Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Raíces Reales de Polinomios Multivariados

Enrique A. Tobis
etobis@dc.uba.ar

Directora
Dra. Alicia M. Dickenstein
Departamento de Matemática
FCEyN—Universidad de Buenos Aires

Co–director
Dr. Min Chih Lin
Departamento de Computación
FCEyN—Universidad de Buenos Aires

Tesis para acceder al grado de
Licenciado en Ciencias de la Computación

30 de diciembre de 2003

Agradecimientos

A mi familia, por el apoyo constante en mi aventura por la ciencia.

A María Angélica, por ser mi mejor y más paciente compañera.

A Daniel, por compartir su sapiencia.

A Alicia, por ser mi guía experta mientras entraba, trabajosamente, al mundo de los polinomios.

Resumen

Uno de los problemas más importantes en la Geometría Algebraica Computacional (GAC) es determinar la cantidad de ceros reales comunes de un conjunto de polinomios multivariados con coeficientes reales. En el presente trabajo se explican distintos algoritmos para realizar ese cálculo cuando los polinomios tienen finitos ceros complejos.

El otro problema abordado es el de la determinación de signos. Esto es, dado un conjunto de polinomios $\mathcal{P}$ y un ideal I de dimensión cero, decir qué condiciones de signos pueden satisfacer los polinomios de $\mathcal{P}$ al ser evaluados en los elementos de la variedad generada por I, lo cual permite localizar los ceros reales de I.

Todos los algoritmos fueron implementados en SINGULAR [GPS/01], un programa libre de amplia difusión en la comunidad de investigación en GAC.

Abstract

One of the main problems in Computational Algebraic Geometry is determining the number of real common roots of a set of multivariate polynomials with real coefficients. In this work, we explain several different algorithms which perform that calculation, assuming the polynomials have a finite number of complex zeros.

The other problem we address is sign determination, namely, given a set $\mathcal{P}$ of polynomials, and a zero-dimensional ideal I, which are the sign conditions realized by evaluating the polynomials in $\mathcal{P}$ at the points of the variety generated by I. This can be used to locate the real zeros of I.

All the algorithms have been implemented in SINGULAR [GPS/01], a free program widely used in the CAG research comunity.

Índice general

Capítulo 1

Introducción

Numerosas situaciones pueden modelarse mediante sistemas de ecuaciones polinomiales, como problemas de movimiento directo en robots seriales o el problema inverso en robots paralelos [Mou/93], problemas en teoría de juegos como la determinación de equilibrios de Nash totalmente mezclados [Dat/03], problemas de control con feedback [RS/98], problemas químicos de estequiometría [Gat/01], etc. En estas situaciones "reales", muchas veces es relevante no sólo determinar si los sistemas tienen solución en el cuerpo algebraicamente cerrado de los números complejos, sino también si existen soluciones *reales*, es decir, vectores de soluciones cuyas coordenadas son números reales.

Asimismo, la determinación de las posibles soluciones reales de polinomios multivariados con coeficientes reales, es un ingrediente esencial en importantes problemas de clasificación de estructuras geométricas, como por ejemplo la determinación de métricas de Einstein en variedades homogéneas compactas [BWZ/03].

Un enfoque para encontrar y contar raíces reales es el numérico. Podemos tratar de aproximar el valor de las raíces reales del polinomio, y así obtener la cantidad buscada. Sin embargo, estos métodos pueden traer problemas. Existen polinomios muy mal condicionados numéricamente para este problema. Un ejemplo clásico, debido a James H. Wilkinson [Wil/94], es el polinomio

$$W(x) = \prod_{i=1}^{20}(x+i)$$

Este polinomio tiene sus veinte raíces reales $(-1, -2, \ldots, -20)$. Consideremos el polinomio

$$\tilde{W}(x) = W(x) + 10^{-9}x^{19}$$

$\tilde{W}(x)$ difiere de $W(x)$ en una variación muy pequeña de uno solo de sus coeficientes. Esto nos podría llevar a pensar que sus raíces están distribuidas

en forma similar a las de $W(x)$. Sin embargo, mientras que $W(x)$ tiene 20 raíces reales, $\tilde{W}(x)$ tiene sólo 12 raíces reales y 4 pares de raíces complejas. De este ejemplo, podemos deducir que el enfoque numérico puro no es necesariamente bueno para atacar el problema. En este trabajo, utilizamos un enfoque estrictamente simbólico. Para el ejemplo del polinomio de Wilkinson, la respuesta del método simbólico implementado es prácticamente instantánea (ver el ejemplo 2.12).

En la sección 2.1.1, estudiamos la Regla de los Signos de Descartes, que permite acotar la cantidad de raíces reales de un polinomio univariado. Esta regla tiene la siguiente importante consecuencia: el número de raíces reales de un polinomio está acotado en términos del número de monomios presentes, y esta cota es ajustada. En el caso multidimensional, Khovanskii [Kho/91] encontró una cota para un sistema de n polinomios reales en n variables. Esta cota depende del número de monomios presentes, pero es enorme, y difiere mucho de las cantidades encontradas empíricamente.

La conjetura de Kushnirenko proponía una cota muy inferior, en particular conjeturaba que que dos trinomios bivariados no podían tener más de cuatro raíces reales no degeneradas en el cuadrante positivo. Esta conjetura fue refutada por Bertrand Hass [Haa/02], quien presentó el par

$$x^{108} + 1{,}1y^{54} - 1{,}1y; y^{108} + 1{,}1x^{54} - 1{,}1x$$

que tiene cinco raíces reales en el primer cuadrante. Recientemente se demostró que 5 es la cantidad máxima para el número de raíces reales con coordenadas positivas de dos trinomios en dos variables, más algunas extensiones [RLW/03], pero una buena cota general no sólo no ha sido demostrada, sino tampoco conjeturada.

El objetivo de este trabajo es el de obtener una implementación eficiente de métodos simbólicos para la determinación del número y la localización de ceros reales de polinomios multivariados, o tan eficiente como sea posible, dadas las limitaciones de la complejidad inherente del problema. Esta implementación tiene como objeto ayudar a la solución de problemas concretos y servir de apoyo a la investigación teórica, con vistas tanto a posibles aplicaciones prácticas como a posibles aplicaciones teóricas, proveyendo una herramienta de experimentación para apoyo de la investigación en GAC. Es por esto que hemos implementado los algoritmos en el programa SINGULAR, desarrollado desde 1984 en la Universidad de Kaiserslautern, Alemania, de excelente performance y de muy buena difusión entre matemáticos que trabajan en aplicaciones de la Geometría Algebraica.

La referencia básica para todos los desarrollos del trabajo es [BPR/03]. Cualquier resultado que no aparezca con una referencia puntual, se podrá en-

contrar en los capítulos 2, 8, 11 y 12 de ese libro. Las explicaciones de la próxima sección se pueden encontrar, con más detalles, en [CLO/98], [CLO/97] y [GP/02].

A continuación, daremos algunas definiciones básicas. Ellas nos permitirán dar un marco teórico al trabajo. Luego detallamos algunas de las características de SINGULAR.

1.1 Sistemas de ecuaciones polinomiales

El problema que en el que nos concentraremos en este trabajo es: dados m polinomios $f_1, \ldots, f_m$ en $\mathbb{R}[X_1, \ldots, X_n]$, y sabiendo que tienen un número finito de raíces comunes en $\mathbb{C}^n$, decir cuál es la cantidad de esos ceros que pertencen a $\mathbb{R}^n$. Este problema es de naturaleza no lineal. Para resolverlo, vamos a traducirlo a uno de Álgebra Lineal. De esa manera podremos utilizar todas las herramientas de que disponemos en ese campo.

Los polinomios en una variable con coeficientes en un cuerpo $\mathbb{K}$, forman un anillo ($\mathbb{K}[X]$). Este anillo tiene la propiedad de ser íntegro. Esto quiere decir que no existen divisores propios de cero. Si tomamos un conjunto de polinomios $\mathcal{P} = \{f_1, \ldots, f_s\} \subseteq \mathbb{K}[X]$, definimos el ideal generado por $\mathcal{P}$, $<\mathcal{P}>$, como el conjunto $\{\sum_{i=1}^{s} u_i(x)P_i\}$, donde $u_i(x) \in \mathbb{K}[X]$. $\mathbb{K}[X]$ es un dominio principal, que quiere decir que todo ideal está generado por un elemento del mismo. En este caso, ese generador será $\mathrm{mcd}(P_1, \ldots, P_s)$. Este máximo común divisor se puede ir calculando de a pares, utilizando el algoritmo que se remonta a Euclides. Para decidir si un polinomio pertenece a un ideal, basta ver si es divisible por el generador del ideal.

Ahora, consideraremos los polinomios en varias variables, $X_1, \ldots, X_n$, con coeficientes en un cuerpo $\mathbb{K}$. Este conjunto también es un anillo, y lo notaremos $\mathbb{K}[X_1, \ldots, X_n]$. Este anillo es nuevamente un dominio íntegro, pero no es principal. Esto implica que un ideal puede no estar generado por un solo polinomio. El Teorema de la Base de Hilbert muestra que en este nuevo anillo, todo ideal está generado por un número finito de polinomios.

Antes de continuar, vamos a establecer algunas cuestiones de notación. Para no tener que escribir muchos nombres de variables, usaremos X^α, $\alpha \in \mathbb{N}^n$, para indicar $X_1^{\alpha_1} \cdots X_n^{\alpha_n}$ ($\mathbb{N} = \{0, 1, \ldots\}$). Además, usaremos indistintamente las mayúsculas y las minúsculas para referirnos a las variables. Cuando $n \leq 3$, usaremos x, y y z para referirnos a x_1, x_2 y x_3, respectivamente. En el caso multivariado, el grado de un monomio X^α es $\sum_{i=1}^{n} \alpha_i$. El grado de un polinomio multivariado será el máximo de los grados de sus monomios.

Ahora sí, retomamos nuestra discusión sobre los ideales de polinomios en varias variables. En el caso de una variable, vimos la existencia de un

algoritmo para determinar si un polinomio pertenece a un ideal. Para generalizar al caso multivariado el algoritmo de división del caso univariado, podríamos decir que dividir un polinomio $p \in \mathbb{K}[X_1, \ldots, X_n]$ por otro polinomio $s \in \mathbb{K}[X_1, \ldots, X_n]$ es dar polinomios $q, r \in \mathbb{K}[X_1, \ldots, X_n]$ tales que $p = qs + r$, con $r = 0$ o de menor grado que s. Pero esto nos puede traer problemas: $x^2 + 3x + y^2$ dividido $3x$ nos daría $x^2 + 3x + y^2 = 3x(x/3 + 1) + y^2$. Aquí, r tiene grado 2 y s tiene grado 1. Esto se podría evitar si dijéramos que y^2 es menor que $3x$. Esta idea da lugar a los órdenes monomiales. Un orden monomial es una relación de orden definida sobre el conjunto de monomios x^α, $\alpha \in \mathbb{N}^n$,que verifica tres propiedades: es compatible con el producto, es un orden total y es un buen orden. Esta última es la que nos permite garantizar la finitud de varios algoritmos: si obtenemos monomios siempre menores, en algún momento terminamos. El monomio inicial de un polinomio es el mayor de sus monomios, de acuerdo al orden monomial. Un ejemplo de orden monomial es el lexicográfico: se toma $x^\alpha < x^\beta \Leftrightarrow (\exists i)(1 \leq i \leq n \wedge \alpha_i < \beta_i \wedge (\forall j)(1 \leq j < i \Rightarrow \alpha_j = \beta_j))$. Este orden es útil para la eliminación de variables, pero no suele ser el más apropiado para implementaciones.

Ahora que tenemos un orden monomial, podemos resolver satisfactoriamente el problema de la pertenencia a un ideal multivariado generado por un solo polinomio. ¿Qué pasa si nuestro ideal está generado por varios polinomios, digamos $f_1, \ldots, f_s$? En este caso, podemos seguir generalizando nuestro algoritmo de división. Dividir un polinomio p por un conjunto de polinomios $f_1, \ldots, f_s$ es dar un conjunto de polinomios $q_1, \ldots, q_s, r$ tales que $\sum_{i=1}^{s} q_i f_i + r = p$ y que el monomio inicial de r no es divisible por ninguno de los monomios iniciales de los f_i. Esto nos daría una primera aproximación al problema de la pertenencia a un ideal generado por más de un polinomio: si $r = 0$, el polinomio pertenece al ideal. Lamentablemente, al buscar los q_i vamos a tener que dar algún orden a los f_i, y este orden no podrá salir solamente del orden monomial. Esto lleva a que un polinomio puede estar en un ideal, pero su resto, con algún orden de los $f_1, \ldots, f_s$, al dividirlo por los generadores puede no dar cero.

Para resolver este nuevo problema, Buchberger ideó un algoritmo que permite obtener un conjunto de generadores con la propiedad de que los restos de la división por ellos no dependen del orden que se use para dividir. Un conjunto de generadores con esta propiedad se llama base de Gröbner. No constituye una base en el sentido tradicional, ya que no tenemos una noción de independencia lineal, pero sí forman un conjunto de generadores. Este algoritmo es el que ha permitido la expansión explosiva del área en los últimos años. Desafortunadamente, se puede demostrar que su complejidad en el peor caso es doblemente exponencial en el número de variables del

anillo. Sin embargo, existen implementaciones muy eficientes para un número moderado de variables, o para familias de polinomios que satisfagan ciertas hipótesis geométricas.

Dado un conjunto finito de polinomios $\{f_1, \ldots, f_s\}$ con coeficientes en un cuerpo $\mathbb{K}$, el conjunto de ceros comunes en $\overline{\mathbb{K}}^n$, donde $\overline{\mathbb{K}}$ es la clausura algebraica de $\mathbb{K}$, coincide con los ceros $\mathbf{V}(I) = \{p \mid p \in \overline{\mathbb{K}}^n \wedge f(p) = 0 \; \forall \; f \in I\}$, con $I = < f_1, \ldots, f_s >$ el ideal generado por los polinomios dados, es decir el conjunto de todas las combinaciones lineales de $f_1, \ldots, f_s$ con coeficientes en el anillo $\mathbb{K}[X_1, \ldots, X_n]$. $\mathbf{V}(I)$ se llama la variedad generada por el ideal I. La variedad generada por un ideal puede ser finita o infinita, dependiendo del ideal. Si es finita, entonces el ideal se dice "de dimensión cero".

A partir de un ideal I de polinomios en $\mathbb{K}[X_1, \ldots, X_n]$, podemos considerar otro conjunto, $\mathbb{K}[X_1, \ldots, X_n]/I$, el conjunto cociente de la relación $f \sim g$ sii $f - g \in I$. Este cociente es un $\mathbb{K}$–espacio vectorial, y más aún, es un anillo. Aquí es donde entrará en juego el Álgebra Lineal. La dimensión de este espacio vectorial es igual a la cantidad de ceros complejos del ideal, contados con multiplicidad. Entonces, el ideal I es de dimensión cero si, y sólo si el cociente al que da lugar tiene dimensión finita. Determinar si la dimensión del cociente es finita es sencillo teniendo una base de Gröbner para el ideal. Para que la dimensión sea finita, es necesario y suficiente que para cada una de las variables haya un polinomio en la base que tenga como monomio inicial una potencia positiva de esa variable. Si tenemos una base de Gröbner para el ideal, es fácil ver que se puede encontrar una base para el cociente: todos los monomios mónicos que no sean divisibles por los monomios iniciales de ningún polinomio de la base.

En el presente trabajo, estudiaremos ideales de dimensión cero en el anillo $\mathbb{R}[X_1, \ldots, X_n]$. En este caso, $\overline{\mathbb{K}} = \mathbb{C}$. Nos concentraremos en acotar y contar la cantidad de raíces reales de estos ideales. Es decir, $\#(\mathbf{V}(I) \cap \mathbb{R}^n)$.

1.2 El sistema SINGULAR

SINGULAR es un programa libre para cálculo simbólico desarrollado desde 1984 en la Universidad de Kaiserslautern. Sus principales áreas de aplicación son la Geometría Algebraica, el Álgebra Conmutativa y la Teoría de Singularidades. Los principales objetos con los que trabaja SINGULAR son los polinomios en una o más variables, con coeficientes en distintos tipos de cuerpos, como $\mathbb{Q}$, $\mathbb{Z}_p$ ($p \leq 32003$), extensiones algebraicas y trascendentes, números reales de punto flotante y números complejos del mismo tipo (con hasta 32767 dígitos decimales). En cada anillo que se utilice, se debe declarar un orden monomial. Se puede elegir entre el lexicográfico, grevlex, órdenes

con peso, etc.

SINGULAR ha sido empleado exitosamente en numerosas aplicaciones, como por ejemplo, robótica y geometría molecular, diseño de circuitos, estudios epidemiológicos y el estudio de reacciones químicas que surgen durante el moldeado en caliente del vidrio.

El programa ha sido desarrollado bajo Unix, pero también ha sido transportado a otros sistemas operativos, como Windows o MAC Os. La manera más común de utilizarlo es dentro del editor Emacs. Para ello, los autores han desarrollado (y distribuyen con el programa) un conjunto de macros que facilitan su uso.

Los principales algoritmos de las áreas que aborda han sido implementados en su kernel (desarrollado en C++), e incluyen el algoritmo de Buchberger (y variantes optimizadas), factorizaciones, resultantes e intersección de ideales, entre otros. Además de estos algoritmos, el programa puede emplear bibliotecas de funciones desarrolladas en un lenguaje similar a C. La distribución de SINGULAR incluye unas 50 bibliotecas con múltiples desarrollos. Precisamente, este trabajo fue implementado en tres bibliotecas escritas en ese lenguaje. Nuestro objetivo es la incorporación de estas bibliotecas a la distribución oficial, una vez que hayan sido ajustadas a las convenciones de formato de los autores.

El trabajo en SINGULAR gira en torno de anillos. Antes de poder hacer algo útil, debe declararse el anillo de polinomios en el que se trabajará. Esa declaración incluye el cuerpo del que saldrán los coeficientes, los nombres de las variables y el orden monomial que se utilizará. Una vez declarado eso, se pueden definir ideales (dando sus generadores), polinomios y listas, entre otros. Los cómputos pueden utilizar números enteros (o racionales) de tamaño arbitrario. También se pueden utilizar parámetros. Para más detalles, ver el apéndice A.

Además de los algoritmos y las bibliotecas, la distribución del programa incluye una profusa documentación, con muchos ejemplos. La documentación de las bibliotecas es generada automáticamante. Para ello, los autores han definido un formato estándar de documentación, que permite esa generación.

Capítulo 2

Desarrollo

2.1 Raíces reales de polinomios univariados

En esta sección, estudiaremos los distintos algoritmos implementados para hallar la cantidad de raíces reales de un polinomio en una variable. Todas las funciones mencionadas aquí pertenecen a la biblioteca `urrc.lib` (por univariate real roots counting).

2.1.1 Regla de Descartes

Como primera aproximación a la cantidad de raíces reales de un polinomio univariado, implementamos la regla de los signos de Descartes. Sea $L = (a_0, \ldots, a_n)$ una lista de números reales. La variación de signos de L, notada $V(L)$, está definida como la cantidad de pares $(i, i+k)$, $k \geq 1$, tales que:

1. $a_i a_{i+k} < 0$
2. $a_{i+r} = 0$, $0 < r < k$.

Por ejemplo, $V(1, -1, 0, 1) = \#\{(1, -1), (-1, 1)\} = 2$.

La Regla de Descartes dice que el número de raíces reales positivas de un polinomio, contadas con multiplicidad, está acotado superiormente por la variación de signos de la lista de coeficientes del polinomio. Además, esta cota es siempre congruente a la cantidad de raíces positivas módulo 2. Entonces, si obtenemos un número impar de variaciones, sabemos con seguridad que hay al menos una raíz positiva. Para contar las variaciones de signos, basta considerar sólo aquellos términos del polinomio que tienen coeficientes no nulos. Así, $V(x^3 - x^2 + 1) = 2$, donde $V(p)$ es la función de variación de signos, aplicada a la secuencia de coeficientes $(1, -1, 0, 1)$, o bien a la secuencia $(1, -1, 1)$.

Si un número real negativo a es raíz de un polinomio $p(x)$, entonces $-a$ es raíz de $p(-x)$. Esto quiere decir que la regla de los signos también nos permite obtener una cota para la cantidad de raíces reales negativas de un polinomio. Además, es trivial decidir si 0 es raíz de un polinomio, y con qué multiplicidad. De esa forma, tenemos acotada la cantidad total de raíces reales.

Afortunadamente, esta cota superior es igual a la cantidad de raíces positivas cuando todas las raíces son reales. Además, nos da una cota para las raíces contadas con multiplicidad. Por ejemplo, $v(x^2 - 4x + 4) = v((x-2)^2) = 2$. Aquí vemos que el 2 aparece contado dos veces.

En nuestra biblioteca, la implementación de esta regla está repartida en tres funciones `varsigns`, `descartes` y `descartest`.

La función `varsigns` toma una lista de números y calcula la cantidad de variaciones de signos que aparecen en ella, ignorando los ceros.

Ejemplo 2.1

```
> varsigns(list(1,-1,0,1));
2
```

Más adelante, veremos otras aplicaciones de la función de variación de signos.

El comando `descartes` toma un polinomio y calcula la cota para la cantidad de raíces reales positivas contadas con multiplicidad, usando las propiedades mencionadas arriba.

Ejemplo 2.2

```
> descartes((x-1)*(x-2)*(x+4)*(x+1)^2);
2
> descartes((x-1)*(x-2)*(x+4)*(x^2+1));
4
```

El comando `descartest` acota el número total de raíces reales de un polinomio (positivas, negativas y/o cero) contadas con multiplicidad.

Ejemplo 2.3

```
> descartest((x-1)*(x-2)*(x+4)*(x+1)^2);
5
> descartest((x-1)*(x-2)*(x+4)*(x^2+1));
5
```

En el ejemplo de arriba, vemos cómo las raíces son contadas con su multiplicidad. Además, también podemos ver que la cota no nos da la cantidad exacta de raíces si el polinomio tiene además raíces complejas.

Un detalle importante con respecto a la función `descartes` es que tuvimos que desarrollar dos funciones auxiliares: una para decirnos si el polinomio es univariado (`isuni`), y otra para decirnos cuál es esa única variable (`whichvariable`). La complejidad de ambas es $O(n)$, donde n es la cantidad de variables del anillo sobre el cuál estamos trabajando. Esto no afecta demasiado la complejidad de la función, ya que la cantidad de términos de un polinomio suele ser mayor que la cantidad de variables del anillo. En definitiva, la complejidad de `descartes` es $O(\text{máx}(n, d))$, donde d es el grado del polinomio y n la cantidad de variables del anillo. La de `descartest` es la misma.

El comando `whichvariable` recibe un monomio y nos dice cuál de las variables del anillo aparece en él. Si aparece más de una variable, la función devuelve 0. Si no, devuelve la variable que corresponde.

Ejemplo 2.4

```
> whichvariable(2*x^8*y^2);
0
> whichvariable(12*y^6);
y
> whichvariable(3);
x
```

En este ejemplo vemos que si el monomio recibido es constante, entonces `whichvariable` devuelve la primera variable del anillo en el que se está trabajando. Además, se ve que la función ignora los coeficientes.

El comando `isuni` nos devuelve 0 si el polinomio no es univariado, y nos devuelve la variable en caso de que sí lo sea. Esto nos ahorra una llamada a `whichvariable` en las funciones que usan `isuni`.

Ejemplo 2.5

```
> isuni(2*x*y^3+3*x^8+2*z*x);
0
> isuni(y^3-y^12+3*y-2);
y
```

El desarrollo de los resultados de esta sección se encuentra en [BR/90].

2.1.2 Teorema de Boudan-Fourier

El Teorema de Boudan-Fourier nos permite generalizar la Regla de Descartes a un intervalo de la recta. Es decir, nos permite acotar las raíces de un polinomio en un intervalo, con multiplicidad.

Si $P \in \mathbb{R}[X]$ es un polinomio de grado d, definimos $\mathrm{Der}(P)$ como la lista $(P, P', \ldots, P^{(d)})$. Definimos $\mathrm{Der}(P, x)$ como la lista que resulta de evaluar $\mathrm{Der}(P)$ en el punto x. Con estas definiciones, el Teorema de Boudan-Fourier dice que

1. $\#\{x | x \in (a, b] \wedge P(x) = 0\} \leq V(\mathrm{Der}(P, a)) - V(\mathrm{Der}(P, b))$
2. $\#\{x | x \in (a, b] \wedge P(x) = 0\} \equiv V(\mathrm{Der}(P, a)) - V(\mathrm{Der}(P, b)) \pmod 2$

Es decir, nos da una cota para la cantidad de raíes de P en $(a, b]$, y esta cota tiene la misma paridad que la cantidad exacta de raíces. En este Teorema, las raíces están contadas con multiplicidad.

En nuestra biblioteca, implementamos la función `boufou`, que aplica el Teorema para calcular la cota.

Ejemplo 2.6

```
> boufou((x-1)*(x-2)*(x+4)*(x+1)^2,-2,10);
4
> boufou((x-1)*(x-2)*(x+4)*(x^2+1),-2,10);
4
```

Calcular la derivada de un polinomio es $O(d)$, donde d es el grado del mismo. La función debe calcular d derivadas. En conclusión, `boufou` es $O(d^2)$.

2.1.3 La secuencia de Sturm

La primera técnica que implementamos para contar la cantidad de raíces reales de un polinomio en una variable fue la basada en la secuencia de Sturm. Ésta cuenta las raíces sin multiplicidad. Esto es, nos da la cantidad de raíces reales distintas.

Algoritmo 1 Secuencia de Sturm

Entradas: Un polinomio $P \in \mathbb{R}[X]$.
Salida: Una lista de polinomios $\mathrm{St}_1, \ldots, \mathrm{St}_s$ una secuencia de Sturm de P.

1: $\mathrm{St}_1 \leftarrow P$
2: $\mathrm{St}_2 \leftarrow \mathrm{St}_1'$
3: $i \leftarrow 3$
4: **mientras** $-\mathrm{Resto}(\mathrm{St}_{i-2}, \mathrm{St}_{i-1}) \neq 0$ **hacer**
5: $\quad \mathrm{St}_i \leftarrow -\mathrm{Resto}(\mathrm{St}_{i-2}, \mathrm{St}_{i-1})$
6: $\quad i \leftarrow i + 1$
7: **fin mientras**

A partir de un polinomio $P \in \mathbb{R}[X]$, el Algoritmo 1 nos dará una lista $(\mathrm{St}_1, \ldots, \mathrm{St}_s)$ de polinomios. Estos polinomios forman una secuencia de Sturm de P (ver [BR/90]).

El Teorema de Sturm dice que, si $P \in \mathbb{R}[X]$, $(\mathrm{St}_1, \ldots, \mathrm{St}_s)$ es una secuencia de Sturm de P, a y b son dos números reales, con $a < b$, y $P(a)P(b) \neq 0$, entonces, la cantidad de raíces de P en el intervalo $[a, b]$ está dada por la fórmula

$$V(\mathrm{St}_1(a), \ldots, \mathrm{St}_s(a)) - V(\mathrm{St}_1(b), \ldots, \mathrm{St}_s(b))$$

donde V es la función de variación de signos de la sección anterior.

Este Teorema nos permitiría, en principio, determinar la cantidad de raíces reales de un polinomio P en un intervalo $[a, b]$. Sin embargo, esto puede no ser sencillo si uno de los extremos fuera raíz de P. Supongamos, sin pérdida de generalidad, que $P(a) = 0$. Entonces querríamos dividir P por $x - a$ para obtener un polinomio que no tenga a como raíz, y así poder aplicar el Teorema de Sturm. Pero si a es una raíz múltiple, esto puede ser muy costoso. Para conocer la multiplicidad, debemos derivar P en forma sucesiva hasta que lleguemos a un polinomio que no se anule en a. Si tomamos, por ejemplo, $P = (x-1)^{1000}$, será necesario calcular 1000 derivadas para determinar que 1 tiene multiplicidad 1000. Otra forma de calcular la multiplicidad es realizar un desplazamiento del tipo $x \mapsto (x - a)$, y calcular la multiplicidad de 0. Este enfoque puede ser igual de costoso que el de las derivadas si estamos trabajando con grados altos. Una alternativa es empezar quitándole a todas las raíces de P su multiplicidad. Una versión ligeramente distinta del Teorema de Sturm dice que si $P \in \mathbb{R}[X]$, P tiene todas sus raíces simples, $(\mathrm{St}_1, \ldots, \mathrm{St}_s)$ es una secuencia de Sturm de P, a y b son dos números reales y $a < b$, entonces, la fórmula

$$V(\mathrm{St}_1(a), \ldots, \mathrm{St}_s(a)) - V(\mathrm{St}_1(b), \ldots, \mathrm{St}_s(b))$$

nos da la cantidad de raíces distintas de P en el intervalo $(a, b]$.

Ahora estamos frente a un nuevo problema: ¿cómo quitar la multiplicidad a las raíces de P? Si a es raíz de P con multiplicidad k, entonces también es raíz de P', pero con multiplicidad $k - 1$. Entonces, $(x - a)^{k-1}$ divide a P y a P', y por ende al $\mathrm{mcd}(P, P')$. Este análisis se puede hacer para todas las raíces de P, y nos mostrará que $\mathrm{mcd}(P, P')$ tiene todas las raíces de P con multiplicidad "uno menos" que la multiplicidad que tienen en P. Entonces, hacer

$$\mathrm{red}(P) = \frac{P}{\mathrm{mcd}(P, P')}$$

nos dará un polinomio que tiene las mismas raíces que P, pero todas simples. Este polinomio se llama el reducido de P. Para contar las raíces de P en

$[a, b]$, primero calculamos su reducido, descartamos los extremos, calculamos su Secuencia de Sturm, evaluamos la secuencia en a y en b y contamos las variaciones de signos.

El algoritmo para calcular el reducido de P es idéntico al algoritmo que calcula la secuencia de Sturm de P. Entonces, calcular primero el reducido y sobre eso la secuencia puede involucrar hacer dos veces el mismo trabajo, aunque potencialmente con polinomios de menor grado la segunda vez.

Si P es un polinomio con coeficientes genéricos sobre el cuerpo de los números reales, entonces no tendrá ni a a ni a b como raíces. Si alguno fuera raíz, entonces, tendría multiplicidad 1. El número de pasos requerido en el algoritmo de Euclides para calcular el mcd entre un polinomio P y su derivado no depende del grado de P, sino de la cantidad de raíces distintas que tiene P. Si suponemos que vamos a trabajar con polinomios genéricos, entonces este algoritmo tendrá un número de iteraciones que depende del grado (pues las raíces serán todas distintas). Si suponemos que el polinomio tiene estas características, entonces podemos dividirlo por $(x - a)$ para lograr que a deje de ser una raíz. Con la hipótesis de coeficientes genéricos, tendremos que hacer a lo sumo una sola división. Si, en cambio, suponemos que las raíces de nuestro P tienen multiplicidad alta, entonces nos convendrá utilizar el algoritmo de Euclides.

Elegimos esta opción porque nos permite aplicar el Teorema de Sturm a polinomios que tengan sus raíces en los extremos del intervalo.

En nuestra biblioteca, implementamos la función `sturmseq`, que toma un polinomio y nos devuelve una Secuencia de Sturm utilizando el Algoritmo 1

Ejemplo 2.7

```
> sturmseq(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21));
[1]:
   5x5-200x4+2440x3-9190x2-5565x+48510
[2]:
   25x4-800x3+7320x2-18380x-5565
[3]:
   304x3-6198x2+33860x-39606
[4]:
   31956095/23104x2-198818065/11552x+1002381135/23104
[5]:
   234030853463040000/40847680305961x-
     1476837782848512000/40847680305961
[6]:
   1789393498846277486 89/17642456970116180
```

Implementamos también la función `sturm`, que toma un polinomio P y dos números a y b, y nos devuelve la cantidad de raíces de P en $(a, b]$, utilizando el procedimiento mencionado más arriba.

Ejemplo 2.8

```
> sturm(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21),-2,9);
3
```

Utilizando la función `sturm`, implementamos el comando `allrealst`. Este comando nos devuelve verdadero (1) o false (0), dependiendo de si el polinomio que recibe como parámetro tiene todas sus raíces reales. Para ello, debimos calcular primero una cota para el valor absoluto de las raíces reales de un polinomio. Elegimos utilizar el máximo valor absoluto de los coeficientes del polinomio, más uno (ver [CLO/98]). Para calcular esta cota, antes de considerar los coeficientes, dividimos el polinomio por su coeficiente principal. Así obtenemos un polinomio mónico con las mismas raíces, y sirve nuestra cota. Esta cota se calcula con la función `maxabs`.

Ejemplo 2.9

```
> maxabs(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21));
48511
> sturm(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21),-48511,48511);
5
> allrealst(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21));
1
> allrealst(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21)*(x^2+1));
0
```

El algoritmo para calcular la secuencia de Sturm tiene complejidad $O(d^2)$, donde d es el grado del polinomio P. Esta es una complejidad razonable. Sin embargo, el tamaño de los coeficientes puede crecer en forma cuadrática con respecto al grado de P. Sería deseable encontrar una forma de achicar los coeficientes. De eso trata la siguiente sección. Para una demostración de la correctitud del algoritmo, y de su complejidad, ver [BPR/03] y [GVRR/99].

2.1.4 La secuencia de Sturm–Habicht

Existe una alternativa a la secuencia de Sturm que presenta un mejor comportamiento en sus coeficientes. Esta nueva secuencia, la secuencia de Sturm–Habicht, también nos permite calcular la cantidad de raíces reales de un polinomio. Esta secuencia también está definida algorítmicamente. El algoritmo 2 nos dará la secuencia de Sturm–Habicht de un polinomio P.

Algoritmo 2 Secuencia de Sturm–Habicht

Entradas: Un polinomio $P \in \mathbb{R}[X]$ de grado d.

Salida: Una lista de polinomios $\mathrm{StHa}_1, \ldots, \mathrm{StHa}_{d+1}$ la secuencia de Sturm–Habicht de P.

$\mathrm{StHa}_1 \leftarrow \mathrm{red}(P)$

$\overline{h}_{d+1} \leftarrow 1/h_{d+1}$ {h_{d+1} denota el coeficiente principal del polinomio StHa_{d+1}}

$\mathrm{StHa}_d \leftarrow \mathrm{StHa}'_{d+1}$

$j \leftarrow d+1$

repetir

  $k \leftarrow \mathrm{grado}(\mathrm{StHa}_{j-1})$ {Si $\mathrm{StHa}_{j-1} = 0$, $k \leftarrow 0$}

  si $k < j-2$ **entonces**

    $\mathrm{StHa}_l \leftarrow 0$, $\forall l \ / \ k < l < j-1$

  fin si

  $c_{j-1} \leftarrow h_{j-1}$

  si $k < j-1$ **entonces**

    $\overline{h}_{j-1} \leftarrow c_{j-1}$

    $\overline{h}_l \leftarrow (-1)^{j-l-1} \frac{\overline{h}_{l+1} c_{j-1}}{\overline{h}_j}$, $\forall l \ / \ k \leq l < j-1$.

  fin si

  $\mathrm{StHa}_k \leftarrow \frac{\overline{h}_k \mathrm{StHa}_{j-1}}{c_{j-1}}$

  $h_k \leftarrow \overline{h}_k$

  $\mathrm{StHa}_{k-1} \leftarrow -\frac{1}{h_j \overline{h}_j} \mathrm{Resto}(c_{j-1} h_k \mathrm{StHa}_j, \mathrm{StHa}_{j-1})$

hasta que StHa_1 ya haya sido calculado

En nuestra biblioteca, el comando `sthaseq` nos devuelve una secuencia de Sturm–Habicth para un polinomio dado.

Ejemplo 2.10

```
> sthaseq(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21));
[1]:
   5x5-200x4+2440x3-9190x2-5565x+48510
[2]:
   25x4-800x3+7320x2-18380x-5565
[3]:
   190000x3-3873750x2+21162500x-24753750
[4]:
   79890237500x2-994090325000x+2505952837500
[5]:
   633090735000000000x-3995081433000000000
[6]:
   3678904075392000000000000
```

En este ejemplo, vemos que los coeficientes de esta secuencia tienen un tamaño menor que en el caso de la secuencia de Sturm y también nos permite calcular la cantidad de raíces reales de P. Dada una lista de números reales $l = (a_0, \ldots, a_d)$, definimos $Perm(l)$ como el número de pares $(i, i+k)$, $k \geq 1$ que verifican

1. $a_i a_{i+k} > 0$
2. $a_{i+r} = 0$, $0 < r < k$.

Esta función nos da las permanencias de signos en la lista l, ignorando los ceros. Por ejemplo, $Perm(1, 0, 0, -3, -2) = 1$.

Si tomamos $l = a_0, \ldots, a_d$, la lista de coeficientes principales de los polinomios distintos de cero de la secuencia de Sturm–Habicht de P, entonces la cantidad de raíces reales de P está dada por

$$Perm(l) - V(l)$$

donde V es la función de variación de signos. Utilizando esta propiedad, implementamos la función `stha`, que nos da la cantidad de raíces reales de un polinomio.

Ejemplo 2.11

```
> stha(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21));
5
> stha(5*(x-3)*(x+2)*(x-11)*(x-7)*(x-21)*(x2+2x+2));
5
```

Utilizando la función `stha`, construimos la función `allreal`, que nos dice si todas las raíces de un polinomio univariado son reales. El siguiente ejemplo ilustra los comentarios sobre el polinomio de Wilkinson de la Introducción.

Ejemplo 2.12

```
> poly p = (x+1)*(x+2)*(x+3)*(x+4)*(x+5)*(x+6)*(x+7)*(x+8)
           *(x+9)*(x+10)*(x+11)*(x+12)*(x+13)*(x+14)*(x+15)
           *(x+16)*(x+17)*(x+18)*(x+19)*(x+20);
> allreal(p)
1
> poly pprime = p + 1/1000000000*x^19;
> allreal(pprime);
0
> stha(pprime);
12
```

El cálculo de la secuencia de Sturm–Habicht tiene complejidad $O(d^2)$, donde d es el grado de P. Además, si τ es la cantidad de bits del coeficiente más grande de P, entonces el tamaño de los coeficientes de los polinomios de la secuencia es $O(\tau d)$. Para la correctitud del algoritmo, y un análisis de su complejidad, ver [GVRR/99]. Existe una formulación esencialmente idéntica de la secuencia de Sturm–Habicht, pero utilizando subresultantes. Los resultados son los mismos, y se pueden ver en [GVT/97] y [BPR/03].

2.2 Raíces reales de sistemas polinomiales

En esta sección, estudiaremos dos métodos que utilizan el Álgebra Lineal para resolver el problema de contar la cantidad de raíces reales del sistema $f_1 = \ldots = f_k = 0$, con $f_i \in \mathbb{R}[X_1, \ldots, X_n]$. Nuevamente recordamos que nuesta primera hipótesis, en estos sistemas, es que el número de soluciones complejas es finito. En la Introducción vimos cómo se puede verificar esto utilizando bases de Gröbner. Las funciones descriptas aquí componen la biblioteca `mrrc.lib`

2.2.1 Formas bilineales y sus signaturas

Sean $f_1, \ldots, f_k$ nuestros polinomios. Consideremos el ideal $I =< f_1, \ldots, f_k >$. Como vimos en la Introducción, el cociente $A = \mathbb{R}[X_1, \ldots, X_n]/I$ tiene estructura de espacio vectorial. Además, si el número de elementos de $\mathbf{V}(I)$ es finito, entonces A es un $\mathbb{R}$–espacio vectorial de dimensión finita. Como vimos en la Introducción, A admite una base $B = \{x^{\alpha_1}, \ldots, x^{\alpha_s}\}$ formada

por monomios; más aún, dado un orden monomial, hay una manera estándar de obtener esta base de A: todos los monomios que no son divisibles por ninguno de los monomios iniciales de los elementos de una base de Gröbner del ideal I para el orden dado.

Con estas hipótesis, podemos estudiar las funciones de multiplicación por un polinomio $f \in \mathbb{R}[X_1, \ldots, X_n]$

$$m_f : A \to A,$$

dadas por $m_f([g]) = [fg] = [f][g]$. Estas funciones resultan ser transformaciones $\mathbb{R}$–lineales. Como tales, pueden ser representadas en la base B por una matriz. Nuestro comando `mat`, nos da esta matriz. Para ello, debemos pasarle como parámetros el polinomio f, una base de monomios y una base de Gröbner para el ideal.

Ejemplo 2.13

```
> ideal i = x^2+y^2+z^2-4,x^2+2*y^2-5,x*z-1;
> i = std(i);
> i;
i[1]=xz-1
i[2]=y2-z2-1
i[3]=x2+2y2-5
i[4]=2z3+x-3z
> ideal b = qbase(i);
> print(mat(x+2*y*z-3*x^5+2*x*y*z,b,i));
0,   40, 18, 2, 0,  0,  2,  0,
-9,  0,  0,  -1,2,  -20,0,  0,
7,   0,  0,  3, 0,  18, 2,  2,
5,   2,  5,  0, 40, 2,  18, 0,
-5/2,2,  -1, -9,0,  -1, 0,  -20,
0,   -42,-20,0, 2,  0,  0,  2,
13/2,2,  5,  7, 0,  5,  0,  18,
-1,  2,  -1, 0, -42,2,  -20,0
```

En este ejemplo, utilizamos la función `qbase` para obtener una base del cociente A. Podríamos haber utilizado el comando `kbase` de SINGULAR, pero éste no nos da una base ordenada. `qbase` se encarga de hacer ese trabajo, y de dejar la base ordenada de acuerdo al orden monomial con el que se está trabajando.

Consideremos ahora las funciones bilineales $\varphi_h : A \times A \to \mathbb{R}$, con $h \in \mathbb{R}[X_1, \ldots, X_n]$, dadas por $\varphi_h([f], [g]) = \text{traza}(m_{hfg})$. La traza de una transformación lineal es la suma de los elementos de la diagonal de una matriz

que represente a f en cualquier base. Afortunadamente, nuestra función φ_h es bilineal y simétrica (como el producto interno en $\mathbb{R}^n$). Esto quiere decir que podemos calcular su matriz asociada en la base B, tomando

$$([\varphi_h]_B)_{ij} = \varphi_h(x^{\alpha_i}, x^{\alpha_j}) = \text{traza}(m_{hx^{\alpha_i}x^{\alpha_j}}) = ([\varphi_h]_B)_{ji}$$

Estas funciones tienen una propiedad importante (demostrada por Hermite, ver [BPR/03]), y es que $\text{sig}(\varphi_h) = \#\{p \mid p \in \mathbf{V}(I) \cap \mathbb{R}^n \wedge h(p) > 0\} - \#\{p \mid p \in \mathbf{V}(I) \cap \mathbb{R}^n \wedge h(p) < 0\}$. Si φ es una función bilineal simétrica, $\text{sig}(\varphi)$, la signatura de φ, es la cantidad de autovalores positivos menos la cantidad de autovalores negativos de cualquier matriz asociada a φ.

Entonces, restaría ver si podemos calcular fácilmente su signatura. Pero la matriz de φ_h es simétrica y tiene todas sus coordenadas en $\mathbb{R}$. A partir de eso, se puede demostrar que todos sus autovalores son reales. Entonces, podemos aplicar la regla de Descartes a los polinomios $\chi(x)$ y $\chi(-x)$ para obtener la cantidad exacta de autovalores positivos y negativos de φ_h ($\chi(x)$ es el polinomio característico de φ_h).

La función `mat` nos permite obtener fácilmente las trazas de las funciones m_f. La función `bil` calcula la matriz de φ_h. Esta toma los mismos parámetros que `mat`, pero nos devuelve la matriz de φ_h en la base recibida.

Para realizar todos los cálculos descriptos arriba, tuvimos que implementar varias funciones además de `mat`.

A partir de `mat` pudimos construir `bil`, que toma los mismos parámetros—polinomio, base del cociente, ideal—, pero nos devuelve la matriz de φ_h. En realidad, si bien `mat` nos da la matriz de la función de multiplicación, para armar la matriz asociada a la forma bilineal sólo necesitamos la traza de esa matriz. Entonces, con la función `matrace` directamente calculamos la suma de los elementos de la diagonal. La cantidad de elementos a calcular es m. Esto es una mejora considerable frente a la cantidad total de elementos de la matriz—m^2.

Ejemplo 2.14

```
> matrix m = bil(3x2+2zy,b,i); print(m);
33,0, 0, 0, 22,42,19,0,
0, 96,60,22,0, 0, 0, 28,
0, 60,42,19,0, 0, 0, 22,
0, 22,19,18,0, 0, 0, 24,
22,0, 0, 0, 60,28,36,0,
42,0, 0, 0, 28,60,22,0,
19,0, 0, 0, 36,22,24,0,
0, 28,22,24,0, 0, 0, 36
```

Una vez que tuvimos `bil` funcionando, fue sencillo calcular la signatura de estas matrices. Como mencionamos antes, utilizamos `varsigns` con el polinomio característico de la matriz devuelta por `bil`. Eso quedó implementado en la función `symmsig` (la matriz debe ser simétrica para poder calcular su signatura de esa forma).

De todo lo que vimos hasta ahora, tomando $h = 1$, podemos deducir que $\text{sig}(\varphi_1)$ nos dará la cantidad de puntos reales en $\mathbf{V}(I)$. Luego, podemos calcular como sigue el número de raíces reales del ideal i del ejemplo 2.13:

Ejemplo 2.15

```
> symmsig(bil(1,b,i));
8
```

Si calculamos una Base de Gröbner lexicográfica de i, para lo cual hay que declarar el orden lexicográfico en el anillo, obtendremos:

$$\texttt{i} = < 2z^4 - 3z^2 + 1, y^2 - z^2 - 1, x + 2y^2z - 5z > .$$

A partir de esta información, se ve fácilmente que el ideal tiene efectivamente 8 raíces reales. Sin embargo, el cálculo de Bases de Gröbner para órdenes lexicográficos es en general muy costoso, y no es posible en general detectar el número de soluciones reales fácilmente a partir de este cálculo.

Siguiendo los lineamientos del equipo de desarrollo de SINGULAR para las funciones de bibliotecas estándar, la función `symmsig` verifica que su parámetro tenga la forma apropiada, a saber, que sea una matriz simétrica. Revisar esto lleva tiempo, y no es necesario si sabemos que vamos a llamar a la función con un matriz que es efectivamente cuadrada y simétrica. Entonces, implementamos un procedimiento interno a la biblioteca, `mysymmsig`, que realiza el mismo cálculo, pero sin verificar que la matriz sea simétrica. Este comando se puede utilizar a través del procedimiento `SQ`, que calcula la Sturm Query de un polinomio en un ideal. Esto es, dado un polinomio P y un conjunto de puntos Z (que puede estar dado como las raíces de un ideal), decidir cuánto vale

$$SQ(P, Z) = \#\{p \in Z \mid P(p) > 0\} - \#\{p \in Z \mid P(p) < 0\}\}$$

La Sturm Query será de vital importancia en la sección 2.3

Nuestra fución `SQ` recibe un polinomio, una base de un cociente y una Base de Gröbner para el ideal al que corresponde el cociente.

Ejemplo 2.16

```
> SQ(1,b,i);
8
```

Esta Sturm Query es $O(N^4)$, donde N es la dimensión $\dim_{\mathbb{R}}(A)$.

2.2.2 Proyección racional univariada

Explicaremos, ahora, otro enfoque utilizado para atacar el problema de contar la cantidad de raíces reales de ideales de dimensión cero, que se remonta a ideas de Kronecker. En este caso, para acelerar el procedimiento, utilizaremos números aleatorios. Nuevamente, aprovecharemos la estructura de $\mathbb{R}$–espacio vectorial de nuestro cociente A.

Sea p un polinomio que separa los puntos de nuestra $\mathbf{V}(I)$. Esto quiere decir que $x, y \in \mathbf{V}(I), x \neq y \Rightarrow p(x) \neq p(y)$. Supongamos que $\mathbf{V}(I) = \{p_1, \ldots, p_r\}$, cada uno con multiplicidad m_i. Entonces, $\sum_{i=1}^{r} m_i = N = \dim_{\mathbb{R}}(A)$.

Si consideramos polinomios lineales de la forma $a(x) = \alpha_1 x_1 + \cdots + \alpha_n x_n$ con $(\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$, entonces se puede demostrar que con coeficientes aleatorios, el polinomio separa puntos con probabilidad 1. En la computadora no podemos trabajar con números reales, pero trabajando con los racionales también tenemos buena probabilidad de encontrar un polinomio que separe puntos. Además, se sabe que $\exists j, 0 \leq j \leq (n-1)\binom{r}{2}$ tal que $a(j)(x) = x_1 + jx_2 + \cdots + j^{n-1}x_n$ separa puntos. Entonces, podríamos probar todos los posibles polinomios generados a partir de los distintos j, pero esto sería muy costoso. En cambio, podemos generar un polinomio $a(x)$ utilizando coeficientes pseudoaleatorios provistos por SINGULAR. Esto fue lo que implementamos en la función `rndlnrpoly`.

Ejemplo 2.17

```
> rndlnrpoly();
380035x+636642y+240875z
```

Si llamamos χ_a al polinomio característico de la transformación lineal m_a, tenemos que, cuando a separa los puntos de I, el número de raíces reales de χ_a es el mismo que el de raíces reales de su reducido, y resulta ser el número de puntos reales en $\mathbf{V}(I)$. Esto nos muestra que una forma de resolver nuestro problema es calcular el número de raíces reales de este χ_a. Para ello, tenemos dos opciones. Podemos calcular la matriz de m_a en una base B de A. Luego, calculamos el polinomio característico de esa matriz, y contamos la cantidad de raíces reales de ese polinomio. Todo eso se puede hacer con las funciones que ya tenemos implementadas, pero existe otra posibilidad: se puede demostrar que

$$\sum_{i=1}^{r} m_i a(p_i)^k = \text{traza}(m_{a^k}) \quad k = 0, \ldots, N-1$$

Además, tenemos que

$$\chi_a(t) = \det(tI_n - m_a) = \prod_{i=1}^{r}(t - a(p_i))^{m_i}$$

A partir de estas sumas de potencias, empleando las fórmulas de Newton, se pueden reconstruir los coeficientes del $\chi_a(t)$. A saber, tenemos que

$$(N-l)a_l = -\sum_{j>l}^{N} a_j S_{j-l}$$

donde $S_k = \sum_{i=1}^{N} t_i^k$.

Implementamos las propiedades mencionadas más arriba en varias funciones. `powersums` toma un polinomio y calcular las sumas de potencias utilizando las trazas antes explicadas.

Ejemplo 2.18

```
> list l = powersums(380035x+636642y+240875z,b,i);
> l;
[1]:
   0
[2]:
   9220281204746
[3]:
   0
[4]:
   2051519390675875869477 6025
[5]:
   0
[6]:
   92909654529649883173700359632750689029/2
[7]:
   0
[8]:
   42121675310270788189424129727177390151483908873 6817/4
```

En el ejemplo de arriba, utilizamos el mismo ideal, y la misma base que en el ejemplo 2.15. Con las sumas de potencias, utilizamos la función `symmfunc`, que nos da las funciones simétricas de las raíces del polinomio, usando las fórmulas de Newton.

Ejemplo 2.19

```
> list s = symmfunc(l);
> s;
[1]:
   1
[2]:
   0
[3]:
   -4610140602373
[4]:
   0
[5]:
   219915988405374158916862 33/4
[6]:
   0
[7]:
   -428180038775247336115546872870408774
[8]:
   0
[9]:
   2069850570566494500126804908252247581436268644
```

Ahora armamos un polinomio con estos coeficientes, y calculamos la cantidad de raíces reales que tiene

Ejemplo 2.20

```
> poly h = polynomial(s);
> stha(h);
8
```

Para simplificar al usuario la obtención del polinomio característico que describimos más arriba, armamos la función `rndchpoly`. Esta realiza todos los pasos descriptos. Como mencionamos oportunamente, el polinomio lineal utilizado en los cálculos es generado con coeficientes pseudo-aleatorios. Estos polinomios tienen una probabilidad muy alta de separar los puntos de $\mathbf{V}(I)$. Por eso, creemos que pueden resultar muy útiles a la hora de trabajar en conjeturas. La ventaja que traen es que permiten calcular la cantidad de raíces reales de un ideal en forma más rápida que empleando `bil` y `symmsig`.

Ejemplo 2.21

```
> rndchpoly(b,i);
**********************************************
* WARNING: This polynomial was obtained using  *
```

```
* pseudorandom numbers. If you want to verify  *
* the result, please use the command            *
*                                               *
* verify(p,b,i)                                 *
*                                               *
* where p is the polynomial, b is the monomial *
* bases used, and i the grobner basis of the    *
* ideal                                         *
************************************************
x8-1946227338346x6+1341689659017014022929546x4-
  3863537580406791222194990769635154 34x2+
  39440448444863095864418999087260012246397750161
> verify(_,b,i);
Verification successful
1
```

Como podemos ver en el ejemplo anterior, implementamos el comando `verify`, que calcula la cantidad de raíces complejas en $\mathbf{V}(I)$ (el rango de la matriz de φ_1) y compara eso con la cantidad de raíces simples (reales y complejas) del polinomio obtenido. Este cálculo es $O(N^4)$, es decir que es tan costoso como calcular una Sturm Query, y por eso se incluye sólo para realizar verificaciones. Nuestro algoritmo, utilizando el polinomio lineal pseudoaleatorio, es $N^{O(1)}$.

2.3 Determinación de signos

El otro problema que encaramos en este trabajo fue, dado un ideal de dimensión cero $I \subseteq \mathbb{R}[X_1, \ldots, X_n]$, un conjunto de polinomios $P_1, \ldots, P_s$ y una lista σ de s signos 0, $+$ o $-$, decir cuál es la cantidad de puntos p de $\mathbf{V}(I) \cap \mathbb{R}^n$ que verifican $\text{signo}(P_1(p)) = \sigma_i, \ldots, \text{signo}(P_s(p)) = \sigma_s$.

Un ejemplo en el que este problema surge es, dado un ideal I y una circunferencia en el plano dada por $(x-a)^2 + (y-b)^2 = r^2$, puede interesar calcular cuántos puntos de $\mathbf{V}(I)$ están dentro de la circunferencia. La ecuación de ésta se puede reescribir como $x^2+y^2-2ax-2by+a^2-b^2-r^2 = 0$. El lado izquierdo es un polinomio en x e y. Llamémoslo P. Si $P(v) = 0$, entonces v está sobre la circunferencia. Si $P(v) < 0$, v está dentro de la circunferencia. Entonces, el problema se puede traducir a: ¿en cuántas raíces de I el resultado de evaluar P es negativo? Este ejemplo se puede generalizar a cualquier polígono: cada lado está determinado por una recta, que se representa con un polinomio (de grado 1) igualado a 0.

Si hacemos $P_i = x_i$, es decir, tomamos como polinomios las distintas coordenadas de los puntos, podemos calcular la cantidad de raíces en el primer ortante[1], es decir, con raíces reales con todas sus coordenadas positivas, de un ideal de polinomios.

Comenzaremos analizando el caso $s = 1$, es decir, un sólo polinomio P y un ideal I de dimensión cero. Ya hemos visto que $\text{sig}(\varphi_h) = \#\{p \mid p \in \mathbf{V}(I) \cap \mathbb{R}^n \wedge h(p) > 0\} - \#\{p \mid p \in \mathbf{V}(I) \cap \mathbb{R}^n \wedge h(p) < 0\}$. Notamos $(P > 0)$, $(P < 0)$, $(P = 0)$ a los subconjuntos de $\mathbf{V}(I)$ en los que evaluar P resulta mayor, menor o igual a 0, respectivamente. Además, notaremos $SQ(P, \mathbf{V}(I))$ a $\text{sig}(\varphi_P)$. Por último, haremos $c(Z) = \#Z$, con Z un conjunto finito (posiblemente vacío). Entonces, es fácil ver que

$$\begin{aligned} SQ(1, \mathbf{V}(I)) &= c(P = 0) + c(P > 0) + c(P < 0) \\ SQ(P, \mathbf{V}(I)) &= c(P > 0) - c(P < 0) \\ SQ(P^2, \mathbf{V}(I)) &= c(P > 0) + c(P < 0) \end{aligned}$$

Esto se puede representar como el siguiente sistema lineal

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} c(P = 0) \\ c(P > 0) \\ c(P < 0) \end{pmatrix} = \begin{pmatrix} SQ(1, \mathbf{V}(I)) \\ SQ(P, \mathbf{V}(I)) \\ SQ(P^2, \mathbf{V}(I)) \end{pmatrix}$$

El vector de la derecha se puede calcular con las funciones que ya tenemos implementadas en la biblioteca `mrrc.lib`. Además, la matriz de coeficientes es inversible, por lo que el sistema está determinado y se puede resolver sin mayores dificultades.

Ahora queremos extender el problema a un conjunto $\mathcal{P} = \{P_1, \ldots, P_s\}$. Es decir, queremos calcular la cantidad de ceros de un ideal I en los que el resultado de evaluar cada P_i tiene signo predeterminado σ_i. Por ejemplo, todos positivos. Para ello, necesitaremos nueva notación. Con α^i notaremos el i-ésimo exponente del conjunto $\{0, 1, 2\}^s$, ordenados lexicográficamente, pero de derecha a izquierda (p.ej. $(0, 0, \ldots, 1, 0) > (0, 0, \ldots, 0, 1)$). Análogamente, usaremos σ^i para el i-ésimo vector de signos del conjunto $\{0, 1, -1\}^s$, con el mismo orden de antes (con $-1 > 1$). Si σ es un vector de signos y $\alpha \in \{0, 1, 2\}^s$ es un exponente, definimos

$$\sigma^\alpha = \sigma_1^{\alpha_1} \times \cdots \times \sigma_2^{\alpha_s}$$

Además, notaremos $c(\sigma) = c(P_1 = \sigma_1, \ldots, P_s = \sigma_s)$. Si α es un exponente, notaremos $\mathcal{P}^\alpha = P_1^{\alpha_1} \cdots P_s^{\alpha_s}$.

[1] Generalización de octante

Con esta nueva notación, podemos armar un nuevo sistema:

$$M_s \begin{pmatrix} c(\sigma^1) \\ \vdots \\ c(\sigma^{3^s}) \end{pmatrix} = \begin{pmatrix} SQ((P)^{\alpha^1}, \mathbf{V}(I)) \\ \vdots \\ SQ((P)^{\alpha^{3^s}}, \mathbf{V}(I)) \end{pmatrix}$$

donde

$$(M_s)_{ij} = (\sigma^j)^{(\alpha^i)}$$

Nuevamente, los valores del vector de la derecha se pueden calcular con las funciones que ya tenemos implementadas. Para calcular la matriz de $3^s \times 3^s$ M_s, podemos utilizar su definición, e ir evaluando cada signo en cada exponente. Por fortuna, esta matriz se puede calcular de otra manera. Si hacemos

$$M_1 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 1 & 1 \end{pmatrix}$$

Entonces nuestra M_s está definida por

$$M_s = M_{s-1} \otimes M_1$$

donde $\otimes$ representa el producto tensorial de matrices. Este producto tensorial está implementado en la función `tensor` de SINGULAR.

De esta manera, nos ahorramos muchísimo tiempo de ejecución: el tiempo de cálculo de M_s pasa a ser despreciable. Sin embargo, tenemos un número exponencial en s de Sturm Queries para calcular. Esto puede no ser necesario. La cantidad de $c(\sigma)$ que pueden ser distintas de cero está acotada por la cantidad de raíces del ideal I. Entonces, sería bueno optimizar estos cálculos.

Si resolvemos el sistema para P_1, tendremos ciertos $c(\sigma)$ iguales a 0. Supongamos que no hay raíces de I en los que evaluar P dé negativo. Entonces, sólo puede dar positivo o 0. Entonces, la misma información que teníamos en el sistema de 3×3 se puede resumir en el siguiente sistema de 2×2

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c(P=0) \\ c(P>0) \end{pmatrix} = \begin{pmatrix} SQ(1, \mathbf{V}(I)) \\ SQ(P, \mathbf{V}(I)) \end{pmatrix}$$

Si m es la cantidad de incógnitas que no se anulan al resolver el sistema de 3×3, el nuevo sistema se obtiene eliminando del vector de incógnitas aquellas que sabemos que se anulan. Del vector de Sturm Queries, nos quedamos con aquellas que corresponden a los m primeros exponentes de la lista $\{0, 1, 2\}$. La matriz que nos queda dependerá de qué incógnitas no se anulan (ver [BPR/03]).

Algoritmo 3 Determinación de Signos

Entradas: Un conjunto de polinomios $\{P_1, \ldots, P_s\}$ y una base de Gröbner de un ideal I.

Salida: Un vector C con las condiciones de signos realizables por los P_i en $\mathbf{V}(I)$.

$M \leftarrow (1)$

$i \leftarrow 1$

$M' \leftarrow \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 1 & 1 \end{pmatrix}$

mientras $i < s$ **hacer**

$sq \leftarrow \begin{pmatrix} SQ(1, \mathbf{V}(I)) \\ SQ(P_i, \mathbf{V}(I)) \\ SQ(P_i^2, \mathbf{V}(I)) \end{pmatrix}$

$M_i \leftarrow (M')^{-1}c$

si Ninguna coordenada de c es 0 **entonces**

$signos_i \leftarrow \{0, 1, -1\}$

$exp_i = \{0, 1, 2\}$

si no, si Una sola coordenada de c es 0 **entonces**

$exp_i = \{0, 1\}$

si $c(P = 0) \neq 0 \wedge c(P > 0) = 0$ **entonces**

$signos_i \leftarrow \{0, 1\}$

$M_i \leftarrow \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$

si no, si $c(P = 0) \neq 0 \wedge c(P < 0) = 0$ **entonces**

$signos_i \leftarrow \{0, -1\}$

$M_i \leftarrow \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}$

si no, si $c(P > 0) \neq 0 \wedge c(P < 0) = 0$ **entonces**

$signos_i \leftarrow \{1, -1\}$

$M_i \leftarrow \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

fin si

si no, si Hay dos coordenadas de c que son 0 **entonces**

$signos_i \leftarrow$ el signo que corresponde a la coordenada distinta de 0.

$M_i \leftarrow (1)$

$exp_i = \{0\}$

fin si

$M \leftarrow M_i \otimes M.$

$Signos \leftarrow Signos \times signos_i$

$Exponentes \leftarrow Exponentes \times exp_i$

fin mientras

$SQ \leftarrow \begin{pmatrix} SQ(\mathcal{P}^{Exponentes_1}\mathbf{V}(I)) \\ \vdots \\ SQ(\mathcal{P}^{Exponentes_k}\mathbf{V}(I)) \end{pmatrix}$

$C \leftarrow M^{-1}SQ$

Si llamamos M_i' a la matriz que resulta de realizar este mismo procedimiento con el polinomio P_i, veremos que:

- M_i' es inversible.
- Cada sistema descarta los valores que ya sabemos que son cero para cada polinomio.
- Los vectores de los lados derechos de los sistemas tiene a lo sumo la misma cantidad de elementos. En general, tendrán menos, y los polinomios con los que habrá que trabajar tendrán grado más chico.

Una vez que hicimos esto para todos los polinomios, construimos la matriz M', dada por

$$M' = M_1' \otimes \cdots \otimes M_s'$$

Por las propiedades del producto tensorial, esta matriz resulta inversible.

Además armamos el vector de incógnitas de la siguiente manera. Supongamos que para P_1, tenemos $c(P = 0) = 0$. Entonces, nos quedamos con los signos 1 y -1. Supongamos que de P_2 nos quedamos con 0 y 1. Para combinar P_1 y P_2, los σ que utilizaremos serán $(1,0),(-1,0),(1,1),(-1,1)$. Es decir, una especie de producto cartesiano de listas. El mismo procedimiento se realiza con los exponentes.

Al hacer esto con cada P_i, tendremos un sistema potencialmente más pequeño que el original (o naïf), que nos permitirá obtener la misma información, pero calculando menos Sturm Queries. Este procedimiento, descripto en el algoritmo 3 está implementado en la función `signcnd`. Esta función recibe un ideal $\mathcal{P}$ y un ideal I, y devuelve las condiciones de signos realizables por los elementos de $\mathcal{P}$ al ser evaluados en los ceros de I.

La complejidad del procedimiento de determinación de signos es $sN^{O(1)}$, si se van descartando los signos que no ocurren. Aquí, s es el número de polinomios de P, y N es la dimensión del cociente entre el anillo base y el ideal I.

Para lograr una verdadera mejora en el tiempo de ejecución, no sólo es necesario ir descartando filas del sistema lineal. También es necesario almacenar las Sturm Queries calculadas con cada polinomio, ya que estas vuelven a ser útiles en el sistema final, y calcularlas nuevamente haría que nuestro algoritmo fuera peor que la implementación naïf.

Veamos un ejemplo. Supongamos que estamos trabajando con el ideal i del ejemplo 2.15. Queremos calcular el número de raíces reales que este ideal tiene en cada octante. Entonces, hacemos

Ejemplo 2.22

```
> signcnd(ideal(x,y,z),i);
[1]:
   [1]:
      [1]:
         1
      [2]:
         1
      [3]:
         1
   [2]:
      [1]:
         1
      [2]:
         -1
      [3]:
         1
   [3]:
      [1]:
         -1
      [2]:
         1
      [3]:
         -1
   [4]:
      [1]:
         -1
      [2]:
         -1
      [3]:
         -1
[2]:
   [1]:
      2
   [2]:
      2
   [3]:
      2
   [4]:
      2
```

Analicemos la salida de esta función. El resultado está compuesto por

dos listas. Llamémoslas *condiciones* y *cantidades*. Ambas tiene la misma cantidad de elementos. Los elementos de la primera indican las condiciones de signos realizables por los polinomios del primer ideal en los ceros del segundo. Cada uno de los elementos de *condiciones* es una lista que tiene un elemento por cada polinomio del primer ideal. Estas listas indican los signos que toma cada polinomio en la condición. Por ejemplo, el segundo elemento de *condiciones* es la lista $(1, -1, 1)$. Esto indica que la condición $x > 0$, $y < 0$ y $z > 0$ es realizable en los ceros de i. El segundo elemento de *cantidades* indica cuántos elementos de $\mathbf{V}(I)$ satisfacen esta condición. El resto de los elementos se interpretan de la misma manera. En total, hay cuatro condiciones realizables, con dos ceros en cada una. Eso da un total de 8 raíces, consistente con nuestros cálculos previos.

Veamos otro ejemplo que nos da más información sobre las raíces de i.

Ejemplo 2.23

```
signcnd(ideal(x^2+y^2+z^2-4),i);
[1]:
   [1]:
      [1]:
         0
[2]:
   [1]:
      8
> signcnd(ideal((x-1)^2+y^2+z^2-4),i);
[1]:
   [1]:
      [1]:
         1
   [2]:
      [1]:
         -1
[2]:
   [1]:
      4
   [2]:
      4
```

Aquí vemos que todas las raíces reales de i están ubicadas sobre la bola de radio 2. Si desplazamos esta bola 1 unidad en el eje x, tenemos la mitad de las raíces adentro y la mitad de las raíces afuera.

Para brindar soporte a la investigación en conjeturas como la de Kushni-

renko, implementamos el comando `fstoct`, que recibe un ideal y devuelve el número de raíces reales que tiene con todas sus coordenadas positivas.

Con fines comparativos, dejamos en la biblioteca la función `anyoct`. Esta toma un ideal y una lista de signos (0, 1 o −1), y devuelve la cantidad de raíces del ideal que satisfacen la condición expresada por la lista. Esta función utiliza la manera naïf de resolver el problema, y es notablemente más lenta que signcnd.

Capítulo 3

Conclusiones

En este trabajo abordamos dos problemas de vital importancia en la Geometría Algebraica: contar la cantidad de raíces reales de un sistema de ecuaciones polinomiales, y determinar la ubicación de esas raíces. Ambos problemas surgen en una gran variedad de situaciones, y lamentablemente, no tienen solución eficiente. Afortunadamente, sí existen soluciones efectivas, costosas, pero que pudimos implementar satisfactoriamente.

Además, estos problemas podrían, a priori, no ser pasibles de resolución con métodos algorítmicos. Después de todo, el problema de calcular las raíces de un polinomio no tiene, salvo en casos particulares, solución algorítmica.

Los procedimientos implementados pueden ser útiles para resolver problemas concretos con un número moderado de variables. Además, pueden servir para disparar métodos de aproximación de raíces reales, cuando no se cuenta con información adicional sobre las mismas. Por ejemplo, se puede utilizar el comando `signcnd` para "acorralar" soluciones utilizando bolas y rectas. Es importante destacar que estas bibliotecas serán puestas a disposición de la comunidad científica, incorporándolas a la distribución de SINGULAR.

El desarrollo del presente trabajo nos enfrentó con las dificultades de traducir los algoritmos "matemáticos" encontrados en la literatura, en algoritmos "computacionales". Los distintos trabajos consultados toman un enfoque muy distendido a la hora de decir qué es un algoritmo. Esto se ve manifestado en "instrucciones" de altísimo nivel, que tienen una complejidad no despreciable con respecto a la de todo el algoritmo. Eso hizo que no pudiéramos implementar algunas optimizaciones: los manejos de índices hacían muy complicado el problema.

SINGULAR es un programa bastante bueno para el trabajo simbólico con polinomios. Sin embargo, por su modelo de desarrollo, no ha sido exhaustivamente depurado, y presenta algunos problemas. En particular, en el manejo de la memoria.

3.1 Trabajo a futuro

Algunos temas que se pueden seguir elaborando en un futuro inmediato son:

- **Uso de órdenes monomiales con peso.** Otra manera de calcular la cantidad de raíces reales de un sistema de ecuaciones polinomiales, bajo circunstancias especiales que involucran órdenes con peso, consiste el tomar el coeficiente principal del Jacobiano de ciertas transformaciones. Encontrar un peso que sirva para esto es un problema de combinatoria, pero se puede trabajar suponiendo dado un peso.

- **Optimización del algoritmo de determinación de signos.** Existe una versión del algoritmo que permite calcular aún menos Sturm-Queries, pero requiere un manejo complicado y cuidadoso de índices. Para detalles, ver [BPR/03].

- **Adecuación de la documentación de las bibliotecas desarrolladas al estándar de** SINGULAR . Los autores de SINGULAR han definido una serie de lineamientos a los que debe ajustarse la documentación de las bibliotecas que forman parte de la distribución del programa, a fin de poder generar automáticamente la documentación de las mismas.

- **Manejar polinomios sin expandir.** Si en SINGULAR ingresamos $(x-1)^1000$, el programa lo expande para representarlo internamente. En muchas situaciones, por ejemplo si deseamos hacer el desplazamiento $x \mapsto x+1$, puede resulta útil guardar el polinomio en forma compacta.

Apéndice A

Breve introducción al uso de SINGULAR

El programa SINGULAR puede bajarse libremente de:

http://www.singular.uni-kl.de

Se instala y se ejecuta desde la línea de comandos. En forma opcional, pero recomendable, se lo puede ejecutar dentro de algún editor de textos que lo permita. El programa viene con archivos para facilitar su integración con Emacs.

El programa mostrará el símbolo > cuando esté listo para recibir entrada. Para terminar una secuencia de entrada, se debe ingresar ;.

Antes de poder hacer cualquier cosa útil, debemos declarar el anillo en el que vamos a trabajar. La declaración debe incluir el cuerpo de coeficientes, los nombres de las variables y el orden monomial que se utilizará.

Por ejemplo,

Ejemplo A.1

```
> ring r = 0,(x,y,z),lp;
> r;
//   characteristic : 0
//   number of vars : 3
//        block   1 : ordering lp
//                  : names    x y z
//        block   2 : ordering C
> ring s = 13,x,dp;
> s;
//   characteristic : 13
//   number of vars : 1
//        block   1 : ordering dp
```

```
//                   : names    x
//         block   2 : ordering C
> ring t = (0,a,b),(x,y),Dp;
> t;
t;
//   characteristic : 0
//   2 parameter    : a b
//   minpoly        : 0
//   number of vars : 2
//         block   1 : ordering Dp
//                   : names    x y
//         block   2 : ordering C
```

Los anillos `r`, `s` y `t` tienen los órdenes lexicográfico, grevlex y deglex, respectivamente. El anillo `t` permitirá trabajar con los coeficientes paramétricos `a` y `b`.

Las definiciones de ideales, polinomios, etc. son locales a un anillo. Para cambiar de anillos, se utiliza el comando `setring`.

Ejemplo A.2

```
> setring(r);
> poly p = x^2+3*y^5;
> p;
x2+3y5
> setring(s);
> p;
   ? error occurred in STDIN line 52: '> p;'
   skipping text from ';'
```

En este ejemplo, vemos que SINGULAR utiliza una notación abreviada para los polinomios. $x^2 + 3 * y^5$ se puede escribir también como `x2+3y5`. Además, comprobamos que las definiciones de objetos dentro de un anillo son locales a él.

Una vez que tenemos definido un anillo, lo más común será definir ideales, y tomar bases de grobner para ellos.

Ejemplo A.3

```
> setring(r);
> ideal i = x4-2zy,y2+zx-2xy,2z3;
> i = std(i);
> i;
i[1]=z3
```

```
i[2]=2y5z+3y4z2-64y2z2+32yz3
i[3]=2y6+3y5z+3y4z2-64y3z+32y2z2
i[4]=2xy-xz-y2
i[5]=x3z2+2x2y3+x2y2z-16y2z+8yz2
i[6]=x4-2yz
```

El ideal obtenido es una Base de Gröbner para el ideal `i`, utilizando el orden lexicográfico.

Otro comando útil para nuestros propósitos es `vdim`. Este nos da la dimensión del cociente entre el anillo en el que estamos trabajando y un ideal. Esta dimensión será correcta sólo si le pasamos como parámetro una Base de Gröbner para un ideal de dimensión cero.

Ejemplo A.4

```
> vdim(i);
24
```

Además de ideales y polinomios, SINGULAR permite trabajar con listas, matrices y vectores. Las listas son la manera más cómoda que brinda el programa de manejar memoria dinámica.

Las funciones `vdim` y `std` forman parte del kernel de SINGULAR. El programa trae muchas otras funciones en bibliotecas estándar. Por ejemplo, `linalg.lib` incluye rutinas para invertir matrices, calcular determinantes, polinomios caracterísiticos, rango, etc. Para acceder a una biblioteca, esta debe estar en el directorio LIB de la distribución. El comando para utilizarlas es:

Ejemplo A.5

```
> LIB "linalg.lib";
// ** loaded linalg.lib (1.10.2.12,2002/04/12)
// ** loaded general.lib (1.38.2.7,2002/04/12)
// ** loaded poly.lib (1.33.2.5,2002/04/09)
// ** loaded inout.lib (1.21.2.3,2002/02/20)
// ** loaded elim.lib (1.14.2.2,2002/02/20)
// ** loaded ring.lib (1.17.2.1,2002/02/20)
// ** loaded matrix.lib (1.26.2.1,2002/02/20)
// ** loaded random.lib (1.16.2.1,2002/02/20)
```

Estas bibliotecas contienen definiciones de procedimientos escritas en un lenguaje de programación muy similar a C. Las tres bibliotecas de este trabajo fueron implementadas en él. Este lenguaje brinda las estructuras básicas de control (for, while, if), además de permitir trabajar con todos los tipos de SINGULAR. Veamos un ejemplo de procedimiento:

Ejemplo A.6

```
proc reverse(list l) // Returns l reversed
{
  int i;
  list result;

  for (i = 1;i <= size(l);i++) {
    result = list(l[i]) + result;
  }
  return (result);
}
```

El procedimiento `reverse`, de la biblioteca `urrc.lib`, da vuelta los elementos de una lista. Aquí vemos que no se declara el tipo de retorno de una función. Simplemente se devuelve lo que corresponde.

El chequeo de tipos se realiza en tiempo de ejecución. Esto es, en ocasiones, problemático al desarrollar: no se revisa que las instrucciones sean correctas. De hecho, ni siquiera se revisa que todos los identificadores que aparecen en una función hayan sido definidos previamente. Sólo se hace si en algún momento se ejecuta la instrucción en la que aparecen. Esto es muy poco amigable si de detectar errores se trata.

Las funciones del kernel, además de las funciones de las bibliotecas estándar, vienen con importante documentación. Para acceder a ella desde SINGULAR, existe el comando `help`.

Ejemplo A.7

```
> help vdim;
```

abrirá una ventana de ayuda. La manera en que se muestra esta ayuda es configurable, y depende en parte del sistema operativo que se esté utilizando. Incluso, se puede hacer que la ayuda aparezca en el mismo entorno de SINGULAR. `> help help;` mostrará cómo hacerlo.

Apéndice B

Resumen de los comandos implementados

B.1 `urrc.lib` - Univariate Real Roots Counting

Comando	Comportamiento	Pág.
`isuni(poly p)`	Devuelve una variable si el polinomio es univariado. Si no, devuelve 0.	11
`whichvariable(poly p)`	Recibe un monomio y devuelve una variable si es univariado. Si no, devuelve 0.	11
`varsigns(list l)`	Recibe una lista de números y devuelve la cantidad de variaciones de signos que aparecen en ella.	10
`boufou(poly p,number a,number b)`	Devuelve una cota superior, congruente módulo 2, para la cantidad de raíces del polinomio univariado `p` en el intervalo `(a,b]`, con multiplicidad	12
`descartes(poly p)`	Devuelve una cota, congruente módulo 2, para la cantidad de raíces positivas del polinomio univariado `p`.	10

`descartest(poly p)`	Devuelve una cota para la cantidad de raíces reales del polinomio univariado `p`.	10
`allrealst(poly p)`	Devuelve verdadero si, y sólo si, todas las raíces del polinomio univariado `p` son reales.	15
`maxabs(poly p)`	Devuelve una cota para el valor absoluto máximo de las raíces reales del polinomio univariado `p`	15
`sturm(poly p,number a,number b)`	Devuelve la cantidad de raíces reales del polinomio univariado `p` en el intervalo `(a,b]`, contadas sin multiplicidad.	15.
`sturmseq(poly p)`	Devuelve una secuencia de Sturm para el polinomio univariado `p`.	14
`allreal(poly p)`	Devuelve verdadero si, y sólo si, el polinomio univariado `p` tiene todas sus raíces reales.	18
`stha(poly p)`	Devuelve la cantidad de raíces reales del polinomio univariado `p`	17
`sthaseq(poly p)`	Devuelve una secuencia de Sturm–Habicth para `p`	17

B.2 `mmrc.lib` - Multivariate Real Roots Counting

Comando	Comportamiento	Pág.
`symmsig(matrix m)`	Devuelve la signatura de la matriz simétrica `m`	21
`SQ(poly h,ideal B,ideal I)`	Devuelve la Sturm-Query del polinomio `h`, usando la base `B` para el cociente entre el anillo base y el ideal `I`	21

`bil(poly h,ideal B,ideal I)`	Devuelve la matriz en la base `B` asociada a la forma bilineal simétrica $\varphi_h : A \times A \to \mathbb{R}$, dada por $([f],[g]) \mapsto \text{traza}(m_{hfg})$. A es el cociente entre el anillo base y el ideal `I`, y `B` es una base para ese cociente.	20.
`matrace(poly f,ideal B,ideal I)`	Calcula la traza de la matriz en la base `B` de la transformación lineal $m_f : A \to A$, donde A es el cociente entre el anillo base y el ideal `I`. `B` es una base de A.	20
`mat(poly f,ideal B,ideal I)`	Calcula la matriz en la base `B` de la transformación lineal $m_f : A \to A$, donde A es el cociente entre el anillo base y el ideal `I`. `B` es una base de A.	19
`rndchpoly(ideal B,ideal I)`	Devuelve un el polinomio característico correspondiente a la transformación lineal m_a, para un polinomio lineal pseudoaleatorio a que, se espera, separa los puntos del ideal `I`. `B` debe ser una base de monomios del cociente entre el anillo base y el ideal `I`.	24
`verify(poly p,ideal B,ideal i)`	Devuelve verdadero si, y sólo si, el polinomio `p` separa los puntos del ideal `I`. `B` debe ser una base de monomios del cociente entre el anillo base y el ideal `I`.	25
`rndlnrpoly()`	Devuelve un polinomio lineal en cada variable del anillo con coeficientes pseudoaleatorios.	22
`powersums(poly f,ideal B,ideal I)`	Calcula las sumas de potencias de los resultados de evaluar el polinomio `f` en los puntos de `I` a través de las trazas de m_{f^k}.	23

`symmfunc(list S)`	Toma una lista de sumas de potencias y calcula las funciones simétricas utilizando las fórmulas de Newton.	23
`qbase(ideal i)`	Devuelve una base de monomios del cociente entre el anillo base y el ideal `i`. Esta base es un ideal ordenado en forma descendente, de acuerdo al orden monomial en el que se está trabajando.	19

B.3 `signdet.lib` - Determinación de Signos

Comando	Comportamiento	Pág.
`fstoct(ideal I)`	Devuelve la cantidad de raíces reales del ideal `I` con todas sus coordenadas positivas	32
`signcnd(ideal P,ideal I)`	Devuelve una lista con las condiciones de signos satisfechas por los polinomios del ideal `P` al ser evaluados las raíces reales del ideal `I`	29
`anyoct(ideal i,list quad)`	Devuelve la cantidad de raíces reales del ideal `i` en el ortante determinado por la lista de signos (0, 1 o -1) `quad`.	32

Bibliografía

[BPR/03] Saugata Basu, Richard Pollack, y Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*, volumen 10 de *Algorithms and Computation in Mathematics*. Springer-Verlag Berlin Heidelberg, 2003.

[BR/90] Riccardo Benedetti y Jean-Jacques Risler. *Real algebraic and semi-algebraic sets*. Hermann, 1990.

[BWZ/03] C. Böm, M Wang, y W. Ziller. A variational approach for compact homogeneous einstein manifolds. por aparecer en GAFA, 2003.

[CLO/97] David Cox, John Little, y Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate Texts in Mathematics. Springer-Verlag New York, 2da edición, 1997.

[CLO/98] David Cox, John Little, y Donal O'Shea. *Using Algebraic Geometry*, volumen 185 de *Graduate Texts in Mathematics*. Springer-Verlag New York, 1998.

[Dat/03] Ruchira S. Datta. Using computer algebra to compute Nash equilibria. En *Proceedings ISSAC 2003*, páginas 74–79. ACM Press, 2003.

[Gat/01] Karin Gatermann. Counting stable solutions of sparse polynomial systems in chemistry. En E. Green, S Hosten, R. Laubenbacher, y V. Powers, editores, *Symbolic Computation: Solving Equations in Algebra, Geometry, and Engineering*, volumen 286 de *Contemporary Math*, páginas 53–69, Providence, Rhode Island, 2001. AMS.

[GMR/97] Michel Goossens, Frank Mittelbach, y Sebastian Rahtz. *The LaTeX Graphics Companion.* Addison-Wesley Longman, Inc, 1997.

[GMS/94] Michel Goossens, Frank Mittelbach, y Alexander Samarin. *The LaTeX Companion.* Addison-Wesley, 2da edición, 1994.

[GP/02] Gert-Martin Greuel y Gerhard Pfister. *A Singular Introduction to Commutative Algebra.* Springer-Verlag, 2002.

[GPS/01] Gert-Martin Greuel, Gerhard Pfister, y Hans Schönemann. SINGULAR 2.0.3. A Computer Algebra System for Polynomial Computations, Centre for Computer Algebra, University of Kaiserslautern, 2001. `http://www.singular.uni-kl.de`.

[GVRR/99] Laureano González-Vega, Fabrice Rouillier, y Marie-Françoise Roy. Symbolic recipes for real solutions. En Arjeh M. Cohen, Hans Cuypers, y Hans Sterk, editores, *Some Tapas of Computer Algebra*, volumen 4 de *Algorithms and Computation in Mathematics*, capítulo 2, páginas 34–65. Springer-Verlag Berlin Heidelberg, 1999.

[GVT/97] Laureano González-Vega y Guadalupe Trujillo. Multivariate Sturm–Habicht sequences: Real root counting on n-rectangles and triangles. *Revista Matemática Complutense*, 10(Special Issue), 1997.

[Haa/02] Bertrand Haas. A simple counter-example to Kushnirenko's conjecture. *Beiträge zue Algebra und Geometrie*, 43(1):1–8, 2002.

[Kho/91] Askold Geogievich Khovanski. *Fewnomials.* AMS Press, Providence, Rhode Island, 1991.

[Lam/85] Leslie Lamport. *LaTeX— A Document Preparation System — User's Guide and Reference Manual.* Addison-Wesley, 2da edición, 1985.

[Mou/93] Bernard Mourrain. The 40 generic positions of a parallel robot. En Bernard Mourrain, editor, *Proc. Intern. Symp. on Symbolic and Algebraic Computation*, páginas 173–182, Kiev, Ucrania, Julio 1993. ACM Press.

[RLW/03] J. Maurice Rojas, Tien-Yien Li, y Xiaoshen Wang. Counting real connected components of trinomials curve intersections and m-nomial hypersurfaces. *Discerte and Computational Geometry*, 30(2):379–414, 2003.

[RS/98] J. Rosenthal y Frank Sottile. Some remarks on real and complex output feedback. *Sys. and Control Lett.*, (33):73–80, 1998.

[Wil/94] James H. Wilkinson. *Rounding Errors in Algebraic Processes.* Dover, 1994.

Ficha Técnica

El presente trabajo está hecho íntegramente en LaTeX. Utilizamos los paquetes `babel`, algorithm, algorithmic, doublestroke, amsmath, theorem, xspace, longtable y color. Además de la documentación de cada paquete, consultamos los siguientes libros: [GMS/94], [Lam/85] y [GMR/97]. Los paquetes algorithm y algorithmic, así como el estilo de bibliografía fueron adaptados al español y a los requerimientos del Departamento de Computación por el autor. Todo fue realizado con el editor Emacs.