \subseteq R$ un ideal. Entonces*

$$\equiv_I \text{ es una relación de equivalencia.}$$

□

Esta congruencia es parecida a la de los enteros módulo un natural (de hecho, la congruencia de los enteros módulo un natural es un subcaso de ésta tomando como ideal a todos los múltiplos del módulo). Vale que la clase de equivalencia del 0 es el propio ideal:

Lema 2.28. *Sean R un anillo, $I \subseteq R$ un ideal y $a \in R$. Entonces*

$$a \in I \Leftrightarrow a \equiv_I 0.$$

□

Ahora volvemos al álgebra libre y a partir de ahora fijamos un cuerpo K.

Observación 2.29. *Con las definiciones que tenemos ahora el Problema 1 se puede escribir como: dado un conjunto finito $G \subseteq K\langle X\rangle$ y un elemento $f \in K\langle X\rangle$, determinar si $f \in (G)$.*

Esta observación es tan importante que de hecho el Problema 1 se llama 'problema de pertenencia al ideal'.

Si bien (G) para un anillo general está definido usando combinaciones lineales con coeficientes en el anillo, para el álgebra libre es útil la siguiente equivalencia que dice que alcanza con considerar solo monomios y elementos del cuerpo como cofactores.

Lema 2.30. *Sea $G \subseteq K\langle X\rangle$. Entonces*

$$(G) = \{\sum_{i=0}^{n} c_i m_i g_i m_i' : n \in \mathbb{N} \cup \{0\}, c_1, \dots, c_n \in K, m_1, \dots, m_n, m_1', \dots, m_n' \in \langle X\rangle, g_1, \dots, g_n \in G\}.$$

Demostración: Probemos que f está en uno si y solo si está en el otro.

Ida ($\Rightarrow$): Supongamos $f \in (G)$. Sean:

- $g_1, \dots, g_n \in G, f_1, \dots, f_n, f_1', \dots, f_n' \in K\langle X\rangle$ tales que $f = \sum_{i=1}^{n} f_i g_i f_i'$, los cuales existen porque estamos suponiendo $f \in (G)$ y por la definición de (G).
- Para cada $i \in \{1, \dots, n\}$:
  - $c_{i,1}, \dots, c_{i,n_i} \in K, m_{i,1}, \dots, m_{i,n_i} \in \langle X\rangle$ tales que $f_i = \sum_{j=1}^{n_i} c_{i,j} m_{i,j}$.
  - $c_{i,1}', \dots, c_{i,n_i'}' \in K, m_{i,1}', \dots, m_{i,n_i'}' \in \langle X\rangle$ tales que $f_i' = \sum_{j=1}^{n_i'} c_{i,j}' m_{i,j}'$.

Con esto tenemos:

$$\begin{aligned} f &= \sum_{i=1}^{n} f_i g_i f_i' \\ &= \sum_{i=1}^{n} (\sum_{j=1}^{n_i} c_{i,j} m_{i,j}) g_i (\sum_{j=1}^{n_i'} c_{i,j}' m_{i,j}') \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n_i} \sum_{j'=1}^{n_i'} c_{i,j} c_{i,j'}' m_{i,j} g_i m_{i,j'}'. \end{aligned}$$

Esto último es una expresión que se transforma a la forma que queremos.

Vuelta ($\Leftarrow$): La vuelta es cierta porque, como $c_i m_i, m_i' \in K\langle X\rangle$, es un caso particular de la definición.

□

2.3 Reducción de polinomios

Ahora definiremos una relación de reducción en los polinomios no conmutativos la cual depende del orden monomial y de un conjunto generador G, cuya clausura reflexo-transitiva es igual a $\equiv_{(G)}$. La relación que definiremos siempre es terminante, pero no siempre confluente. Para los casos en los que sí sea confluente la Observación 2.8 nos permitirá chequear si dos polinomios son equivalentes y en particular, por el Lema 2.28, si un polinomio está en (G). A los casos en los que no sea confluente después trataremos de convertirlos en confluentes.

Definición 2.31. *Sean $G \subseteq K\langle X\rangle$ y $f, f' \in K\langle X\rangle$. Se define la relación $\to_{\leq,G}$ del siguiente modo:*

$$f \to_{\leq,G} f' \Leftrightarrow \exists a, b \in \langle X\rangle, g \in G : \mathrm{lm}_{\leq}(agb) \in \mathrm{sop}(f) \wedge f' = f - \frac{f_{\mathrm{lm}_{\leq}(agb)}}{\mathrm{lc}_{\leq}(g)} agb.$$

Cuando vale $f \to_{\leq,G} f'$ se dice que f se reduce a f' (vía G).

Por ejemplo, si tenemos $f = 2zyx + 3xyzxx$, $g_0 = x - yzx$ y $G = \{g_0\}$ tenemos:

$$f \to_{\leq,G} f + 3xg_0x = 3xxx + 2zyx.$$

Como teníamos la Definición 2.1 y la Definición 2.4 quedan definidas automáticamente las relaciones $\to^*_{\leq,G}$, $\leftrightarrow^*_{\leq,G}$ y $\downarrow_{\leq,G}$.

A continuación la prueba de que esta relación efectivamente achica.

Teorema 2.32. *Sean $G \subseteq K\langle X\rangle$ y $f, f' \in K\langle X\rangle$. Entonces*

$$f \to_{\leq,G} f' \Rightarrow f' < f.$$

Demostración: Supongamos el antecedente. Por definición de reducciones tomemos $a, b \in \langle X\rangle, g \in G$ tales que:

(i) $\mathrm{lm}_{\leq}(agb) \in \mathrm{sop}(f)$.

(ii) $f' = f - \frac{f_{\mathrm{lm}_{\leq}(agb)}}{\mathrm{lc}_{\leq}(g)} agb$.

Sean:

- $c_1, \ldots, c_n \in K, m_1, \ldots, m_n \in \langle X\rangle$ con $m_1 > m_2 > \cdots > m_n$ tales que $f = \sum_{i=1}^{n} c_i m_i$.
- i tal que $m_i = \mathrm{lm}_{\leq}(agb)$, el cual existe por (i).

Notar también que:

(iii) $m_i = \mathrm{lm}_{\leq}(\frac{f_{\mathrm{lm}_{\leq}(agb)}}{\mathrm{lc}_{\leq}(g)} agb)$.

(ii) y (iii) implican que los términos $c_1m_1, c_2m_2, \ldots, c_{i-1}, m_{i-1}$ son iguales en f y en f' y no hay nada más en el medio, porque f' es f con términos menores o iguales a $\mathrm{lm}_{\leq}(\frac{f_{\mathrm{lm}_{\leq}(agb)}}{\mathrm{lc}_{\leq}(g)} agb)$ restadas (por (ii)).

Además, por (iii) y (ii) vale que $f'_{m_i} = 0$, por ende, el término que sigue después de m_{i-1} (si es que hay) es menor que m_i.

Combinando que los primeros $i-1$ términos son iguales y el i es menor en f' que en f, por la definición de $<$ para polinomios vale que $f' < f$.

□

La relación $\to_{\leq,G}$ tiene varias propiedades útiles que se prueban a continuación.

Lema 2.33. *Sean $G \subseteq K\langle X\rangle$ y $f_0, f_1, f \in K\langle X\rangle$. Entonces*

$$f_0 \to_{\leq,G} f_1 \Rightarrow f_0 + f \downarrow_{\leq,G} f_1 + f.$$

Demostración: Supongamos el antecedente $f_0 \to_{\leq,G} f_1$.

Sean $g \in G, a, b \in K\langle X\rangle$ tales que $f_1 = f_0 - \frac{f_{0_{\mathrm{lm}_\leq(agb)}}}{g_{\mathrm{lm}_\leq(agb)}} agb$, los cuales existen por definición de $\to_{\leq,G}$.

Dividamos la demostración en casos según la pertenencia de $\mathrm{lm}_\leq(agb)$ a $\mathrm{sop}(f)$:

Caso $\mathrm{lm}_\leq(agb) \notin \mathrm{sop}(f)$**:** Partiendo de la condición de a, g, b hacemos lo siguiente:

$$\begin{aligned}
&f_1 = f_0 - \frac{f_{0_{\mathrm{lm}_\leq(agb)}}}{g_{\mathrm{lm}_\leq(agb)}} \\
&\Rightarrow f_1 + f = f_0 - \frac{f_{0_{\mathrm{lm}_\leq(agb)}}}{g_{\mathrm{lm}_\leq(agb)}} + f && \text{Definición de } \to_{\leq,G} \\
&\Rightarrow f_0 + f \to_{\leq,G} f_1 + f && \text{Definición de } \downarrow, \text{ ambos se reducen a } f_1 + f \\
&\Rightarrow f_0 + f \downarrow_{\leq,G} f_1 + f.
\end{aligned}$$

Caso $\mathrm{lm}_\leq(agb) \in \mathrm{sop}(f)$**:**

Subcaso $f_{\mathrm{lm}_\leq(agb)} = -f_{0_{\mathrm{lm}_\leq(agb)}}$ En este caso $\mathrm{lm}_\leq(agb)$ se cancela en la suma $f_0 + f$. Y como además $\mathrm{lm}_\leq(agb) \notin \mathrm{sop}(f_1)$ por el antecedente, tenemos:

$$\begin{aligned}
&f_1 + f \to_{\leq,G} f_1 + f - \frac{f_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb && \text{Subcaso} \\
&\Rightarrow f_1 + f \to_{\leq,G} f_1 + f + \frac{f_{0_{\mathrm{lm}_\leq(agb)}}}{g_{\mathrm{lm}_\leq(agb)}} agb && \text{Condición de } g, a \text{ y } b \\
&\Rightarrow f_1 + f \to_{\leq,G} f_0 + f \\
&\Rightarrow f_1 + f \downarrow_{\leq,G} f_0 + f.
\end{aligned}$$

Subcaso $f_{\mathrm{lm}_\leq(agb)} \neq -f_{0_{\mathrm{lm}_\leq(agb)}}$**:** En este caso $\mathrm{lm}_\leq(agb)$ no se cancela en la suma $f_0 + f$, así que podemos aplicar $\to_{\leq,G}$ con a, g, b:

$$\begin{aligned}
&f_0 + f \to_{\leq,G} f_0 + f - \frac{(f_0 + f)_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb \\
&\Rightarrow f_0 + f \to_{\leq,G} f_0 + f - \frac{f_{0_{\mathrm{lm}_\leq(agb)}} + f_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb \\
&\Rightarrow f_0 + f \to_{\leq,G} f_0 + f - \frac{f_{0_{\mathrm{lm}_\leq(agb)}}}{g_{\mathrm{lm}_\leq(agb)}} agb - \frac{f_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb && \text{Condición de } g, a \text{ y } b \\
&\Rightarrow f_0 + f \to_{\leq,G} f_1 + f - \frac{f_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb.
\end{aligned}$$

Llamemos (i) a este último resultado.

Además por el caso y el hecho de que $\mathrm{lm}_\leq(agb) \notin \mathrm{sop}(f_1)$ también tenemos $\mathrm{lm}_\leq(agb) \in$

$\mathrm{sop}(f_1 + f)$, entonces:

$$
\begin{aligned}
& f_1 + f \to_{\leq,G} f_1 + f - \frac{(f_1 + f)_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb \\
& \Rightarrow f_1 + f \to_{\leq,G} f_1 + f - \frac{f_{1_{\mathrm{lm}_\leq(agb)}} + f_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb \quad \left.\begin{array}{l} \mathrm{lm}_\leq(agb) \notin \\ \mathrm{sop}(f_1) \text{ porque} \\ f_0 \to_{\leq,G} f_1 \text{ y} \\ \text{definición de} \\ \to_{\leq,G} \end{array}\right. \\
& \Rightarrow f_1 + f \to_{\leq,G} f_1 + f - \frac{0 + f_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb \\
& \Rightarrow f_1 + f \to_{\leq,G} f_1 + f - \frac{f_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb.
\end{aligned}
$$

Llamemos (ii) a este último resultado.
Por (i) e (ii) tenemos $f_0 + f \downarrow_{\leq,G} f'_0 + f$.

□

Lema 2.34. *Sean $G \subseteq K\langle X\rangle$, $f, f' \in K\langle X\rangle$, $c \in K$ y $m, m' \in \langle X\rangle$. Entonces*

$$f \to_{\leq,G} f' \Rightarrow cmfm' \to_{\leq,G} cmf'm'.$$

Demostración: Supongamos el antecedente.

Sean $g \in G$ y $a, b \in \langle X\rangle$ tales que $f' = f - \frac{f_{\mathrm{lm}_\leq(agb)}}{\mathrm{lc}_\leq(g)} agb$, los cuales existen por definición de $\to_{\leq,G}$. Tenemos:

$$
\begin{aligned}
& f' = f - \frac{f_{\mathrm{lm}_\leq(agb)}}{\mathrm{lc}_\leq(g)} agb \\
& \Rightarrow cmf'm' = cm(f - \frac{f_{\mathrm{lm}_\leq(agb)}}{\mathrm{lc}_\leq(g)} agb)m' \\
& \Rightarrow cmf'm' = cmfm' - \frac{(cmfm')_{\mathrm{lm}_\leq(cmagbm')}}{\mathrm{lc}_\leq(g)} cmagbm' \quad \left.\begin{array}{l} \\ \\ \text{Definición } \to_{\leq,G} \end{array}\right. \\
& \Rightarrow cmf'm' \to_{\leq,G} cmfm'.
\end{aligned}
$$

□

Ahora el ya anunciado teorema de que la clausura reflexo transitiva de $\to_{\leq,G}$ es una congruencia.

Teorema 2.35. *Sea $G \subseteq K\langle X\rangle$. Entonces*

$$\leftrightarrow^*_{\leq,G} = \equiv_{(G)}.$$

Demostración: Probemos las dos inclusiones.

Prueba de $\leftrightarrow^*_{\leq,G} \subseteq \equiv_{(G)}$: Como $\leftrightarrow^*_{\leq,G}$ y $\equiv_{(G)}$ son relaciones de equivalencia y además por el Lema 2.3 $\leftrightarrow^*_{\leq,G}$ es la mínima relación de equivalencia que contiene a $\to_{\leq,G}$, alcanza con probar $\to_{\leq,G} \subseteq \equiv_{(G)}$. Para eso supongamos $f \to_{\leq,G} f'$ y probemos $f \equiv_{(G)} f'$.

Sean $g \in G, a, b \in K\langle X\rangle$ tales que $\mathrm{lm}_\leq(agb) \in f$ y $f' = f - \frac{f_{\mathrm{lm}_\leq(agb)}}{g_{\mathrm{lm}_\leq(agb)}} agb$, los cuales existen por definición de $\to_{\leq,G}$. Tenemos:

$$
\begin{aligned}
& f \equiv_{(G)} f' \\
& \Leftrightarrow f - f' \in (G) \\
& \Leftrightarrow f - (f - \frac{f_{\mathrm{lm}_\leq(agb)}}{\mathrm{lc}_\leq(g)} agb) \in (G) \\
& \Leftrightarrow \frac{f_{\mathrm{lm}_\leq(agb)}}{\mathrm{lc}_\leq(g)} agb \in (G).
\end{aligned}
$$

Y esto último es claramente cierto por la definición de (G) (Definición 2.22).

Prueba de $\equiv_{(G)} \subseteq \leftrightarrow^*_{\leq,G}$: Supongamos $f \equiv_{(G)} f'$ y probemos $f \leftrightarrow^*_{\leq,G} f'$. Sean:

- $g = f - f'$.
- $c_1, ..., c_n \in K, m_1, ..., m_n, m'_1, ..., m'_n \in \langle X \rangle$, $g_1, ..., g_n \in G$ tales que $g = \sum_{i=1}^{n} c_i m_i g_i m'_i$, los cuales existen porque por definición de $\equiv_G$ tenemos $g \in (G)$ y por el Lema 2.30.
- $f_0 = f$.
- Para cada $i \in \{1, ..., n\}$: $f_i = f_{i-1} - c_i m_i g_i m'_i$.

Tenemos entonces:

$$\begin{aligned}
&\forall i \in \{1, ..., n\} : g_i \to_{\leq,G} 0 \\
&\Rightarrow \forall i \in \{1, ..., n\} : c_i m_i g_i m'_i \to_{\leq,G} 0 && \text{Lema 2.34} \\
&\Rightarrow \forall i \in \{1, ..., n\} : c_i m_i g_i m'_i + f_i \downarrow_{\leq,G} 0 + f_i && \text{Lema 2.33} \\
&\Rightarrow \forall i \in \{1, ..., n\} : f_{i-1} \downarrow_{\leq,G} f_i && \text{Definición de los } f_i \\
&\Rightarrow \forall i \in \{1, ..., n\} : f_{i-1} \leftrightarrow^*_{\leq,G} f_i \\
&\Rightarrow f_0 \leftrightarrow^*_{\leq,G} f_n && f_n = f - g = f' \\
&\Rightarrow f \leftrightarrow^*_{\leq,G} f_n.
\end{aligned}$$

□

Lema 2.36. *Sean $G \subseteq K\langle X \rangle, f, f' \in K\langle X \rangle$. Entonces*

$$f \to^*_{\leq,G} f' \Rightarrow (f \in (G) \Leftrightarrow f' \in (G)).$$

Demostración: Si asumimos $f \to^*_{\leq,G} f'$, tenemos por el Teorema 2.35 que $f \equiv_{(G)} f'$ y entonces por el Lema 2.28 vale que $f \in (G) \Leftrightarrow f' \in (G)$. □

Teorema 2.37. *Sea $G \subseteq K\langle X \rangle$. Entonces*

$$\to_{\leq,G} \text{ es terminante.}$$

Demostración: Por contradicción. Supongamos que $\to_{\leq,G}$ no es terminante. Por definición podemos tomar una sucesión $P \in K\langle X \rangle^{\mathbb{N}}$ tal que:

$$\forall i \in \mathbb{N} : P_i \to_{\leq,G} P_{i+1}.$$

Por el Teorema 2.32 tenemos que:

$$\forall i \in \mathbb{N} : P_i > P_{i+1}.$$

Pero esto contradice el Lema 2.19 que dice que no hay sucesiones estrictamente decrecientes infinitas en $K\langle X \rangle$. □

El siguiente teorema caracteriza las formas normales de $\to$.

Teorema 2.38. *Sean $G \subseteq K\langle X \rangle, f \in K\langle X \rangle$. Entonces*

$$f \text{ está en forma normal con respecto a } \to_{\leq,G} \Leftrightarrow \nexists g \in G, m \in \text{sop}(f) : \text{lm}_{\leq}(g) | m.$$

Demostración: Probemos ida y vuelta por separado.

Ida ($\Rightarrow$): Por contradicción, supongamos el antecedente y que tenemos $g \in G, m \in \text{sop}(f)$ tales que $\text{lm}_{\leq}(g) | m$.

Sean $a, b \in \langle X \rangle$ tales que $m = agb$, los cuales existen por la definición de divisibilidad.

Entonces tenemos $f \to_{\leq,G} f - \frac{f_{\text{lm}_{\leq}(agb)}}{\text{lc}_{\leq}(g)} agb$ y por ende f no está en forma normal.

Vuelta ($\Leftarrow$): Por contrarrecíproca, supongamos que f no está en forma normal con respecto a $\to_{\leq,G}$ y probemos existe g que satisface el $\exists$. Sean:

- $f' \in K\langle X\rangle$ tal que $f \to_{\leq,G} f'$, el cual existe por el antecedente.
- $a, b \in \langle X\rangle, g \in G$ tales que $\mathrm{lm}_{\leq}(agb) \in \mathrm{sop}(f)$ y $f' = f - \frac{f_{\mathrm{lm}_{\leq}(agb)}}{\mathrm{lc}_{\leq}(g)} agb$, los cuales existen por la definición de $\to_{\leq,G}$.

Con esto es claro que g satisface el $\exists$.

□

Probamos que $\to_{\leq,G}$ es terminante, pero no que sea confluente, ya que (como habíamos anticipado) no siempre lo es. Por ejemplo, siguiendo con el mismo ejemplo de antes en el que ya teníamos $f = 2zyx + 3xyzxx$, $g_0 = x - yzx$, si ahora agregamos $g_1 = y - zxx$ y $G = \{g_0, g_1\}$ tenemos, como antes:

$$f \to_{\leq,G} f + 3xg_0x = 3xxx + 2zyx = f'.$$

Y también:

$$f \to_{\leq,G} f - 3xyg_1 = 3xyy + 2zyx = f''.$$

Pero ningún monomio principal de G divide a ningún monomio de f' o f'' así que f' y f'' no se pueden reducir más.

La siguiente definición nos permite hablar más cómodamente de una forma normal de un elemento (tanto en las definiciones y teoremas como en los algoritmos).

Definición 2.39. *Sea $e_{\leq} : \mathscr{P}(K\langle X\rangle) \to K\langle X\rangle \to K\langle X\rangle$. Se define que*

$e_{\leq}$ *es un reductor*
$\Leftrightarrow \forall G \subseteq K\langle X\rangle, f \in K\langle X\rangle : e_{\leq}(G)(f)$ *es forma normal de f con respecto a $\to_{\leq,G}$.*

Un ejemplo de reductor podría calcularse con el siguiente seudocódigo.

Algoritmo 1: Ejemplo de reductor

Data: $G = \{g_1, ..., g_n\} \subseteq K\langle X\rangle, f \in K\langle X\rangle$
Result: $f' \in K\langle X\rangle$

```
1  f' ← f
2  i ← 1
3  while i ≤ n do
4      r ← False
5      for m ∈ sop(f') do
6          if lm_≤(g_i)|m then
7              calcular a, b ∈ ⟨X⟩ tales que m = a lm_≤(g_i) b  f' ← f' − (f'_{lm_≤(g_i)} / lc_≤(g_i)) a g_i b
8              r ← True
9              break
10     if r then
11         i ← 1
12     else
13         i ← i + 1
14 return f'
```

Este algoritmo consiste básicamente en siempre buscar entre los elementos de G si hay alguno con el cual reducir, y parar cuando ya no hay ninguno.

Una propiedad sobre los reductores que necesitaremos es que mantienen la pertenencia a ideales (lo cual tiene mucho sentido por la definición).

Lema 2.40. *Sean $e_{\leq}$ un reductor, $G \subseteq K\langle X\rangle$ y $f \in (G)$. Entonces*

$$e_{\leq}(G)(f) \in (G).$$

Demostración: Es consecuencia directa de la definición y del Lema 2.36. □

2.4 Bases de Gröbner

Que $\to_{\leq,G}$ fuera siempre confluente sería muy útil porque significaría que para cualquier clase de equivalencia de $\equiv_{(G)}$ se podría siempre llegar a una misma forma normal y así determinar si dos elementos son equivalentes. En particular se podría determinar si un elemento está en el ideal viendo si se llega a 0 como forma normal. Los casos en los que sí es confluente se llaman bases de Gröbner y después algo que haremos es calcularle una base de Gröbner de un ideal generado por un conjunto que no es base de Gröbner.

Definición 2.41. *Sean I un ideal de $K\langle X\rangle$ y $G \subseteq K\langle X\rangle$. Se define que*

$$G \text{ es una base de Gröbner de } I \Leftrightarrow (G) = I \wedge \to_{\leq,G} \text{ es confluente.}$$

Además se dice que "G es una base de Gröbner" si lo es de algún ideal.

Una consecuencia directa de la definición es el siguiente lema.

Lema 2.42. *Sea G una base de Gröbner. Entonces*

$$\to_{\leq,G} \text{ es Church-Rosser.}$$

Demostración: Es una aplicación directa del Teorema 2.7. □

Las bases de Gröbner se definieron por la propiedad más importante que queremos que tengan, pero las siguientes equivalencias las harán mucho más cómodas de trabajar.

Teorema 2.43. *Sean I un ideal de $K\langle X\rangle$ y $G \subseteq K\langle X\rangle$. Las siguientes afirmaciones son equivalentes:*

(1) *G es una base de Gröbner de I.*

(2) $\forall f \in K\langle X\rangle : f \in I \Leftrightarrow f \to^*_{\leq,G} 0$.

(3) $(G) = I \wedge \forall f \in K\langle X\rangle : f \in I \Rightarrow f \to^*_{\leq,G} 0$.

(4) $(G) = I \wedge \forall f \in I - \{0\} : \exists g \in G : \mathrm{lm}_{\leq}(g) | \mathrm{lm}_{\leq}(f)$.

(5) $\forall f \in I - \{0\} : \exists c_1, \ldots, c_n \in K, g_1, \ldots, g_n \in G, a_1, \ldots, a_n, b_1, \ldots, b_n \in \langle X\rangle : \mathrm{lm}_{\leq}(a_i g_i b_i) \leq \mathrm{lm}_{\leq}(f) \wedge f = \sum_{i=1}^{n} c_i a_i g_i b_i$.

Demostración: Probaremos **(1)** ⇒ **(2)**, **(2)** ⇒ **(3)**, **(3)** ⇒ **(1)**, **(4)** ⇒ **(3)**, **(2)** ⇒ **(5)** y **(5)** ⇒ **(4)**.

(1) ⇒ (2): Supongamos que G es una base de Gröbner de I y tomemos $f \in K\langle X\rangle$. Tenemos que probar $f \in I \Leftrightarrow f \to^*_{\leq,G} 0$. Vamos de un lado para el otro:

$$\begin{aligned}
& f \in I \\
\Leftrightarrow\ & f \equiv_I 0 && \text{Lema 2.28} \\
\Leftrightarrow\ & f \leftrightarrow^*_{\leq,G} 0 && \text{Teorema 2.35} \\
\Leftrightarrow\ & f \downarrow_{\leq,G} 0 && \text{Por Lema 2.42, } \to_{\leq,G} \text{ es Church-Rosser, definición de Church-Rosser} \\
\Leftrightarrow\ & \exists f' \in K\langle X\rangle : f \to^*_{\leq,G} f' \wedge 0 \to^*_{\leq,G} f' && \text{Definición de } \downarrow \\
\Leftrightarrow\ & \exists f' \in K\langle X\rangle : f \to^*_{\leq,G} f' \wedge f' = 0 && \text{Teorema 2.32 y Lema 2.18} \\
\Leftrightarrow\ & f \to^*_{\leq,G} 0.
\end{aligned}$$

(2) ⇒ (3): Supongamos el antecedente $\forall f \in K\langle X\rangle : f \in I \Leftrightarrow f \to^*_{\leq,G} 0$. Tenemos que probar que $(G) = I \wedge \forall f \in K\langle X\rangle : f \in I \Rightarrow f \to^*_{\leq,G} 0$. Probemos cada término del $\wedge$ por separado:

Prueba de $(G) = I$: Es cierto porque **(2) ⇒ (1)** y $(G) = I$ es parte de la definición de base de Gröbner.

Prueba de $\forall f \in K\langle X\rangle : f \in I \Rightarrow f \to^*_{\leq,G} 0$: Es cierto por **(2)**.

(3) ⇒ (1): Supongamos el antecedente $(G) = I \wedge \forall f \in K\langle X\rangle : f \in I \Rightarrow f \to^*_{\leq,G} 0$. Tenemos que probar que G es una base de Gröbner de I, es decir $(G) = I \wedge \to_{\leq,G}$ es confluente. La parte de $(G) = I$ es válida porque es parte de **(3)**.

Para la otra parte, por definición de confluente, alcanza con probar $\forall f, f_0, f_1 \in K\langle X\rangle : f \to^*_{\leq,G} f_0 \wedge f \to^*_{\leq,G} f_1 \Rightarrow f_0 \downarrow_{\leq,G} f_1$. En tal caso, vale por lo siguiente:

$$
\begin{aligned}
& f \to^*_{\leq,G} f_0 \wedge f \to^*_{\leq,G} f_1 \\
& \Rightarrow f_0 \leftrightarrow^*_{\leq,G} f_1 && \text{Teorema 2.35} \\
& \Rightarrow f_0 \equiv_{(G)} f_1 && \text{Definición de } \equiv_G \\
& \Rightarrow f_0 - f_1 \in (G) && \text{Antecedente} \\
& \Rightarrow f_0 - f_1 \to^*_{\leq,G} 0 && \text{Lema 2.33} \\
& \Rightarrow (f_0 - f_1) + f_1 \downarrow_{\leq,G} 0 + f_1 \\
& \Rightarrow f_0 \downarrow_{\leq,G} f_1.
\end{aligned}
$$

(4) ⇒ (3): Supongamos el antecedente $(G) = I \wedge \forall f \in I - \{0\} : \exists g \in G : \mathrm{lm}_\leq(g) | \mathrm{lm}_\leq(f)$. Tenemos que probar $(G) = I \wedge \forall f \in K\langle X\rangle : f \in I \Rightarrow f \to^*_{\leq,G} 0$. La parte de $(G) = I$ es válida porque es parte de **(4)**. Para la otra parte probémoslo por contradicción. En particular tomemos el mínimo f tal que $f \in I$ pero no se cumple que $f \to^*_{\leq,G} 0$.

Por **(4)** sea $g \in G$ tal que $\mathrm{lm}_\leq(g) | \mathrm{lm}_\leq(f)$ y sean también:

- $a, b \in \langle X\rangle$ tales que $a\,\mathrm{lm}_\leq(g)b = \mathrm{lm}_\leq(f)$.
- $f' = f - \frac{f_{\mathrm{lm}_\leq(agb)}}{\mathrm{lc}_\leq(g)} agb$.

Notar que:

- $f' \in I$ ya que $f \in I$ y $g \in (G) = I$,
- $f \to_{\leq,G} f'$ por definición de $\to_{\leq,G}$ y
- $f' < f$ por el Teorema 2.32.

Ahora como no vale $f \to^*_{\leq,G} 0$, tampoco puede valer $f' \to^*_{\leq,G} 0$. Sin embargo, esto contradice que f sea mínimo.

(2) ⇒ (5): Supongamos **(2)** y tomemos $f \in I - \{0\}$.

Por **(2)** tenemos que $f \to^*_{\leq,G} 0$.

Sean:

- $n \in \mathbb{N}, f_0, \ldots, f_n \in K\langle X\rangle$ tales que $f_0 = f$, $f_n = 0$ y $f_0 \to_{\leq,G} f_1 \to_{\leq,G} \cdots \to_{\leq,G} f_n$, los cuales existen porque $f \to^*_{\leq,G} 0$ y por el Lema 2.2.
- Para cada $i \in \{1, \ldots, n\}$, $g_i \in G, a_i, b_i \in X$ tales que $\mathrm{lm}_\leq(a_i g_i b_i) \in \mathrm{sop}(f_{i-1})$ y $f_i = f_{i-1} - \frac{(f_{i-1})_{\mathrm{lm}_\leq(a_i g_i b_i)}}{\mathrm{lc}_\leq(g_i)} a_i g_i b_i$ los cuales existen por definición de $\to_{\leq,G}$.

Probaremos que el $\exists$ de **(5)**, se satisface con $c_i = \frac{(f_{i-1})_{\mathrm{lm}_\leq(a_i g_i b_i)}}{\mathrm{lc}_\leq(g_i)}, g_i, a_i, b_i$. O sea, tenemos que probar $\mathrm{lm}_\leq(a_i g_i b_i) \leq \mathrm{lm}_\leq(f)$ y $f = \sum_{i=1}^n \frac{(f_{i-1})_{\mathrm{lm}_\leq(a_i g_i b_i)}}{\mathrm{lc}_\leq(g_i)} a_i g_i b_i$.

Prueba de $\lm_\leq(a_i g_i b_i) \leq \lm_\leq(f)$ **fijando** i**:** Por el Teorema 2.32 y por transitividad de $<$ tenemos $f_{i-1} \leq f$. Además tenemos $a_i g_i b_i \leq f_{i-1}$ porque $\lm_\leq(a_i g_i b_i) \in \sop(f_{i-1})$, así que por transitividad de $\leq$ vale $\lm_\leq(a_i g_i b_i) \leq \lm_\leq(f)$.

Prueba de $f = \sum_{i=1}^{n} \frac{(f_{i-1})_{\lm_\leq(a_i g_i b_i)}}{\lc_\leq(g_i)} a_i g_i b_i$**:** Es consecuencia directa de como se eligieron los g_i, a_i, b_i.

(5) ⇒ (4): Supongamos **(5)** y probemos **(4)**. Para eso tenemos que probar $(G) = I$ y $\forall f \in I : \exists g \in G - \{0\} : \lm_\leq(g) | \lm_\leq(f)$.

Prueba de $(G) = I$**:** Tomemos $f \in I$ y probemos $f \in (G)$.

Por **(5)** tenemos que $f = \sum_{i=1}^{n} c_i a_i g_i b_i$ con $c_i \in K$, $a_i, b_i \in \langle X \rangle$ y $g_i \in G$.

Esto encaja exactamente con la definición de (G), así que queda probado $f \in (G)$.

Prueba de $\forall f \in I - \{0\} : \exists g \in G : \lm_\leq(g) | \lm_\leq(f)$**:** Fijemos $f \in I - \{0\}$.

Por **(5)** tenemos que $f = \sum_{i=1}^{n} c_i a_i g_i b_i$ con $c_i \in K$, $a_i, b_i \in \langle X \rangle$, $g_i \in G$ y $\lm_\leq(a_i g_i b_i) \leq \lm_\leq(f)$.

Como $\lm_\leq(a_i g_i b_i) \leq \lm_\leq(f)$, sí o sí tiene que pasar para algún j que $\lm_\leq(a_j g_j b_j) = \lm_\leq(f)$ y para este g_j vale que $\lm_\leq(g_j) | \lm_\leq(f)$.

Con esto se termina la prueba. □

Observación 2.44. ***(2)*** *del Teorema 2.43 nos da una manera precisa de responder al Problema 1. Para decidir si f está en el ideal generado por G, primero le calculamos una base de Gröbner al ideal. Luego si pudimos encontrar una base de Gröbner finita, aplicamos el Algoritmo 1 con esta base y si el resultado es 0 entonces f está en el ideal.*

Combinando esta observación con el Lema 2.25 tenemos el siguiente colorario.

Colorario 2.45. *Sean $G, G' \subseteq K\langle X \rangle$ bases de Gröbner, entonces*

$$(G) = (G') \Leftrightarrow (\forall g \in G : g \to_{\leq, G'} 0) \wedge (\forall g' \in G' : g' \to_{\leq, G} 0)$$

□

2.5 Algoritmo de Buchberger

En esta sección se explica el primer algoritmo para calcular bases de Gröbner llamado Algoritmo de Buchberger.

Antes de poder calcular bases de Gröbner algo que estaría bueno hacer es poder computar si un conjunto es una base de Gröbner, porque ninguna de las equivalencias del Teorema 2.43 es directamente calculable ya que hacen cuantificaciones sobre conjuntos infinitos. Para eso definiremos algo llamado S-polinomio de dos polinomios y los usaremos para enunciar un teorema que nos da una forma computable de determinar si un conjunto es una base de Gröbner. Los S-polinomios entre dos polinomios multiplican cada polinomio por monomios a cada lado y por un escalar de forma que los monomios principales se cancelen. O sea, para polinomios f y g tendremos una cuenta $kafb - k'cgd$ de forma que se cancelen los monomios principales.

Primero definimos las ambigüedades, que representan los polinomios para los cuales puede pasar eso.

Definición 2.46. *Sean $m, m', a, b, c, d \in \langle X \rangle$. Se define*

$$(a, b, c, d, m, m') \text{ es una ambigüedad} \Leftrightarrow amb = cm'd.$$

La ambigüedad (a, b, c, d, m, m') se define como:

- *de superposición* $\Leftrightarrow (a = \varepsilon = d \wedge |b| < |m'| \wedge |c| < |m|) \vee (b = \varepsilon = c \wedge |a| < |m'| \wedge |d| < |m|)$.

- *de inclusión* $\Leftrightarrow a = \varepsilon = b \vee c = \varepsilon = d$.
- *relevante* $\Leftrightarrow$ *es de superposición o de inclusión.*

Si $f, f' \in K\langle X\rangle$ *se define que* (a,b,c,d,f,f') *es una ambigüedad si y solo si* $(a,b,c,d,\mathrm{lm}_{\leq}(f), \mathrm{lm}_{\leq}(f'))$ *es una ambigüedad y lo mismo para ambigüedades de superposición, de inclusión y relevantes.*

Además se define

$$\mathrm{amb}(f,f') = \{(a,b,c,d,f,f') : (a,b,c,d,f,f') \text{ es una ambigüedad relevante}\}.$$

Finalmente para $F \subseteq K\langle X\rangle$ *se define*

$$\mathrm{amb}(F) = \bigcup_{f,f' \in F - \{0\}} \mathrm{amb}(f,f').$$

Notar que por cómo es la definición de ambigüedad, cuando es relevante siempre hay una parte de m y una parte de m' que son iguales y que no están en los monomios a, b, c y d. Los casos relevantes son los únicos importantes, y por eso son los que se usan para definir amb. No relevantes siempre hay entre cualquier par de monomios, por ejemplo tomando $a = d = \varepsilon$, $b = m'$ y $c = m$.

Definición 2.47. *Sean* $a,b,c,d \in \langle X\rangle, f, f' \in K\langle X\rangle$ *y* $\alpha = (a,b,c,d,f,f')$ *una ambigüedad. Se define el S-polinomio de* α *como*

$$S(\alpha) = \frac{afb}{\mathrm{lc}_{\leq}(f)} - \frac{cgd}{\mathrm{lc}_{\leq}(f')}.$$

Por ejemplo, si $f = 2cba + 3dbcda$ y $f' = b + bcd$, entonces una ambigüedad de inclusión sería

$$\alpha = (\varepsilon, \varepsilon, d, a, f, f').$$

Siendo su S-polinomio

$$S(\alpha) = \frac{f}{3} - df'a = \frac{2}{3}cba - dba.$$

Y una ambigüedad de superposición sería

$$\alpha' = (bc, \varepsilon, \varepsilon, bcda, f, f').$$

Siendo su S-polinomio

$$S(\alpha') = \frac{bcf}{3} - f'bcda = bbcda - \frac{2}{3}bccba.$$

Ahora probaremos que efectivamente los monomios principales se cancelan y después enunciaremos el teorema para decidir si un conjunto es una base de Gröbner.

Lema 2.48. *Sea* $\alpha = (a,b,c,d,f,g)$ *una ambigüedad. Entonces:*

$$\mathrm{lm}_{\leq}(afb) = \mathrm{lm}_{\leq}(cgd).$$

Demostración: Las siguientes igualdades siguen por la definición de ambigüedad tanto para monomios como para polinomios.

$$\mathrm{lm}_{\leq}(afb) = a\,\mathrm{lm}_{\leq}(f)b = c\,\mathrm{lm}_{\leq}(g)d = \mathrm{lm}_{\leq}(cgd).$$

□

Eso permite definir el monomio principal de una ambigüedad.

Definición 2.49. *Sea $\alpha = (a, b, c, d, f, g)$ una ambigüedad. Se define el monomio principal de α como*

$$\mathrm{lm}_{\leq}(\alpha) = \mathrm{lm}_{\leq}(afb).$$

Notar que por el Lema 2.48 también vale que $\mathrm{lm}_{\leq}(\alpha) = \mathrm{lm}_{\leq}(cgd)$.

Lo de que los monomios principales se cancelen produce el siguiente lema.

Lema 2.50. *Sea $\alpha = (a, b, c, d, f, g)$ una ambigüedad. Entonces*

$$\mathrm{lm}_{\leq}(S(\alpha)) < \mathrm{lm}_{\leq}(\alpha).$$

Demostración: Esto es porque en la resta $\frac{afb}{\mathrm{lc}_{\leq}(f)} - \frac{cgd}{\mathrm{lc}_{\leq}(g)}$ los monomios principales se cancelan. □

Otra propiedad importante es que estos S-polinomios son cerrados en el ideal, es decir, que el S-polinomio de dos elementos de un ideal es un elemento del ideal.

Lema 2.51. *Sean $I \subseteq K\langle X\rangle$ un ideal $f, g \in I$ y $\alpha = (a, b, c, d, f, g)$ una ambigüedad. Entonces*

$$S(\alpha) \in I.$$

Demostración: En la definición de S se ve que es una combinación lineal de f y g con elementos de $K\langle X\rangle$. Más precisamente, con los elementos $\frac{a}{\mathrm{lc}_{\leq}(f)}$, b, $\frac{c}{\mathrm{lc}_{\leq}(g)}$ y d. Como $f, g \in I$ e I es un ideal, esa combinación pertenece a I. □

Ahora si enunciamos el teorema para decidir si un conjunto es una base de Gröbner.

Teorema 2.52 (Condición de Buchberger)**.** *Sean I un ideal de $K\langle X\rangle$ y $G \subseteq K\langle X\rangle$. Entonces son equivalentes:*

(1) *G es una base de Gröbner de I.*

(2) $\forall\, \alpha \in \mathrm{amb}(G) : S(\alpha) \to^{*}_{\leq, G} 0$.

□

La demostración se puede encontrar por ejemplo en [Hof23], en donde algo similar a esto está enunciado en el Teorema 2.4.54. No lo demostramos acá porque requiere varios lemas adicionales que no son pertinentes para esta tesis.

Con esto ya se puede explicar la idea del algoritmo de Buchberger. Supongamos que tenemos un conjunto que no sabemos si es base de Gröbner o no y estamos chequeando si lo es con el Teorema 2.52. Si nos encontramos con un S-polinomio f que se reduce a un polinomio g distinto de 0 podemos agregar g al conjunto y con eso f pasa a sí reducirse a 0. Se podría pensar que agregar g cambiaría el ideal generado, pero por el Lema 2.51 sabemos que g es un elemento del ideal y por el Lema 2.36 eso implica que f también pertenece al ideal. Por ende, por el Lema 2.24 se puede agregar al conjunto y que el ideal generado siga siendo el mismo. El problema de agregar un nuevo polinomio es que si bien hay un S-polinomio que pasa a sí reducirse a 0 también aparecen nuevas ambigüedades. Eso, sin embargo, no es tanto problema porque lo que se hace es repetir hasta que en algún momento todas las ambigüedades se reducen a 0. Si eso no pasa nunca simplemente el algoritmo no termina nunca, lo que refleja la no decidibilidad del problema de pertenencia a un ideal.

Ese proceso de agregar polinomios infinitamente lo definimos matemáticamente del siguiente modo.

Definición 2.53. *Sean $G \subseteq K\langle X\rangle$ y $e_{\leq}$ un reductor. Se definen:*

- $\mathrm{B}^{0}_{e_{\leq}}(G) = G$.
- $\mathrm{B}^{i+1}_{e_{\leq}}(G) = \mathrm{B}^{i}_{e_{\leq}}(G) \cup \{e_{\leq}(\mathrm{B}^{i}_{e_{\leq}}(G))(S(\alpha)) : \alpha \in \mathrm{amb}(\mathrm{B}^{i}_{e_{\leq}}(G))\}$.

- $\mathrm{B}_{e_\leq}(G) = \bigcup_{i=0}^{\infty} \mathrm{B}^i_{e_\leq}(G)$.

A $\mathrm{B}_{e_\leq}(G)$ *lo llamamos base de Buchberger de* (G).

El nombre base de Buchberger es inventado para esta tesis, otros autores no le han dado ningún nombre concreto a esos conjuntos.

Con el siguiente teorema se enuncia que la base de Buchberger es una base de Gröbner y que si hay una base de Gröbner finita entonces en alguna iteración finita se llega, o sea, algún $B^i_{e_\leq}(G)$ es igual a la base de Buchberger.

Teorema 2.54. *Sean* $G \subseteq K\langle X\rangle$ *y* $e_\leq$ *un reductor. Entonces*

(1) $\mathrm{B}_{e_\leq}(G)$ *es una base de Gröbner de* (G).

(2) (G) *tiene una base de Gröbner finita* $\Rightarrow \exists i \in \mathbb{N} \cup \{0\} : (\mathrm{B}^i_{e_\leq}(G))$ *es una base de Gröbner.*

Es esperable que **(1)** sea cierto por cómo definimos la base de Buchberger, mientras que **(2)** es más sorprendente. Para demostrar ambos ítems primero probaremos algunos lemas en el contexto de este teorema, o sea con G y $e_\leq$ fijados.

Lema 2.55. $\forall i \in \mathbb{N} \cup \{0\} : \mathrm{B}^i_{e_\leq}(G) \subseteq (G)$.

Demostración: Procedamos por inducción sobre i. El caso base se cumple porque $G \subseteq (G)$. Para el caso inductivo, supongamos que la afirmación es cierta para i y probemos que también vale para $i+1$. Para eso tomemos $f \in \mathrm{B}^{i+1}_{e_\leq}(G)$ y probemos que $f \in (G)$.

Por la definición recursiva de $\mathrm{B}^{i+1}_{e_\leq}$ vale que $f \in \mathrm{B}^i_{e_\leq}(G) \vee \exists \alpha \in \mathrm{amb}(\mathrm{B}^i_{e_\leq}(G)) : f = e_\leq(\mathrm{B}^i_{e_\leq}(G))(\mathrm{S}(\alpha))$. Dividamos en casos según ese $\vee$.

Caso $f \in \mathrm{B}^i_{e_\leq}(G)$**:** Es válido por ser exactamente la hipótesis inductiva.

Caso $\exists \alpha \in \mathrm{amb}(\mathrm{B}^i_{e_\leq}(G)) : f = e_\leq(\mathrm{B}^i_{e_\leq}(G))(\mathbf{S}(\alpha))$**:** En tal caso, por el Lema 2.40 vale que $f \in \mathrm{B}^i_{e_\leq}(G)$ y por la hipótesis inductiva que $f \in (G)$.

□

Lema 2.56. $\mathrm{B}_{e_\leq}(G) \subseteq (G)$.

Demostración: Esto es consecuencia directa del Lema 2.55 y la definición de $\mathrm{B}_{e_\leq}$. □

Lema 2.57. $(\mathrm{B}_{e_\leq}(G)) = (G)$.

Demostración: Por la definición de $\mathrm{B}^i_{e_\leq}$ tenemos $G \subseteq \mathrm{B}^i_{e_\leq}(G)$ y por ende $G \subseteq (\mathrm{B}_{e_\leq}(G))$. Con el Lema 2.56 aplicando el Lema 2.25 tenemos lo que queremos probar. □

Lema 2.58. $\forall \alpha \in \mathrm{amb}(\mathrm{B}_{e_\leq}(G)) : S(\alpha) \to^*_{\leq, \mathrm{B}_{e_\leq}(G)} 0$.

Demostración: Tomemos $\alpha = (a, b, c, d, f, g) \in \mathrm{amb}(\mathrm{B}_{e_\leq}(G))$. Por definición de amb vale que $f, g \in \mathrm{B}_{e_\leq}(G)$, así que sean:

- $i \in \mathbb{N}$ el mínimo tal que $f \in \mathrm{B}^i_{e_\leq}(G)$,
- $j \in \mathbb{N}$ el mínimo tal que $g \in \mathrm{B}^j_{e_\leq}(G)$ y
- $k = \max(i, j)$.

Por definición de amb tiene que valer que

$$\alpha \in \text{amb}(\text{B}^k_{e_\leq}(G))$$

y entonces por definición de $\text{B}^i_{e_\leq}$ vale que

$$e_\leq(\text{B}^k_{e_\leq}(G))(\text{S}(\alpha)) \in \text{B}^{k+1}_{e_\leq}(G).$$

Esto implica que

$$e_\leq(\text{B}^k_{e_\leq}(G))(\text{S}(\alpha)) \to_{\leq,\text{B}^{k+1}_{e_\leq}(G)} 0$$

y por ende que

$$e_\leq(\text{B}^k_{e_\leq}(G))(\text{S}(\alpha)) \to_{\leq,\text{B}_{e_\leq}(G)} 0.$$

Llamemos (i) a esto último. Por otro lado, por definición de reductor tenemos que

$$\text{S}(\alpha) \to^*_{\leq,\text{B}^k_{e_\leq}(G)} e_\leq(\text{B}^k_{e_\leq}(G))(\text{S}(\alpha))$$

y por ende

$$\text{S}(\alpha) \to^*_{\leq,\text{B}_{e_\leq}(G)} e_\leq(\text{B}^k_{e_\leq}(G))(\text{S}(\alpha)).$$

Llamemos (ii) a esto último. Combinando (i) y (ii) vale que $\text{S}(\alpha) \to^*_{\leq,\text{B}_{e_\leq}(G)} 0$, que es lo que queríamos probar. □

Con estos lemas ahora sí hagamos la demostración del Teorema 2.54.

Demostración del Teorema 2.54: Por el Lema 2.57 y el Lema 2.58 vale la equivalencia de base de Gröbner de la condición de Buchberger (Teorema 2.52), así que **(1)**, o sea, que G es una base de Gröbner queda probado.

Para probar **(2)**, asumamos el antecedente, o sea, que (G) tiene una base de Gröbner finita, y tomemos una base finita $\{g_1, ..., g_n\}$. Ahora tenemos que probar el consecuente, o sea, que $\exists i \in \mathbb{N} \cup \{0\} : (\text{B}^i_{e_\leq}(G))$ es una base de Gröbner. Para cada $i \in \{1, ..., n\}$ sean:

- $g_{i,1}, ..., g_{i,k_i} \in \text{B}_{e_\leq}(G), a_{i,1}, ..., a_{i,k_i}, b_{i,1}, ..., b_{i,k_i} \in \langle X \rangle$ con $\text{lm}_\leq(a_{i,j} g_{i,j} b_{i,j}) \leq \text{lm}_\leq(g_i)$ tales que $g_i = \sum_{j=1}^{k_i} a_{i,j} g_{i,j} b_{i,j}$. Los cuales existen por por **(5)** del Teorema 2.43.
- $G' = \{g_{i,j} : i \in \{1, ..., n\}, j \in \{1, ..., k_i\}\}$.
- $k \in \mathbb{N}$ el mínimo tal que $G' \subseteq \text{B}^k_{e_\leq}(G)$. Notar que k está bien definido porque G' es finito y los $g_{i,j}$ están en $\text{B}_{e_\leq}(G)$.

Vamos a probar que $\text{B}^k_{e_\leq}(G)$ es una base de Gröbner de (G). Por **(5)** del Teorema 2.43 alcanza con probar:

$$\forall f \in (G) - \{0\} : \exists g'_1, ..., g'_n \in \text{B}^k_{e_\leq}(G), a_1, ..., a_n, b_1, ..., b_n \in \langle X \rangle : \text{lm}_\leq(a_i g'_i b_i) \leq \text{lm}_\leq(f) \wedge f = \sum_{i=1}^{n} a_i g'_i b_i.$$

Tomemos $f \in (G) - \{0\}$ y escribámoslo de esa forma. Por **(5)** del Teorema 2.43 y el hecho de que $\{g_1, ..., g_n\}$ es una base de Gröbner, sean $i'_1, ..., i'_{n'} \in \{1, ..., n\}, a'_1, ..., a'_{n'}, b'_1, ..., b'_{n'} \in \langle X \rangle$ tales que:

- $\text{lm}_\leq(a'_i g_{i'_i} b'_i) \leq \text{lm}_\leq(f)$ y
- $f = \sum_{i=1}^{n'} a'_i g_{i'_i} b'_i$.

Con esto tenemos:

$$\begin{aligned} f &= \sum_{i=1}^{n'} a'_i g_{i'_i} b'_i \\ &= \sum_{i=1}^{n'} a'_i \Big(\sum_{j=1}^{k_{i'_i}} a_{i',j} g_{i,j} b_{i',j}\Big) b'_i \qquad \text{Condición de } g_{i,j} \\ &= \sum_{i=1}^{n'} \sum_{j=1}^{k'_i} a'_i a_{i',j} g_{i,j} b_{i',j} b'_i. \end{aligned}$$

Sabemos además que $g_{i',j} \in \mathrm{B}^k_{e_\leq}(G)$ y que $g_{i',j} \leq g_{i'_i} \leq f$. Así que queda f escrito como queríamos y la prueba queda completada. □

Con estos conjuntos podemos de forma directa dar un seudocódigo que los calcula:

Algoritmo 2: Algoritmo de Buchberger

Data: $G \subseteq K\langle X\rangle$, $e_\leq$ un reductor
Result: $B \subseteq K\langle X\rangle$ una base de Gröbner de (G) si es que termina

```
1  B ← G
2  loop
3      ambs ← amb(B)
4      B' ← B
5      for α ∈ ambs do
6          f ← e_≤(B)(S(α))
7          if f ≠ 0 then
8              B' ← B' ∪ {f}
9      if B' = B then
10         break
11     B ← B'
12 return B
```

El algoritmo así, si bien es una implementación literal de la definición de los conjuntos $\mathrm{B}^i_{e_\leq}$ que probamos que es correcta, es muy lento y necesita mejoras. Una mejora muy importante, probablemente la más importante, consiste en usar también los nuevos polinomios que ya fueron agregados para hacer las nuevas reducciones en el paso. Es decir, llamar a $e_\leq$ con B' en lugar de con B en la línea 6.

Por lo visto antes, el cambio tiene sentido que sea correcto y la prueba es similar a la del Teorema 2.54. En la práctica, esta mejora hace una diferencia inmensa en la eficiencia, pasa de no andar rápido casi nunca a andar rápido en una gran cantidad de casos. La justificación teórica de por qué hacer eso hace que pase a ser tanto más rápido no la sé ni la pude encontrar en la literatura.

Hay otras dos optimizaciones que se pueden hacer: una es para descartar algunas ambigüedades sin tener que reducirlas y la otra es para ir sacando algunos polinomios de la base. En la Sección 4.7 se aborda este tema.

2.6 Algoritmo F4

En esta sección se explica el algoritmo F4, que es otro método para calcular bases de Gröbner. La idea de F4 es considerar varias ambigüedades y reducir todos sus S-polinomios al mismo tiempo, reduciendo con los elementos que ya están en la base y entre los propios S-polinomios que se están considerando. Para esto se convierte el problema en una reducción

por filas de una matriz, codificando cada monomio como una columna y cada polinomio como una fila. Para codificar los polinomios como filas de una matriz hacen falta algunas definiciones.

Definición 2.59. *Sea $F \subseteq K\langle X\rangle$. Se define*

$$\operatorname{span}_K(F) = \{\sum_{i=1}^{n} c_i f_i : n \in \mathbb{N}, c_i \in K, f_i \in F\}.$$

Definición 2.60. *Sean $M = \{m_1, \dots, m_n\} \subseteq \langle X\rangle$ con $m_1 > \cdots > m_n$. Se define la función $\operatorname{poli}_M : K^n \to \operatorname{span}_K(M)$ de la siguiente manera:*

$$\operatorname{poli}_M(v) = \sum_{i=1}^{n} v_i m_i.$$

Definición 2.61. *Sean $M = \{m_1, \dots, m_n\} \subseteq \langle X\rangle$ con $m_1 > \cdots > m_n$. Se define $\operatorname{mat}_M : \operatorname{span}_K(M) \to K^n$ como la inversa de poli_M:*

$$\operatorname{mat}_M(f) = \operatorname{poli}_M^{-1}(f).$$

Para una lista de polinomios se define que mat_M produzca una matriz:

$$\operatorname{mat}_M(f_1, \dots, f_m) = \begin{pmatrix} \operatorname{mat}_M(f_1) \\ \vdots \\ \operatorname{mat}_M(f_m) \end{pmatrix}.$$

Estas definiciones son un diccionario polinomios-matrices y matrices-polinomios en la que cada monomio se corresponde con una columna de la matriz.

Con esto, en la reducción, restar un polinomio multiplicado por un escalar y un monomio a cada lado a otro polinomio, o sea, hacer $f \leftarrow f - \frac{f_{\operatorname{lm}_\le(g)}}{\operatorname{lc}_\le(g)} agb$ al reducir f con g, la idea es que pase a ser restarle una fila multiplicada por un escalar a otra fila en la matriz, o sea, hacer $\text{fila}_i \leftarrow \text{fila}_i - c \cdot \text{fila}_j$.

Para poder hacer eso, al comenzar hay que tener en la matriz los polinomios a restar ya multiplicados por los monomios a cada lado, es decir, cuando haga falta hacer $f \leftarrow f - \frac{f_{\operatorname{lm}_\le(g)}}{\operatorname{lc}_\le(g)} agb$ hay que tener agb en la matriz. Tener solo g en la matriz no serviría porque no coincidirían los monomios y por ende no coincidirían las columnas de la matriz.

Para eso definimos el algoritmo de preprocesamiento simbólico que, dado el conjunto por el cual se reduce G y el conjunto a reducir F, calcula todos los polinomios necesarios:

Algoritmo 3: Preprocesamiento simbólico

Data: $G, F \subseteq K\langle X\rangle$ conjuntos finitos
Result: $G' \subseteq \{agb : a, b \in \langle X\rangle, g \in G\}$

```
1  G' ← ∅
2  considerados ← ⋃_{g∈P} sop(g)
3  T ← ⋃_{g∈P} sop(tail_≤(g))
4  while T ≠ ∅ do
5      elegir m ∈ T
6      T ← T − {m}
7      for g ∈ G do
8          if lm_≤(g) | m then
9              calcular a y b tales que m = a lm_≤(g) b
10             G' ← G' ∪ agb
11             nuevos ← {m' ∈ sop(tail_≤(agb)) : m' ∉ considerados}
12             T ← T ∪ nuevos
13             considerados ← considerados ∪ nuevos
14 return G'
```

El seudocódigo se puede entender cómo funciona analizándolo en detalle, pero lo importante es saber que calcula los polinomios que van a hacer falta que sean las filas de la matriz, y no importa demasiado entender cómo funciona por dentro.

Una vez que tenemos los monomios necesarios, armamos la matriz correspondiente a G y F usando la función mat_T y reducimos por fila esta matriz. Con la matriz no es obvio qué hay que hacer, porque hay que solo considerar las filas que corresponden a los polinomios reducidos, pero no necesariamente son las mismas filas que eran antes los polinomios sin reducir porque la reducción por filas puede hacer cosas como intercambiar filas. Para saber cuáles son esas filas lo que se hace es guardar cuáles son los monomios principales de elementos del resultado del preprocesamiento simbólico y después de hacer la reducción por filas agarrar solo las filas que tengan un uno principal en una columna que no corresponda a uno de esos monomios principales.

El siguiente seudocódigo hace eso:

Algoritmo 4: Multireducción

Data: $G, F \subseteq K\langle X\rangle$ conjuntos finitos
Result: $F' \subseteq K\langle X\rangle$

$G' \leftarrow$ Preprocesamiento simbólico(G, F)
monomios $\leftarrow \{\mathrm{lm}_{\leq}(g) : g \in G'\}$
$M \leftarrow \{m : m \in \mathrm{sop}(g), g \in G' \cup F\}$
mat $\leftarrow \mathrm{mat}_M(G' \cup F)$
reducir por filas *mat*
$F' \leftarrow \{\mathrm{poli}_M(v) : v \in \mathrm{filas}(\textit{mat}) : v \neq 0 \wedge \mathrm{lm}_{\leq}(\mathrm{poli}_M(v)) \notin \textit{monomios}\}$
return F'

Con estos dos algoritmos ya podemos escribir el algoritmo F4:

Algoritmo 5: F4

Data: $G \subseteq K\langle X\rangle$, $e_{\leq}$ un reductor
Result: $B \subseteq K\langle X\rangle$ una base de Gröbner de (G) si es que termina

$B \leftarrow G$
loop
    ambs $\leftarrow \mathrm{amb}(B)$
    $B' \leftarrow B$
    while *ambs* $\neq \emptyset$ **do**
        elegir $A \subseteq$ *ambs*
        ambs $\leftarrow$ *ambs* $- A$
        $P \leftarrow \{\mathrm{S}(\alpha) : \alpha \in A\}$
        $P \leftarrow$ Multireducción(B', P)
        $B' \leftarrow B' \cup P$
    if $B' = B$ **then**
        break
    $B \leftarrow B'$
return B

Este algoritmo no especifica cuáles ambigüedades elegir, pero una buena estrategia es elegir todas las de menor grado. Otra opción es elegir todas, pero eso puede hacer que las matrices se vuelvan muy grandes rápidamente.

En [Hof20] se da un algoritmo muy parecido a este (con una diferencia mínima) y se demuestra que es correcto.

Las optimizaciones mencionadas en la sección de Buchberger para descartar ambigüedades y para quitar polinomios de la base también se pueden hacer acá. Otra optimización que se puede hacer es usar un algoritmo de reducción de matrices que anda más rápido en las matrices de este problema en particular, llamado eliminación Faugère-Lachartre. Esto también es abordado en la Sección 4.7.

Capítulo 3
Librería

Como se dijo en la introducción, para este trabajo se implementaron los algoritmos de Buchberger y F4 en C++ junto con estructuras para manejar polinomios no conmutativos en una librería llamada `ncgb`. La librería está disponible en el repositorio `https://github.com/IvanRenison/Non-commutative-Grobner-Bases`.

En este capítulo se explica cómo se usa la librería sin explicar como está implementada y en el siguiente se explica cómo está implementada. En la fecha de la escritura de esta tesis, el último commit tiene el tag `v0.1` y todas las explicaciones de esta tesis se refieren esa versión.

El repositorio tiene varios directorios, pero hay dos que son los más relevantes. El primero es el directorio `ncgb`, que es donde está alojada la librería en sí. Como la librería es genérica sobre algunos parámetros, se usan templates. Las clases y funciones genéricas se definen en archivos `.hpp`, donde se incluye tanto su declaración como su implementación. El otro directorio relevante es `mains`, que tiene varios archivos `.cpp` con funciones `main` que usan la librería. Estos archivos son útiles para ver ejemplos de cómo usar la librería.

Todo el código de la librería está en el namespace `ncgb`. En todos los códigos que veamos asumiremos que `ngcb` está abierto.

3.1 Uso general de la librería

Los polinomios como los definimos en el Capítulo 2 están asociados a un cuerpo y un conjunto de variables; además, muchas operaciones y métodos de los polinomios (y de otras definiciones) dependen de un orden monomial. Para que se puedan variar esas tres cosas se usan templates para hacer las funciones y estructuras de forma genérica. Por ejemplo, el esqueleto de la definición de polinomio es así:

```
template<typename K, typename X = __uint8_t, class ord = DegLexOrd<X>>
struct Poly {
  // … Implementación
};
```

El tipo `K` tiene que tener implementadas todas las operaciones de cuerpo. En particular, la librería asume que tiene definidas las siguientes operaciones y construcciones:

- `+`, `-` (tanto unario como binario), `*` y `/` junto con sus versiones de asignación `+=`, `-=`, `*=` y `/=`. De las versiones de asignación no se usa el resultado, solo el hecho de que modifican el operando izquierdo, así que pueden devolver `void`.
- `==` (y también `!=`, pero ese no hace falta definirlo a mano, porque a partir de C++20 está automáticamente con `==`).
- `<<` para imprimir y `>>` para leer.
- Las expresiones `K(0)` y `K(1)` son validas.

Y como es esperable, las operaciones tienen que cumplir todos los axiomas de cuerpos.

En este trabajo se usaron dos cuerpos distintos, los números racionales, usando el tipo `mpq_class` de la librería GMP que cumple con todos los requisitos de `K` [GMP] y la aritmética modular con una implementación propia en el archivo `extras/ModularArithmetic.hpp`, que sirve para ver un ejemplo de un cuerpo que funciona con la librería (aunque tiene algunas operaciones extra).

El tipo `X` se usa para representar las variables como números y la idea es que se use algún tipo de números sin signo. Usar otros tipos podría funcionar pero no está probado. Por defecto se usa `__uint8_t` que permite tener hasta 256 variables. Para usar más variables hay que cambiarlo a un entero sin signo más grande como `__uint32_t`.

Por último, `ord` es el orden monomial que se usa y está puesto por defecto `DegLexOrd` que es el orden lexicográfico por grado de la Definición 2.12 y ya está definido junto con la definición de monomios. Para usar otro orden hay que definir una estructura que tenga un operador `()` que implemente el testeo del orden. Esta es la forma estándar de definir órdenes para usar en templates en C++. Algo así podría ser el código si es una implementación para un `X` genérico:

```
template<typename X>
struct Orden {
  bool operator()(const Monomial<X>& a, const Monomial<X>& b) const {
    // … Implementación
  }
};
```

Tanto para `X` como para `ord` los valores por defecto están puestos solo en las estructuras de monomios y polinomios, porque en las otras funciones y estructuras no hace falta ya que para usarlas hay que ya tener un monomio o un polinomio (que se pase como argumento) y de ahí C++ infiere `X` y `ord` automáticamente.

La librería también tiene un Makefile con el que se pueden compilar los archivos del directorio `mains` y correr los tests. Este Makefile puede servir de ejemplo de cómo se compila un archivo que usa la librería.

3.2 Monomios

Los monomios están definidos en el archivo `ncgb/nc_monomial.hpp` y lo más importante de su funcionalidad se puede resumir así:

```
template<typename X = __uint8_t>
struct Monomial {
  Monomial();
  Monomial(const std::vector<X>& vals);

  bool operator==(const Monomial& m) const;
  Monomial operator*(const Monomial& m) const;
  void operator*=(const Monomial& m);
  size_t size() const;
  bool empty() const;

  // Returns the positions of m.vals where this monomial divides m
  std::vector<size_t> divide_indexes(const Monomial& m) const;

  // Does this monomial divides m?
  bool divides(const Monomial& m) const;

  // Make all possible divisions of m by this monomial
  std::vector<std::pair<Monomial, Monomial>> divide(const Monomial& m) const;

  friend std::ostream& operator<<(std::ostream& os, const Monomial& m);
  friend std::istream& operator>>(std::istream& is, Monomial& m);

  void nice_print(std::ostream& os = std::cout) const;
```

```
  static Monomial nice_read(std::istream& is = std::cin);
};
```

El único argumento del template es `X`, porque no hace falta nada del cuerpo ni tampoco un orden monomial (de hecho, no tendría sentido que haga falta un orden monomial para definir monomios).

Relacionado a la división, el método más importante es `divide_indexes` que se llama como `m0.divide_indexes(m1)` y devuelve un vector de `size_t` que indica las posiciones de `m1` donde empieza una sub-palabra igual a `m0`.

Respecto a la representación como strings, hay dos pares de métodos. Por un lado están `operator>>` y `operator<<` que leen y escriben en un formato numérico que es cómodo para usar en código. Por otro lado, `nice_read` y `nice_print` que leen y escriben en el formato que se suele usar al escribir a mano los monomios usando las variables a, b, etc.

El formato numérico consiste en un número entero no negativo n seguido de n números x_1, ..., x_n que son los números de variables. Cuando $n = 0$ no hay ningún x_i. Se puede representar así el formato:

$$n \quad x_1 \quad \cdots \quad x_n$$

Por ejemplo, acá hay un monomio en el formato como se suelen escribir a mano:

```
adbda
```

Y acá está el mismo monomio en el formato numérico:

```
5 0 3 1 3 0
```

Para declarar un monomio con `X = __uint8_t`, leerlo en formato numérico e imprimirlo en formato bonito se puede hacer así:

```
Monomial m;
std::cin >> m;
m.nice_print();
```

En el mismo archivo `ncgb/nc_monomial.hpp` está definido el orden lexicográfico por grado `DegLexOrd`.

3.3 Polinomios

Los polinomios están definidos en el archivo `ncgb/nc_polynomial.hpp` y lo más importante de su funcionalidad se puede resumir así:

```
template<typename K, typename X = __uint8_t, class ord = DegLexOrd<X>>
struct Poly {
  Poly();
  Poly(const Monomial<X>& m, K c = K(1));
  Poly(std::vector<std::pair<Monomial<X>, K>> p);

  bool operator==(const Poly& p) const;

  Poly operator+(const Poly& p) const;
  Poly operator-(const Poly& p) const;
  Poly operator*(const Poly& p) const;
  Poly operator*(Monomial<X> m) const;
  Poly operator*(K c) const;
  Poly operator/(K c) const;
  Poly operator-() const;
  void operator+=(const Poly& p);
  void operator-=(const Poly& p);
```

```
  void operator*=(const Poly& p);
  void operator*=(Monomial<X> m);
  void operator*=(K c);

  K coeff(const Monomial<X>& m) const;

  const Monomial<X>& lm() const;
  K lc() const;
  Poly lt() const;
  bool monic() const;
  bool isZero() const;

  friend std::ostream& operator<<(std::ostream& os, const Poly& p);
  friend std::istream& operator>>(std::istream& is, Poly& p);

  void nice_print(std::ostream& os = std::cout) const;
  static Poly nice_read(std::istream& is = std::cin);
};
```

Para construir un polinomio hay tres constructores: el constructor vacío que produce el polinomio 0, un constructor que toma un monomio m y un elemento del cuerpo c y produce el polinomio cm, y un constructor que toma un vector de pares monomio-coeficiente y produce el polinomio que es la suma de cada coeficiente multiplicado con su monomio.

El tipo `Poly` tiene implementadas todas las operaciones entre polinomios (como la suma) con los operadores correspondientes (como `operator+` y `operator+=`, por ejemplo), algunas de las cosas definidas en la Definición 2.15 (`coeff`, `lm`, `lc` y `lt`) y métodos relacionados a la representación de los polinomios como strings.

Al igual que con los monomios, para la representación como strings, hay dos pares de métodos: `operator>>` y `operator<<` para formato numérico, y `nice_print` y `nice_read` para formato visualmente bonito.

El formato numérico consiste en, primero, un número entero no negativo k que es la cantidad de términos, seguido de la descripción de k términos. La descripción de cada término consiste en primero el coeficiente y después la descripción del monomio en el formato numérico de los monomios. Se puede representar así el formato:

$$
\begin{array}{l}
k \\
c_1 \quad n_1 \quad x_{1,1} \quad \cdots \quad x_{1,n_1} \\
\vdots \\
c_m \quad n_m \quad x_{k,1} \quad \cdots \quad x_{k,n_m}
\end{array}
$$

Por ejemplo, acá hay un polinomio en el formato como se suelen escribir a mano:

```
3 aaa - 5 bcc + adbda
```

Y acá está el mismo polinomio en el formato numérico:

```
3
3 3 0 0 0
-5 3 1 2 2
1 5 0 3 1 3 0
```

En el archivo `ncgb/nc_polynomial.hpp` también está definido el orden polinomial de la Definición 2.16 como `PolyOrd`.

Por ejemplo, para declarar un polinomio con `K = mpq_class`, `X = __uint8_t` y `ord = DegLexOrd<X>`, leerlo en formato numérico e imprimirlo en formato bonito se puede hacer así:

```
Poly<mpq_class> p;
std::cin >> p;
p.nice_print();
```

3.4 Buchberger y F4

Los algoritmos de Buchberger y F4 tienen una interfaz similar, así que por eso están explicados juntos.

Como ambos algoritmos tienen el problema de que pueden no terminar, no conviene hacer directamente una función que tome el conjunto generador y devuelva una base de Gröbner porque podría no terminar. Sí tiene sentido tener una función como esa para casos en los que se sabe que hay una base de Gröbner finita, pero no conviene que sea la única forma de usar los algoritmos.

Se podría hacer una función que además del conjunto generador tome un número que sea la cantidad de pasos a ejecutar y que además de devolver un conjunto devuelva si se llegó a una base de Gröbner o no. Esta opción tiene el problema de que una vez que la función retorna no se puede seguir con el cálculo, lo cual en algunos casos podría ser deseable.

Para evitar esos problemas se definió una estructura cuyo constructor toma a los polinomios y tiene un método `next` que calcula un paso más del cálculo de la base de Gröbner y en caso de haber terminado lo indica, y un método `fullBase` para usar solo en el caso de que se sepa que hay una base finita, que hace las llamadas a `next` hasta que termina y devuelve la base.

Para Buchberger la estructura dicha está en el archivo `ncgb/Buchberger.hpp` y se llama `BuchbergerIncremental` y para F4 la estructura está en el archivo `ncgb/F4.hpp` y se llama `F4Incremental`. En el caso de Buchberger el método `next` devuelve un `std::optional<Poly<K, ord>>` que es vacío solo en el caso de que ya se haya llegado a una base de Gröbner y si no tiene un polinomio de la base de Gröbner. En el caso de F4 el método `next` devuelve un `std::vector<Poly<K, ord>>` que es vacío solo en el caso de que ya se haya llegado a una base de Gröbner y si no tiene varios polinomios de la base de Gröbner. En ambos casos devolver vacío es la forma de decir que ya terminó el algoritmo.

La interfaz de Buchberger podría describirse así:

```
template<typename K, typename X, class ord>
struct BuchbergerIncremental {
  BuchbergerIncremental(const std::vector<Poly<K, X, ord>>& G);
  std::optional<Poly<K, X, ord>> next();
  std::vector<Poly<K, X, ord>> fullBase();
};
```

Y la de F4 así:

```
template<typename K, typename X, class ord>
struct F4Incremental {
  F4Incremental(const std::vector<Poly<K, X, ord>>& G);
  std::vector<Poly<K, X, ord>> next();
  std::vector<Poly<K, X, ord>> fullBase();
};
```

En ambos casos el conjunto generador se toma como un vector porque usar un set de C++ sería más costoso innecesariamente.

Para ambos algoritmos hay además una función `inIdeal` que toma un conjunto generador, un polinomio y una cantidad de pasos y trata de decir si el polinomio está en el ideal generado por el conjunto generador haciendo esa cantidad de llamadas a `next`. Esa función devuelve un elemento del siguiente tipo (con los significados que están comentados):

```
enum IdealMembershipStatus {
  InIdeal,    // The element is definitely in the ideal
  NotInIdeal,// The element is definitely not in the ideal
  Unknown     // More steps needed to determine if the element is in the ideal
};
```

Y está declarada así:

```
template<typename K, typename X, class ord>
IdealMembershipStatus
inIdeal(const std::vector<Poly<K, X, ord>>& G, Poly<K, X, ord> f, size_t st);
```

Esa función básicamente trata de responder el Problema 1 de si un polinomio está en el ideal generado por un conjunto generador.

3.5 Representación con cofactores

La función `inIdeal` solo decide la pertenencia al ideal generado, pero en caso afirmativo no da una forma de escribirlo como combinación lineal de elementos del conjunto generador con polinomios como coeficientes.

Con las bases de Gröbner pasa lo mismo, los algoritmos dan la base pero no dan una forma de escribir los elementos de la base usando los elementos del conjunto generador.

Sin embargo, esa información se puede calcular y ese cálculo está implementado para Buchberger en el mismo archivo `ncgb/Buchberger.hpp` en unas funciones y estructuras con nombres que terminan en `Cofactor`. Esta terminación es porque la forma de escribir un polinomio como combinación lineal de otros se llama representación con cofactores.

Estas funciones devuelven elementos del tipo `CofactorPoly` definido en el archivo `ncgb/nc_cofactorPolynomial.hpp`. Este tipo representa un polinomio como una combinación lineal de elementos del conjunto generador. En particular, guarda un vector de elementos m, i, m', c de forma que, si los g_i son el conjunto generador, el polinomio representado es $\sum m g_i m' c$.

Para F4 esto no está implementado (en el siguiente capítulo se explica más del porqué).

3.6 Comparación de bases de Gröbner

En el archivo `ncgb/cmpBases.hpp` está la función `cmpBases` que toma dos conjuntos generadores y, asumiendo que son bases de Gröbner, dice si generan el mismo ideal o no.

3.7 Paralelismo

Como ya se dijo antes, uno de los objetivos del trabajo fue paralelizar el cálculo de bases de Gröbner para que corra en varios hilos al mismo tiempo. La paralelización se hizo parcialmente para F4, o sea, se hizo solo para una parte del código, pero al ser solo parcialmente no hace casi diferencia en el tiempo, así que no sería muy útil usarla actualmente. De cualquier manera, en esta sección se explica cómo hacer que la parte del código que puede ejecutarse en paralelo lo haga.

La paralelización está hecha con OpenMP [OMP], que es una librería de C++ de paralelización, así que para que corra en paralelo es igual que en cualquier otro código paralelizado con OpenMP y a continuación se explica.

Para compilar para que se corra en paralelo hay que usar el flag `-fopenmp`. Solo con usar ese flag sigue igual corriendo con un solo hilo. Para que corra con varios hilos hay varias formas, la más fácil es incluir a OpenMP con `#include <omp.h>` y en el main llamar a la función `omp_set_num_threads` pasándole la cantidad de hilos con los que se quiere correr.

3.8 Ejemplo

Ahora un pequeño ejemplo de uso de la librería, con un ejemplo similar al del archivo `mains/base_Buchberger.cpp`:

```cpp
#include <bits/stdc++.h>
#include <gmpxx.h>
#include "ncgb/Buchberger.hpp"
using namespace std;
using namespace ncgb;

typedef Poly<mpq_class> P;

int main() {
  size_t n;
  cin >> n;
  vector<P> G(n);
  for (size_t i = 0; i < n; ++i) cin >> G[i];

  BuchbergerIncremental<P> bi(G);
  vector<P> base = bi.fullBase();

  cout << base.size() << endl;
  for (P& f : base) f.nice_print();
}
```

Este código, que trabaja sobre los racionales, lee un conjunto generador en formato numérico, le calcula una base de Gröbner, asumiendo que tiene una finita, y la imprime de forma bonita.

Capítulo 4
Implementación

En este capítulo se explican los detalles de cómo está hecha la implementación de la librería.

4.1 Monomios

Los monomios, o sea los elementos de $\langle X \rangle$, como son palabras se implementaron usando vectores de `X`. La base de la implementación es así:

```
template<typename X = __uint8_t>
struct Monomial {
  std::vector<X> vals;
  // … Métodos
};
```

Con esto, usando `X = __uint8_t`, que tiene 8 bits, se pueden tener hasta 256 variables. Cuando los monomios se imprimen o leen de forma bonita solo hay 26 variables, correspondiendo el 0 con la `a`. Si se quiere imprimir de forma bonita un monomio que usa variables mayores o iguales a 26, salta una aserción.

Esta estructura tiene implementadas las operaciones y métodos que se describieron en el capítulo anterior. La mayoría tienen una implementación directa. Las únicas no directas son las relacionadas a la división, porque chequear divisibilidad es chequear si una palabra es sub-palabra de otra, para lo cual, la forma directa de hacerlo llevaría tiempo cuadrático en el largo de las palabras, pero se puede hacer, y se hizo, en tiempo lineal.

Hay muchas formas de hacerlo en tiempo lineal, la que se usó es la función Z, que en el código está en el archivo `ncgb/Zfunc.hpp`. En [CPAZf] se explica la función Z y cómo usarla para chequear si una palabra es sub-palabra de otra.

Junto con la implementación de los monomios, en el archivo `ncgb/nc_monomial.hpp` está la implementación del orden lexicográfico por grado, que también es directa.

4.2 Polinomios

Los polinomios, o sea los elementos de $K\langle X \rangle$, están implementados usando un vector de pares monomio-coeficiente que siempre se mantiene ordenado por el orden monomial. El esqueleto de la implementación es así:

```
template<typename K, typename X = __uint8_t, class ord = DegLexOrd<X>>
struct Poly {
  std::vector<std::pair<Monomial<X>, K>> terms;
  // … Métodos
};
```

En esta estructura, `terms` es el vector de pares monomio-coeficiente mencionado. Con esta estructura para los polinomios todas las operaciones se hacen de forma directa.

4.3 Reducción

La reducción está implementada en el archivo `ncgb/reductions.hpp`, en particular en la función `reduce` que toma un polinomio y un vector de polinomios y reduce el polinomio con

los polinomios del vector. La reducción se hace modificando el propio argumento. Esa función sería una implementación de un reductor concreto, o sea, de un $e_{\leq}$ y trata siempre de reducir primero por los primeros elementos del vector y empezando por los monomios más grandes del polinomio.

La función `reduce` tiene además una versión alternativa que además toma un vector de booleanos que tiene que ser del mismo largo que el vector de polinomios (en C++ se puede tener varias funciones con el mismo nombre si tienen argumentos de distinto tipo) y hace la reducción solo con los polinomios que tengan un `false` en la misma posición en el vector de booleanos. Esta función está para poder implementar la optimización de ir eliminando algunos elementos de la base sin tener que estar modificando un vector.

4.4 Ambigüedades

Las ambigüedades, que son necesarias para el algoritmo de Buchberger, están implementadas en el archivo `ncgb/ambiguities.hpp` y consisten en una estructura así:

```
template<typename X>
struct Amb {
  const Monomial<X>& p, q;
  enum Type { Inclusion, Overlap };
  Type type;
  size_t pos; // position where q starts in p
  Monomial<X> a, b;

  size_t size() const;
  Monomial<X> lm() const;
};
```

Los campos de la ambigüedad son:

- `p` y `q` son referencias a los monomios sobre los cuales es la ambigüedad. El motivo por el cual se guardan en la estructura es para poder implementar una de las optimizaciones.
- `type` indica si la ambigüedad es de inclusión o de superposición (recordar que todas las ambigüedades que se usan en el algoritmo de Buchberger son relevantes).
- `pos` es la posición de `p` donde empieza el pedazo que es en común con `q` y por la cual existe la ambigüedad.
- `a` y `b` son los monomios que hacen que en el caso de las de inclusión `p` sea igual a `aqb` y en el caso de las de superposición `ap` sea igual a `qb`.

En el archivo también está la función `ambiguities` que toma dos monomios y devuelve un vector de `Amb` con todas las ambigüedades entre esos dos monomios. En esta función, al igual que en la divisibilidad de monomios, se usa la función Z para evitar tener que hacer algo cuadrático en los largos de los monomios.

Después, en el archivo está la función `S_poly` que toma una ambigüedad y dos polinomios que deberían ser los polinomios correspondientes y devuelve el S-polinomio correspondiente.

Y por último está la función `checkDeletionCriteria` que implementa la optimización que permite descartar algunas ambigüedades sin tener que reducirlas.

4.5 Buchberger

Como ya se dijo en la Sección 3.4, por el inconveniente de que el algoritmo puede no terminar, se usa una estructura con un método `next`. En esa estructura ya están implementadas las optimizaciones antes mencionadas, pero en esta sección primero se explica cómo sería la estructura sin esas optimizaciones y después en la sección Sección 4.7 se explica algo de cómo se agregan.

La base de esa estructura es así:

```
template<typename K, typename X, class ord>
struct BuchbergerIncremental {
  std::vector<Poly<K, X, ord>> G;
  std::queue<std::tuple<Amb<X>, size_t, size_t>> ambs;
  size_t t = 0;
  // … Métodos
};
```

Los campos de esta estructura guardan lo siguiente:

- `G` es la base de Gröbner que se está construyendo. Al principio se inicializa con los polinomios con los que se llama al constructor.
- `ambs` son las ambigüedades que todavía no se procesaron junto con los índices en `G` de los polinomios a los que corresponde la ambigüedad. Se usa una cola para procesar siempre la que hace más tiempo está esperando.
- La variable `t` está porque como la base de Gröbner incluye a los polinomios originales, las primeras llamadas a `next` tienen que devolver esos polinomios y `t` indica cuántos de esos ya se devolvieron. Cuando `t` es igual al tamaño de `G` es porque ya se devolvieron todos esos.

La estructura además de tener los métodos `next` y `fullBase` tiene varios métodos que se usan internamente. Entre todos los métodos son los siguientes:

- `add_amb`: Este método toma una ambigüedad y dos índices, que son los índices en `G` de los polinomios a los que corresponde la ambigüedad, y los agrega a la cola de ambigüedades. Siempre que hay que agregar una ambigüedad se hace con este método. El motivo por el cual esto está en un método propio es para después, al implementar la optimización de descartar ambigüedades, poder hacerlo solo en este método.
- `add_poly`: Este método agrega un polinomio a `G` agregando además todas las nuevas ambigüedades que aparecen (llamando a `add_amb`). Cada vez que hay que agregar un nuevo polinomio se usa este método, tanto al comienzo cuando se agregan los polinomios iniciales como cuando se agrega un S-polinomio reducido.
- `next`: Este es el método que corre el algoritmo. Lo principal que hace (después de usar la variable `t` para ver si tiene que devolver un elemento de `G`) es sacar ambigüedades de la cola hasta encontrar una que no se reduzca a 0 y cuando la encuentra la agrega a la base (con `add_poly`) y la devuelve. Si se acaban las ambigüedades devuelve vacío (recordar que devuelve `std::optional<Poly<K, ord>>`).
- `fullBase`: Simplemente llama a `next` en un ciclo hasta que devuelva vacío y después devuelve la base.

Lo de las optimizaciones se explica más adelante en la Sección 4.7.

4.6 F4

F4 es un poco más complejo que Buchberger y tiene más partes, así que esta sección está dividida en varias subsecciones para las distintas partes.

4.6.1 Reducción por filas de matrices

Para F4 hace falta hacer la reducción por filas de una matriz. Para esto, en el archivo `ncgb/matrix.cpp` se definió un tipo matriz como un vector de vectores y una función `rref` que hace una eliminación gaussiana directamente y que funciona para cualquier `K`. Sin embargo, la reducción por filas de una matriz es un tema ya muy analizado y hay algunas librerías que lo hacen para distintos tipos `K` mucho más rápido.

Una librería que hace la reducción por filas es FLINT, que lo hace para el tipo `mpq_class` de GMP que ya dijimos que son los números racionales [FLINT; GMP]. Para usarla, en el archivo `ncgb/matrix_mpq_class.hpp` hay una especialización de `rref` para `mpq_class` (en C++ una especialización es cuando hay código definido con un template y se hace una definición aparte para alguna combinación particular de parámetros del template).

Algo malo que tiene esa especialización es que las matrices de racionales con las que trabaja FLINT no están definidas como vectores de vectores, sino que son su propio tipo `fmpq_mat_t`, así que la función tiene que primero copiar la matriz a una matriz de FLINT, después hacer la reducción por filas y por último copiar el resultado de vuelta a la matriz original.

El motivo por el cual se hizo eso es que esas matrices de FLINT se usan de una forma muy particular y complicada. Por ejemplo, para asignarle un valor `x` de tipo `mpq_class` a una posición de la matriz hay que hacer `fmpq_set_mpq(fmpq_mat_entry(mat, i, j), x.get_mpq_t( ));`. Esto hace que si se quisiera directamente en F4 trabajar con las matrices de FLINT no se podría hacer el código genérico para cualquier `K`. Se probó también hacer que las matrices sean una estructura propia y hacer una especialización de `Matrix<mpq_class>` para que use `fmpq_mat_t` por dentro, pero andaba más lento, así que se descartó.

4.6.2 Preprocesamiento simbólico

El preprocesamiento simbólico (Algoritmo 3) está implementado en el propio archivo `ncgb/F4.hpp` en la función `symbolicPreprocessing`. La implementación es directa, así que no hay mucho para comentar.

4.6.3 Reducción

Como en F4 se reducen varios polinomios al mismo tiempo, se implementó una función `multiReduction` que toma dos vectores de polinomios, la base actual y los polinomios a reducir y los reduce todos a la vez.

En el Algoritmo 5 esto se hizo dentro del propio algoritmo de F4 pero en la implementación se hizo aparte porque es bastante lo que tiene que hacer el código.

La implementación lo que hace es construir la matriz con una función llamada `toMatrix`, después llamar a la función `rref` que ya se explicó, después marcar qué columnas corresponden a monomios principales del vector por el cual se está reduciendo (esto para poder saber cuáles polinomios son los reducidos) y por último convertir a polinomios las filas que su coeficiente principal (el primero no nulo de izquierda a derecha) no está marcada como que es un monomio principal del vector por el cual se está reduciendo.

4.6.4 El propio F4

La primera diferencia tiene que ver con la selección de ambigüedades. En Buchberger solo había que elegir una ambigüedad, pero acá hay que elegir varias. Podría haber muchas formas de elegir, pero, como ya se dijo antes, una estrategia que funciona bien es elegir siempre todas

las de menor grado. Para eso se hizo que las ambigüedades estén guardadas por grado en un vector, así:

```
std::vector<std::vector<std::tuple<Amb<X>, size_t, size_t>>> ambs_per_deg;
```

El otro lugar en el que hay diferencia, y mucha, es en `next`. En F4 este método lo que hace es, agarrar todas las ambigüedades de menor grado, poner todos los S-polinomios correspondientes en un vector, hacer la reducción de esos polinomios con la función `multiReduction` y, si hay alguno que queda no nulo, agregar los polinomios reducidos a la base, agregando también las nuevas ambigüedades que aparecen y devolver esos nuevos polinomios, y si quedan todos nulos pasar a las ambigüedades de grado siguiente. Si no quedan más grados de ambigüedades se devuelve vacío.

4.7 Optimizaciones

Como ya se dijo varias veces, hay algunas optimizaciones que se pueden hacer en los algoritmos de Buchberger y F4, que están explicadas en la Sección 4.5 de [Hof20]. Acá solo se explican superficialmente las optimizaciones y se explican cómo están implementadas.

4.7.1 Descartar ambigüedades

La primera optimización consiste en que se pueden descartar algunas ambigüedades, sin tener que reducir su S-polinomio, según si se cumplen ciertas propiedades con otros polinomios que ya están en la base.

En particular algunos teoremas de [Hof20] permiten definir una función con la siguiente signatura:

```
template<typename K, typename X, class ord>
bool
checkDeletionCriteria(
  std::vector<Poly<K, X, ord>>& G, Amb<X>& amb, size_t i, size_t j);
```

Que toma la base, la ambigüedad y a que polinomios corresponde y devuelve `true` si y solo si esa ambigüedad se puede descartar sin reducirla.

Usando esa función, en el método `add_amb` tanto de Buchberger como de F4, antes de agregar la ambigüedad, se agregó una llamada a `checkDeletionCriteria` para solo agregar la ambigüedad si da `false`.

4.7.2 Eliminación de polinomios

La segunda optimización consiste en que cuando se procesa una ambigüedad de inclusión se puede borrar de la base al polinomio grande de la ambigüedad.

Para recordar, una ambigüedad de inclusión entre los polinomio f y g es cuando se tiene que $\mathrm{lm}_{\leq}(f) = a\,\mathrm{lm}_{\leq}(g)b$ con $a, b \in \langle X \rangle$. Esta optimización permite en esas ambigüedades borrar f de la base.

Esta optimización se implementó en Buchberger pero no en F4, porque por algún motivo cuando se probó en F4 anduvo más lento. No se investigó por qué pasó eso.

Para implementar esta optimización en Buchberger lo que se hizo no es directamente borrar el polinomio de `G`, porque para eso habría que también cambiar los índices que están guardados junto con las ambigüedades, sino que se agregó un vector de booleanos `removed` a la estructura, que tiene siempre el mismo largo que `G` y vale `true` solo en las posiciones de polinomios que se borraron. O sea que más bien se está marcando como borrados los polinomios.

A las funciones que se habían explicado, que toman a `G` como parámetro y se usan en Buchberger, se les agregó un parámetro extra que es el vector de booleanos y se hizo que solo trabajen con los polinomios de posiciones que no están en `true`.

4.7.3 Reducción más eficiente en F4

Como ya se dijo antes, para F4 también hay otra optimización que consiste en hacer la reducción por filas de la matriz de forma más eficiente aprovechando la estructura particular que tienen las matrices de F4. Esta reducción se llama reducción Faugère-Lachartre y está explicada en [Hof20].

Esta optimización no se implementó principalmente por el poco soporte para trabajar con matrices de un cuerpo arbitrario, o de una implementación específica de $\mathbb{Q}$, que hay en C++. Sin embargo, queda como trabajo futuro en la Sección 7.2.

4.8 Representación con cofactores

Como se dijo en el capítulo anterior, la representación con cofactores se implementó solo para Buchberger, y no para F4. El motivo es que, como para F4 de cualquier manera no está implementada la optimización de hacer la reducción más eficiente, se hay que seguir trabajando en la parte de la reducción y cuando se tenga lista la versión más eficiente es el momento de implementar la representación con cofactores.

Como se explicó en la Sección 3.5, para la representación con cofactores se usa el tipo `CofactorPoly` que guarda polinomios de un ideal como combinación lineal de elementos de la base.

La estructura de la implementación de este tipo es así:

```
template<typename K, typename X = __uint8_t, class ord = DegLexOrd<X>>
struct CofactorPoly {
  std::vector<std::tuple<Monomial<X>, size_t, Monomial<X>, K>> terms;
  // … Métodos
};
```

Y tiene definidas las operaciones de suma, resta y producto entre polinomios, al igual que con los polinomios normales. Además, tiene un método `add` para agregar una tupla monomio, índice, monomio, coeficiente al polinomio.

A diferencia de los polinomios comunes, acá `terms` no se mantiene ordenado de ninguna manera.

Con `CofactorPoly` se hizo la estructura `BuchbergerIncrementalCofactor`, que es muy parecida a `BuchbergerIncremental` pero con algunos agregados, y se hicieron versiones `Cofactor` de las funciones que se usan.

Para las reducciones se implementó una función que toma, por referencia, un polinomio f, reduce in place a su forma normal f^* y devuelve la diferencia $f - f^*$ como un `CofactorPoly`. La signatura es:

```
template<typename K, typename X, class ord>
CofactorPoly<K, X, ord>
reduceCofactor(
  Poly<K, X, ord>& f, const std::vector<Poly<K, X, ord>>& G,
  const std::vector<CofactorPoly<K, X, ord>>& g_rec);
```

También se hizo una versión de la función de los S-polinomios, que toma la ambigüedad y los polinomios correspondientes f y g y devuelve, además del S-polinomio p, los monomios a, b, a' y b' y los elementos del cuerpo c y c', que hacen que $p = cafb + c'a'fb'$.

```
template<typename K, typename X, class ord>
std::tuple<
  Poly<K, X, ord>,
  std::tuple<Monomial<X>, Monomial<X>, K>,
  std::tuple<Monomial<X>, Monomial<X>, K>>
S_polyCofactor(
  const Amb<X>& amb, const Poly<K, X, ord>& f, const Poly<K, X, ord>& g);
```

Con esto, en la estructura `BuchbergerIncrementalCofactor` se agregó (con respecto a `BuchbergerIncremental`) un campo `std::vector<CofactorPoly<K, ord>> G_rec` que guarda la representación con cofactores de cada polinomio de `G`.

En este campo `G_rec`, a los elementos iniciales simplemente se los agrega como `Monomial(), i, Monomial(), K(1)` y a los elementos que vienen de un S-polinomio se les construye el `CofactorPoly` con los `CofactorPoly` de los polinomios que forman la ambigüedad multiplicados por los monomios y coeficientes que devuelve `S_polyCofactor` y con el `CofactorPoly` de la reducción que devuelve la función `reduceCofactor`.

4.9 Comparación de bases de Gröbner

La función `cmpBases` que se mencionó en la Sección 3.6 que toma dos conjuntos generadores y, asumiendo que son bases de Gröbner, dice si generan el mismo ideal o no está implementada usando tal cual el Colorario 2.45.

4.10 Paralelismo

Cuando se paraleliza siempre es importante primero tratar de optimizar la versión no paralela lo más posible. En este caso lo mejor no paralelo es el algoritmo F4, pero como no está implementada la optimización de la reducción eficiente, no está todavía optimizado al máximo. De cualquier manera, como la reducción de matrices es solo una parte del algoritmo, se puede pensar cómo se podría paralelizar el resto.

El método `add_poly` que ya se explicó que agrega un polinomio a la base y las nuevas ambigüedades correspondientes, sin paralelizar se implementó así:

```
void add_poly(const Poly<K, X, ord>& f) {
  G.push_back(f);
  for (size_t k = 0; k < G.size() - 1; k++) {
    for (auto& amb : ambiguities(G[k].lm(), f.lm())) {
      add_amb(amb, k, G.size() - 1);
    }
    for (auto& amb : ambiguities(f.lm(), G[k].lm())) {
      add_amb(amb, G.size() - 1, k);
    }
  }
}
```

Y el método `add_amb` se implementó así:

```
void add_amb(Amb<X>& amb, size_t i, size_t j) {
  if (checkDeletionCriteria(G, amb, i, j)) {
    return;
  }
  ambs.push({amb, i, j});
}
```

El primer `for` de `add_poly` así como está casi que se podría paralelizar, corriendo sus distintas iteraciones en paralelo. Lo único que lo impide es que `add_amb` hace un `ambs.push` que sería problemático si se llega a ejecutar al mismo tiempo por más de un hilo. Para solucionar eso lo que se hizo es guardar las ambigüedades en varios vectores, uno por cada polinomio de `G` y después al final de la función agregar todas las ambigüedades a `ambs`. Haciendo eso, el código paralelizado quedó así:

```
void add_poly(Poly<K, X, ord>& f) {
  G.push_back(f);
  size_t lim = G.size() - 1;
  std::vector<std::vector<std::tuple<Amb<X>, size_t, size_t>>> to_add(lim);

  #pragma omp parallel for
  for (size_t k = 0; k < lim; k++) {
    for (auto& amb : ambiguities(G[k].lm(), G.back().lm())) {
      if (!checkDeletionCriteria(G, amb, k, lim)) {
        to_add[k].push_back({amb, k, lim});
      }
    }
    for (auto& amb : ambiguities(G.back().lm(), G[k].lm())) {
      if (!checkDeletionCriteria(G, amb, lim, k)) {
        to_add[k].push_back({amb, lim, k});
      }
    }
  }

  for (size_t k = 0; k < lim; k++) {
    for (auto& [amb, i, j] : to_add[k]) {
      size_t d = amb.size();
      while (ambs_per_deg.size() <= d) {
        ambs_per_deg.push_back({});
      }
      ambs_per_deg[d].push_back({amb, i, j});
    }
  }
}
```

El `#pragma omp parallel for` es lo que efectivamente hace que OpenMP paralelice el `for`. En el Capítulo 6 se presentan resultados sobre qué tal anda la paralelización.

Capítulo 5
Tests

Los tests suelen ser una parte fundamental en la implementación de una librería robusta y confiable, así que en este trabajo se testeó todo lo más posible. A medida que se fue escribiendo el código se fueron haciendo tests del código para eliminar los bugs. Además, con algunos de los tests se hizo que realicen mediciones de tiempo para ver qué tan rápido andaba el código. Esto último se llama benchmarking. En esta sección se explican los tests y en el siguiente capítulo los benchmarks que se hicieron.

Todos los tests están en el directorio `test`, el cual está dividido en varios subdirectorios con varios tests cada uno. Para correr todos los tests se puede usar el comando `make test`. En el directorio `.github/workflows` está configurado para que al subir los cambios al repositorio se corran los tests automáticamente, aunque eso no siempre funciona porque cuando se corren los tests en GitHub se tiene que instalar SageMath y a veces falla la instalación.

Algunos tests corren el código de la librería y el código de `operator_gb`, que es la librería hecha por Clemens Hofstadler en su tesis de máster [Hof20] usando SageMath, y comparan los resultados, y otros corren solo código de la librería.

También, algunos tests corren los mains de ejemplo del directorio `mains` y otros directamente incluyen a los archivos de la librería y los testean directamente.

A continuación una lista de lo más importante de los tests:

- Hay tests de las operaciones básicas de monomios, polinomios, reducción, etc. en el directorio `test/internal_tests`.
- Para algunos conjuntos generadores que se sabe que sus ideales generados tienen bases de Gröbner finitas hay un test que calcula una base de Gröbner con Buchberger, una con F4 y una con `operator_gb` y chequea que los resultados sean equivalentes usando `mains/compare_bases.cpp`. Este archivo lo que hace es leer dos conjuntos generadores y, asumiendo que son bases de Gröbner, dice si generan el mismo ideal o no usando la función `cmpBases` explicada en la Sección 3.6. Este test está en el directorio `test/base_tests`.
- Para algunos conjuntos generadores construidos al azar se construye otro polinomio también al azar y tanto con Buchberger como con F4 y con `operator_gb` se ejecutan algunos pasos del cálculo de la base de Gröbner para ver si está el polinomio en el ideal o no. Si alguno(s) de los tres da que sí está en el ideal y otro(s) no, se ejecutan los que no con muchísimos más pasos, para que tenga que terminar sí o sí porque por los que dieron que sí sabemos que (si no hay bugs) sí está. Esto está en el directorio `test/InIdeal_tests`.
- Con respecto a la representación con cofactores, hay un test que corre los algoritmos con y sin representación con cofactores y se fija que el resultado de con cofactores después de convertir los `CofactorPoly` en `Poly` sea igual al resultado sin cofactores. Este test está en el directorio `test/cofactor_tests`.
- Por último hay un test que corre F4 no paralelo y paralelo con distintas cantidades de hilos y se fija que siempre dé el mismo resultado. Esto está en el directorio `test/parallelism_tests`.

Capítulo 6
Benchmarks

En este capítulo se presentan los resultados obtenidos tras correr los tests que incluyen benchmarks. Las corridas se hicieron en la computadora 'atom' de FAMAF que tiene las siguientes características:

- Procesador: AMD EPYC 7643 48-Core Processor
- RAM: 126 GB
- Sistema Operativo: Debian GNU/Linux trixie/sid
- Versión de `g++`: 14.2.0-12

De cualquier manera, esos datos no son muy importantes porque lo importante son las relaciones entre los distintos tiempos.

Los tests que tienen benchmarks son, todos salvo uno, tests que usan los archivos del directorio `mains` y archivos ejecutables hechos en Python con `operator_gb` y son corridos con distintas entradas desde un archivo de Python. Los tiempos en todos salvo uno se miden desde el programa de Python, así que incluyen el tiempo hasta que se inicializa el programa y lee la entrada y el tiempo de impresión de la salida, pero como son siempre entradas y salidas relativamente chicas, eso no debería hacer casi diferencia. En el test en el que los tiempos no se miden desde Python, se miden desde el propio C++.

Todos los tests que tienen benchmarks, salvo uno (que más adelante se explica), funcionan tomando como input un conjunto generador que se sabe que el ideal generado tiene una base de Gröbner finita. Para estos benchmarks se usaron los siguientes conjuntos generadores:

- FK2 = $\{a^2\}$
- FK3 = $\{a^2,\ b^2,\ c^2,\ ac+ba+cb,\ ab+bc+ca\}$
- FK4 = $\{a^2,\ b^2,\ c^2,\ d^2,\ e^2,\ f^2,\ ac+ba+cb,\ ae+da+ed,\ bf+db+fd,\ cf+ec+fe,\ ab+bc+ca,\ ad+de+ea,\ bd+df+fb,\ ce+ef+fc,\ cd+dc,\ be+eb,\ af+fa\}$
- tri1 = $\{a^2-1,\ b^3-1,\ (ababab^2ab^2)^2-1\}$
- tri2 = $\{a^2-1,\ b^3-1,\ (ababab^2)^3-1\}$
- tri3 = $\{a^3-1,\ b^3-1,\ (abab^2)^2-1\}$
- tri4 = $\{a^3-1,\ b^3-1,\ (abaab^2)^2-1\}$
- tri5 = $\{a^2-1,\ b^5-1,\ (abab^2)^2-1\}$
- tri6 = $\{a^2-1,\ b^5-1,\ (ababab^4)^2-1\}$
- tri7 = $\{a^2-1,\ b^5-1,\ (abab^2ab^4)^2-1\}$
- tri8 = $\{a^2-1,\ b^2-1,\ (ababab^3)^2-1\}$
- tri9 = $\{a^2-1,\ b^3-1,\ (abab^2)^2-1\}$
- tri10 = $\{a^2-1,\ b^3-1,\ (ababab^2)^2-1\}$

- tri11 = $\{a^2-1,\ b^3-1,\ (ababab^2)^2-1\}$
- tri12 = $\{a^2-1,\ b^3-1,\ (ababab^2abab^2)^2-1\}$
- tri13 = $\{a^2-1,\ b^3-1,\ (bababababab^2ab^2)^2-1\}$
- trit3 = $\{a^3-1,\ b^3-1,\ c^3-1,\ (ab)^2-1,\ (ac)^2-1,\ (bc)^2-1\}$
- trit4 = $\{a^3-1,\ b^3-1,\ c^4-1,\ (ab)^2-1,\ (ac)^2-1,\ (bc)^2-1\}$
- trit5 = $\{a^3-1,\ b^3-1,\ c^5-1,\ (ab)^2-1,\ (ac)^2-1,\ (bc)^2-1\}$

Los FK provienen de la llamada teoría de álgebras de Nichols [And17], la cual es de interés para Cristian Vay, el director de este trabajo, y otros investigadores de FAMAF. Los FK definen una álgebra llamada Fomin-Kirillov y son una familia infinita parametrizada por un $n \in \mathbb{N}$, que fijando el n se describe a continuación.

- El cuerpo es $\mathbb{Q}$.
- El alfabeto es $\{x_{ij} : i,j \in \{1,...,n\} : i \neq j\}$, donde los $_{ij}$ son un par no ordenado.
- El conjunto generador es:

$$\begin{aligned}&\{x_{ij}^2 : i,j \in \{1,...,n\} : i \neq j\}\\ &\cup \{x_{ij}x_{ik} + x_{ik}x_{jk} + x_{jk}x_{ij} : i,j,k \in \{1,...,n\} : i \neq j \neq k \neq i\}\\ &\cup \{x_{ik}x_{ij} + x_{jk}x_{ik} + x_{ij}x_{jk} : i,j,k \in \{1,...,n\} : i \neq j \neq k \neq i\}\\ &\cup \{x_{ij}x_{kl} + x_{kl}x_{ij} : i,j,k,l \in \{1,...,n\} : i \neq j \wedge k \neq l\}.\end{aligned}$$

Se sabe que para $n \leq 5$ los ideales generados por FKn tienen bases de Gröbner finitas, pero para $n > 5$ no se sabe. En [GV] y [And17] se puede encontrar más información sobre las álgebras de Nichols.

Los tri están porque [Hof20] los usa, con ese mismo nombre, y él los sacó del Teorema 2.12 de [RS02]. Los trit son directamente tomados de la Proposición 1.9 de [RS02].

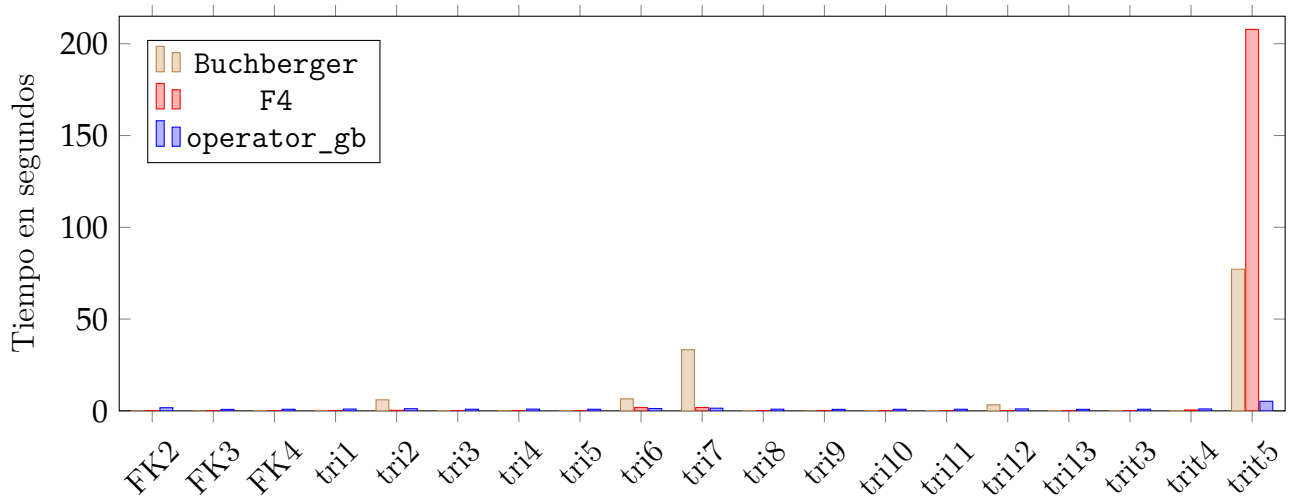
En todos los benchmarks que usan estos conjuntos se trabaja sobre los racionales, porque es para los racionales que se sabe que tienen bases finitas.

En las siguientes secciones se presentan los resultados de cada uno de los benchmarks.

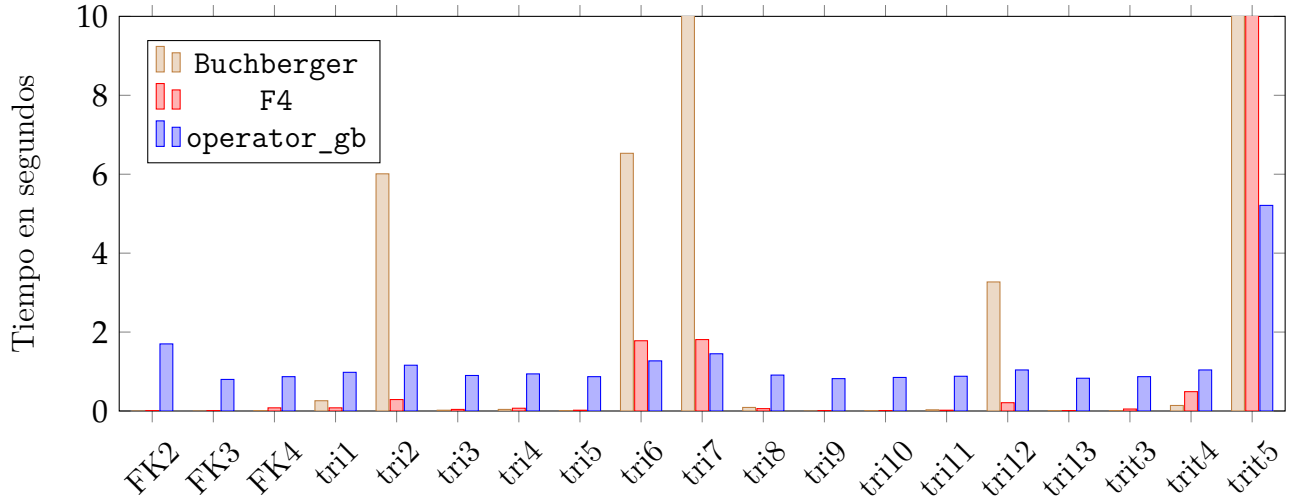
6.1 Ideales con bases de Gröbner finitas

El test que usa los distintos algoritmos para calcular bases de Gröbner de ideales que se sabe que tienen bases de Gröbner finitas incluye mediciones de los tiempos y usa los casos que vimos recién.

En el siguiente gráfico se muestran los tiempos de ejecución de Buchberger de este trabajo (`Buchberger`), de F4 de este trabajo (`F4`) y de `operator_gb` (que usa F4) para esos casos.



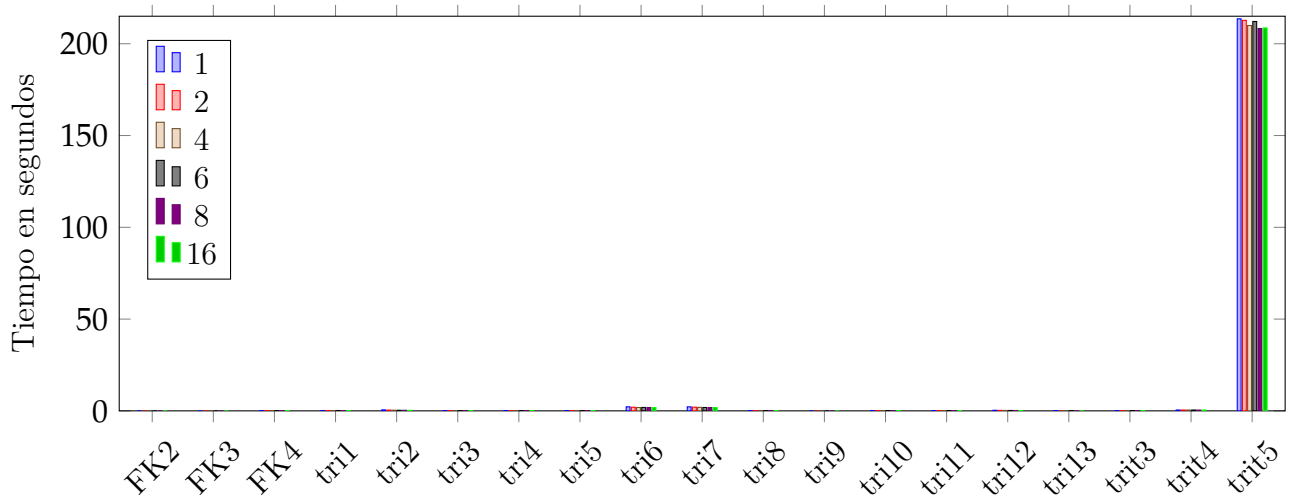
A simple vista se puede ver que para los casos en los que se distingue algo `operator_gb` es mucho más rápido, pero solo se ven los casos que demoran mucho tiempo, los otros no. Así que a continuación está el gráfico de vuelta pero solo con la parte de más abajo y dejando que las barras muy largas se salgan del gráfico.



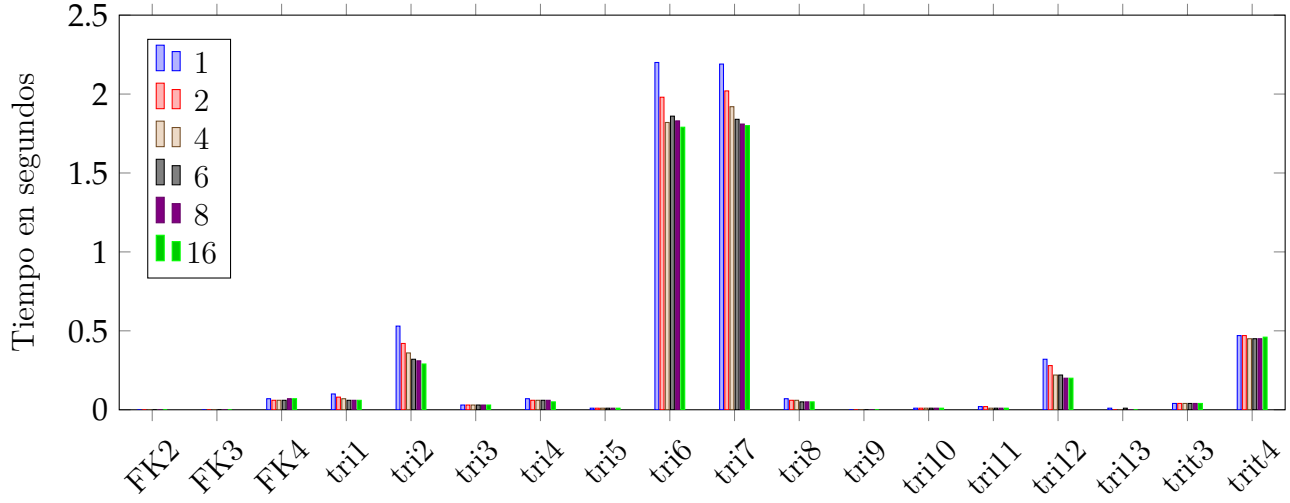
Acá se puede ver que en los casos más chiquitos sí pasa que `Buchberger` y `F4` le ganan a `operator_gb`. Que en los grandes gane `operator_gb` tiene sentido por el hecho de que tiene implementada la optimización de reducir las matrices mas eficientemente. Que en los chicos `Buchberger` y `F4` sean más rápido también tiene sentido por el hecho de que C++ es mucho más rápido que Python.

6.2 Paralelismo

El test que verifica que de el mismo resultado correr con distintas cantidades de hilos también usa los mismos casos y mide los tiempos. En el siguiente gráfico se pueden ver los resultados.



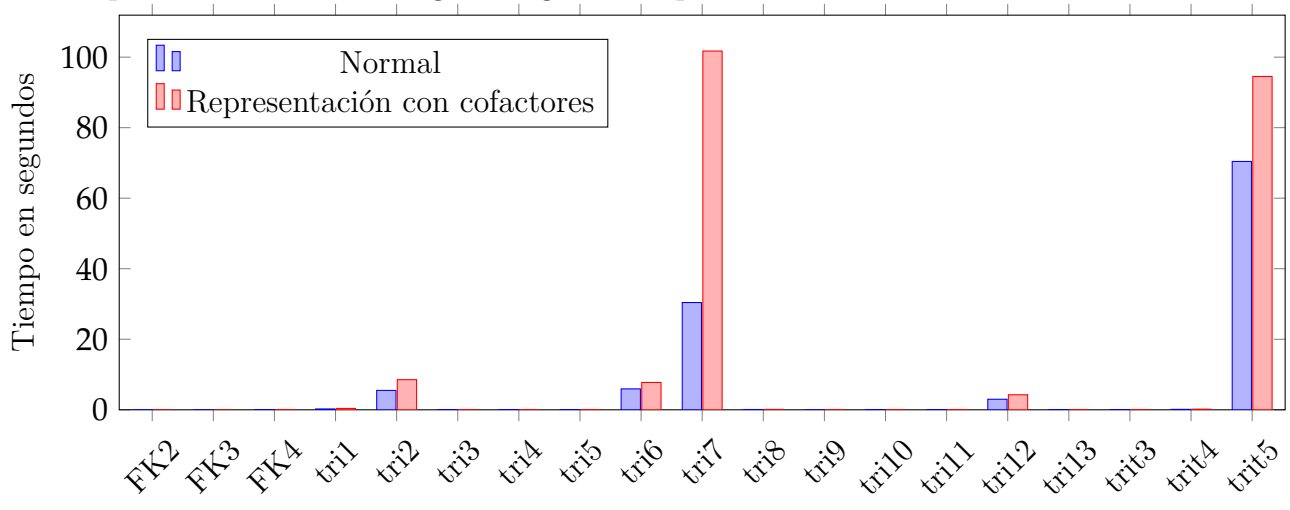
De vuelta a simple vista no se ve casi nada, salvo que en trit5 es casi lo mismo con más hilos. Para poder ver en más detalle, a continuación un gráfico de lo mismo pero sin trit5.



Se puede ver que aumentar los hilos hace que sea un poco más rápido, pero muy poco. Esto tiene sentido por el hecho de que solo la parte de agregar ambigüedades está paralelizada.

6.3 Representación con cofactores

El test que compara ejecutar Buchberger con y sin representación con cofactores también usa los mismos casos y mide los tiempos. Este es el único benchmark que no funciona con mains del directorio `mains` sino que usa sus propios archivos, y además este benchmark mide los tiempos desde C++. En el siguiente gráfico se pueden ver los resultados.



Se puede ver que la representación con cofactores lleva más tiempo, pero en general no tanto más. Acá no hay demasiado más para ver en la parte de abajo, así que no se hace un gráfico

con solo la parte de más abajo.

6.4 Pertenencia a ideales

El test que para conjuntos generadores construidos al azar y polinomios construidos al azar ve con los distintos algoritmos si el polinomio está en el ideal generado o no, también mide los tiempos. Estos corren tanto para los racionales como para la aritmética modular, pero para los racionales incluyen la comparación con `operator_gb` y para la aritmética modular no, porque `operator_gb` es solo para racionales. Este benchmark imprime los tiempos promedios sobre todos los conjuntos generadores construidos al azar y polinomios construidos al azar y también los tiempos máximos. Estos son los resultados para los racionales (tal cual los imprime el test, `Buch` es `Buchberger` y `GB` es `operator_gb`):

```
Average times:
Buch: 0.0041429424s
F4: 0.0105502963s
GB: 0.8995876765s
Max times:
Buch: 0.1320753098s
F4: 0.0925185680s
GB: 4.6477575302s
```

Y estos para la aritmética modular (también tal cual los imprime el test):

```
Average times:
Buch_Zp: 0.0028817248s
F4_Zp: 0.0072190428s
Max times:
Buch_Zp: 0.0779249668s
F4_Zp: 0.0799708366s
```

Estos datos igualmente mucho no dicen, porque como buscan por una cierta cantidad de pasos, y los pasos son distintos en los distintos algoritmos, no se puede comparar. En particular, un paso de `F4` puede corresponder a muchos pasos de `Buchberger`.

Este benchmark se usó principalmente para tenerlo como comparación cuando se hacían cambios en el código. Como se corrían siempre todos los tests y benchmarks antes y después del cambio, este servía para comparar el antes y el después.

Capítulo 7
Conclusiones

El tema de este trabajo fue propuesto por el director Cristian Vay por su interés ya mencionado en usarlo para las álgebras de Nichols. Para mí este era un tema completamente nuevo, así que empecé por estudiar el tema. Primero leyendo [CLO15] que trata sobre bases de Gröbner conmutativas. Mientras leía ese libro hice también una implementación propia, sin mucha prolijidad, de bases de Gröbner conmutativas en C++. Esto fue bueno porque me sirvió de práctica para después hacer la implementación de bases de Gröbner no conmutativas. Después de eso fui leyendo la tesis de master [Hof20] y la tesis de doctorado [Hof23] de Clemens Hofstadler, que son textos sobre bases de Gröbner no conmutativas, e implementando lo explicado en C++.

A continuación menciono algunos aspectos positivas y después algunos posibles trabajos futuros.

7.1 Aspectos positivos de este trabajo

La implementación de bases de Gröbner no conmutativas presentada en esta tesis es la primera en C++ (según mi conocimiento). Esto es muy bueno porque C++ es un lenguaje particularmente rápido y en el que es fácil paralelizar. Esto probablemente sea útil si se quieren correr casos que se demoran mucho.

También, es la primer implementación de F4 que funciona para un cuerpo arbitrario (según mi conocimiento). Esto está bueno para que si en algún momento alguien lo quiere usar para algún cuerpo distinto lo pueda hacer.

La implementación del algoritmo Buchberger incluye la parte de la representación con cofactores, que si bien es algo muy fácil, la única otra implementación que lo hace es `operator_gb` (según mi conocimiento).

7.2 Trabajos futuros

Como ya se dijo varias veces, falta usar la reducción por filas de Faugère-Lachartre que es más eficiente, y ese es uno de los trabajos futuros más importantes para hacer.

Del algoritmo F4 (y de Buchberger) para el caso conmutativo hay varias implementaciones en C++, por ejemplo [OPENF4; RS12; MS17], inclusive en paralelo, como [Ree98; Neu12]. Muchas son solo para los enteros módulo un primo, pero algunas también para los racionales. Es muy probable que analizando esas implementaciones se puedan sacar muchas ideas o implementaciones útiles, en particular, podría ser que ya alguien haya implementado la reducción por filas de Faugère-Lachartre en C++.

Cuando se fue haciendo la librería, muchas veces pasaba que había cambios que parecían optimizaciones de implementación pero al hacerlos el código resultaba andar más lento (a veces por muy poquito), así que se descartaban esos cambios. Sería bueno agregar más tests y probar hacer cambios para ver si mejoran o no los tiempos teniendo más tests, porque podría ser que fueran un poquito más lento justo en los casos de los tests pero que en general sí fueran más rápidas.

Se podría analizar cuánto tiempo se consume en las distintas partes de los algoritmos, porque eso podría ayudar a saber en qué partes se pueden realizar optimizaciones.

En la librería hay una especialización de `rref` para hacer que `rref` para `mpq_class` sea más rápido usando la implementación de la librería. Para aritmética modular se está usando un tipo propio definido en el archivo `extras/ModularArithmetic.hpp` y se usa la versión genérica de `rref`. Es posible que existan librerías que tengan tipos de aritmética modular más rápidos

que el que se implementó y con una reducción por filas más rápida, y en ese caso sería bueno usarlas.

Actualmente la librería no viene con nada para instalarla automáticamente porque en C++ eso no es fácil de hacer. Esto significa que si alguien quiere usar la librería tiene que arreglárselas y probablemente hacer algo medio feo como poner los archivos en alguna carpeta particular. Otras librerías de C++ sí vienen con algo para eso, así que sería bueno hacerlo para esta librería también.

Los ideales de la Definición 2.21 no son los únicos; también existen los ideales a izquierda que se definen de la siguiente manera.

Definición 7.1. *Sean R un anillo e $I \subseteq R$. Se define que I es un ideal a izquierda de R si y solo si:*

(1) $I \neq \emptyset$.

(2) $\forall\, a, b \in I : a + b \in I$.

(3) $\forall\, a \in I, r \in R : ra \in I$.

Con los ideales a izquierda sobre polinomios no conmutativos también hay toda una teoría de bases de Gröbner. En [Hof23] se puede encontrar una explicación sobre esta teoría. Implementar el cálculo de bases de Gröbner de la teoría de ideales a izquierda en C++ es otro trabajo que se podría hacer. Análogamente, también se pueden considerar ideales a derecha.

Como ya se mencionó en el Capítulo 6, no se sabe para $n \geq 6$ si los FKn tienen una base finita o no. Cuando se tenga una versión muy buena de F4, se la podría correr por mucho tiempo para FK6 para ver si llega en algún momento a una base finita.

Actualmente para estudiar el tema de bases de Gröbner, tanto conmutativas como no conmutativas, hay libros y artículos sobre el tema, en particular [CLO15] para conmutativas y [Hof20] para no conmutativas, pero para practicar las implementaciones no hay mucha ayuda. Por ejemplo, cuando implementé las bases de Gröbner conmutativas para practicar tuve que hacer mi propio testing. Para que sea más fácil estudiar el tema sería útil preparar problemas del estilo de programación competitiva sobre el tema. Estos problemas se podrían poner en, por ejemplo, Library Checker, o en un grupo de Codeforces.

Por último, otra cosa que se podría hacer es implementar el cálculo de bases de Gröbner no conmutativas en otros lenguajes, como por ejemplo Rust, que, además de ser un lenguaje muy rápido como C++, maneja la memoria de forma más segura, previniendo los errores de memoria, que en la implementación en C++ podría haber porque se usan referencias en algunos lugares.

Bibliografía

[AK06] Peter Ackermann y Martin Kreuzer. "Gröbner basis cryptosystems". En: **Appl. Algebra Eng. Commun. Comput.** 17.3-4 (2006), páginas 173-194. DOI: 10.1007/s00200-006-0002-0.

[And17] Nicolás Andruskiewitsch. "An introduction to Nichols algebras". En: **Quantization, geometry and noncommutative structures in mathematics and physics. Contributions of the summer school, Villa de Leyva, Columbia**. Cham: Springer, 2017, páginas 135-195. DOI: 10.1007/978-3-319-65427-0_4.

[BN99] Franz Baader y Tobias Nipkow. **Term rewriting and all that**. Cambridge: Cambridge University Press, 1999. DOI: 10.1017/CBO9781139172752.

[CKG24] A. Cohen, J. Knopper y T. GAP Team. **GBNP, computing Gröbner bases of noncommutative polynomials, Version 1.1.0**. https://gap-packages.github.io/gbnp/. GAP package. 2024-08.

[CLO15] David A. Cox, John Little y Donal O'Shea. **Ideals, varieties, and algorithms. An introduction to computational algebraic geometry and commutative algebra**. 4th revised ed. Undergraduate Texts Math. Cham: Springer, 2015. DOI: 10.1007/978-3-319-16721-3. URL: https://epub.jku.at/obvulihs/download/pdf/5013051.

[CPAZf] **Z-function and its calculation**. URL: https://cp-algorithms.com/string/z-function.html.

[Dec+24] Wolfram Decker et al. SINGULAR **4-4-0 — A computer algebra system for polynomial computations**. http://www.singular.uni-kl.de. 2024.

[FLINT] The FLINT team. **FLINT: Fast Library for Number Theory**. Version 3.0.0. 2023. URL: https://flintlib.org.

[GMP] Torbjörn Granlund y the GMP development team. **GNU MP: The GNU Multiple Precision Arithmetic Library**. 2024. URL: http://gmplib.org/.

[GV] Matías Graña y Leandro Vendramin. **Nichols algebras of nonabelian group type**. URL: http://mate.dm.uba.ar/~lvendram/zoo/.

[Hof20] Clemens Hofstadler. "Certifying operator identities and ideal membership of noncommutative polynomials". Johannes Kepler University Linz, 2020-03.

[Hof23] Clemens Hofstadler. "Noncommutative Gröbner bases and automated proofs of operator statements". Johannes Kepler University Linz, 2023-09.

[Mor94] Teo Mora. "An introduction to commutative and noncommutative Gröbner bases". En: **Theoretical Computer Science** 134.1 (1994), páginas 131-173. DOI: https://doi.org/10.1016/0304-3975(94)90283-6. URL: https://www.sciencedirect.com/science/article/pii/0304397594902836.

[MS17] Rusydi H. Makarim y Marc Stevens. "M4GB: An Efficient Gröbner-Basis Algorithm". En: **Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017**. Editado por Michael A. Burr, Chee K. Yap y Mohab Safey El Din. ACM, 2017, páginas 293-300. DOI: 10.1145/3087604.3087638. URL: https://doi.org/10.1145/3087604.3087638.

[NCA] J. William Helton, Mauricio C. de Oliveira, Bob Miller, Mark Stankus. **The NCAlgebra Suite**. URL: https://ncalgebra.github.io.

[Neu12] Severin Neumann. “Parallel Reduction of Matrices in Gröbner Bases Computations”. En: **Computer Algebra in Scientific Computing - 14th International Workshop, CASC 2012, Maribor, Slovenia, September 3-6, 2012. Proceedings**. Editado por Vladimir P. Gerdt et al. Volumen 7442. Lecture Notes in Computer Science. Springer, 2012, páginas 260-270. DOI: 10.1007/978-3-642-32973-9_22. URL: https://doi.org/10.1007/978-3-642-32973-9%5C_22.

[OMP] **OpenMP**. 2024. URL: https://www.openmp.org/.

[OPENF4] Titouan Coladon. **OPENF4 Documentation**. URL: http://nauotit.github.io/openf4.

[Ree98] Alyson Reeves. “A Parallel Implementation of Buchberger’s Algorithm over $\mathbb{Z}_p$ for $p \leq 31991$”. En: **J. Symb. Comput.** 26.2 (1998), páginas 229-244. DOI: 10.1006/JSCO.1998.0208. URL: https://doi.org/10.1006/jsco.1998.0208.

[RS02] Gerhard Rosenberger y Martin Scheer. “Classification of the finite generalized tetrahedron groups”. En: **Combinatorial and geometric group theory. Proceedings of the AMS special session on combinatorial group theory, New York, NY, USA, November 4–5, 2000 and the AMS special session on computational group theory, Hoboken, NJ, USA, April 28–29, 2001**. Providence, RI: American Mathematical Society (AMS), 2002, páginas 207-229.

[RS12] Bjarke Hammersholt Roune y Michael Eugene Stillman. “Practical Groebner Basis Computation”. En: **CoRR** abs/1206.6940 (2012). arXiv: 1206.6940. URL: http://arxiv.org/abs/1206.6940.

[WWP] **Word problem (mathematics)**. URL: https://en.wikipedia.org/wiki/Word_problem_(mathematics).

[Xiu12] Xingqiang Xiu. “Non-commutative Gröbner Bases and Applications”. Tesis doctoral. 2012-07.