# Matemáticas Elementales con Sage

J. L. Tábara (jltabara@gmail.com)

9 de abril de 2015

# Índice general

**1. Aritmética básica** **3**
1.1. Operaciones con enteros . . . . . 3
1.2. Operaciones con números racionales y reales . . . . . 4
1.3. La ayuda en Sage . . . . . 8
1.4. Comparación de números . . . . . 10
1.5. Primos y factorización . . . . . 11
1.6. División euclídea . . . . . 13
1.7. Máximo común divisor y Mínimo común múltiplo . . . . . 14
1.8. Números complejos . . . . . 15
1.9. Miscelánea . . . . . 19

**2. Ecuaciones y sistemas** **21**
2.1. Resolución de ecuaciones elementales en modo exacto . . . . . 21
2.2. Resolución de sistemas . . . . . 23
2.3. Manipulación de ecuaciones . . . . . 24
2.4. Resolución numérica de ecuaciones . . . . . 26

**3. Polinomios** **27**
3.1. Operaciones con polinomios . . . . . 27
3.2. Factorización . . . . . 29
3.3. Raíces de polinomios . . . . . 31
3.4. Fracciones algebraicas . . . . . 32
3.5. Miscelánea . . . . . 33

**4. Gráficas y objetos gráficos** **36**
4.1. Gráficas de funciones . . . . . 36
4.2. Funciones a trozos y varias funciones . . . . . 37
4.3. Diversos objetos gráficos . . . . . 38

**5. Análisis** **41**
5.1. Límites . . . . . 42
5.2. Derivadas . . . . . 43
5.3. Integrales . . . . . 44
5.4. Polinomio de Taylor . . . . . 44

**6. Álgebra Lineal Elemental** **47**
6.1. Vectores y operaciones . . . . . 47
6.2. Matrices y operaciones . . . . . 50
6.3. Determinantes . . . . . 53
6.4. Operaciones elementales con matrices . . . . . 56
6.5. Sistemas de ecuaciones . . . . . 57

**7. Álgebra Lineal** **60**
7.1. Espacios vectoriales . . . . . 60
7.2. Las matrices y los espacios vectoriales . . . . . 64
7.3. Polinomio característico y autovectores . . . . . 66

**8. Teoría elemental de números** **69**
8.1. Los anillos $\mathbb{Z}_n$ . . . . . 69
8.2. Funciones multiplicativas . . . . . 71

**A. Cuaterniones** **75**

**B. Grupos abelianos finitos** **78**

# Capítulo 1

# Aritmética básica

Suele denominarse Aritmética al estudio de las operaciones con números enteros. Nosotros entenderemos por Aritmética también el estudio de las operaciones con otros tipos de números, como pueden ser los racionales y los reales.

## 1.1. Operaciones con enteros

Las operaciones con enteros se realizan en Sage con los operadores habituales. El uso de paréntesis es similar al que se realiza en matemáticas, así como la jerarquía de operaciones. Para calcular potencias se emplea el circunflejo (o el doble asterisco).

```
sage: # Todo lo que aparece tras el signo # es un comentario
sage: 45 + 89
134
sage: 48 - 963
-915
sage: 125*56
7000
sage: 125/5
25
sage: 45/23  # Si la division no es exacta tenemos un numero racional
45/23
sage: 190/24  # Si la fraccion es reducible, Sage la reduce
95/12
```

```
sage: # Podemos escribir dos o mas operaciones separadas por punto y coma
sage: 2^34; 2**34
17179869184
17179869184
sage: # Tambien se pueden separar por comas, y el resultado aparece
sage: # en forma de lista, esto es, entre parentesis
sage: 2^34, 2**34
(17179869184, 17179869184)
sage: 2^(-4)  # Las potencias de exponente negativo producen fracciones
1/16
sage: # Los parentesis y la jerarquia funcionan como se espera
sage: (2 + 6)^2 - 30*2 + 9
13
sage: a = 3  # Las variables se inicializan con el signo =
sage: 2*a, a^2, 3*a - 2
(6, 9, 7)
```

## 1.2. Operaciones con números racionales y reales

Las fracciones se escriben en la forma habitual, con la barra de división. Los números reales se escriben con el punto decimal. Si queremos escribirlos en notación científica, el exponente de la potencia de 10 se escribe tras la letra `e`. La raíz cuadrada se calcula con la función `sqrt()`. Para calcular la raíz $n$-ésima elevamos a $1/n$. Sage simplifica radicales, extrayendo factores siempre que es posible.

```
sage: 4/7 + 8/3
68/21
sage: 3/5 - 6/11
3/55
sage: (3/4 - 4/5)^3 + (2/3) / (34/89)
711949/408000
sage: 1.563 + 3.89
5.45300000000000
```

```
sage: 1.456e25 / 2.456e-12
5.92833876221498e36
sage: (1.123782)^100
117005.737177117
sage: # Si aparecen numeros racionales y decimales en una operacion
sage: # el programa da la solucion en numeros decimales
sage: 4/5 + 0.87
1.67000000000000
sage: sqrt(4), sqrt(8), sqrt(32)
(2, 2*sqrt(2), 4*sqrt(2))
sage: 1024^(1/4)   # La raiz cuarta
4*4^(1/4)
sage: 1024^(1/7)
2*8^(1/7)
```

En Matemáticas existen distintos conjuntos numéricos. Tenemos el conjunto de los números enteros, denotado por $\mathbb{Z}$, el conjunto de los números racionales $\mathbb{Q}$, etc. En Sage hay algo similar, aunque no totalmente igual: en $\mathbb{Z}$ y en $\mathbb{Q}$ el programa utiliza aritmética exacta, sin embargo en $\mathbb{R}$ y en $\mathbb{C}$ utiliza aritmética aproximada, por lo que pueden producirse errores. El conjunto de los enteros se denota en el programa por `ZZ`. Los racionales, los reales y los complejos se denotan `QQ`, `RR` y `CC`. Para saber en que conjunto considera Sage que está cada número utilizamos el método

```
.base_ring()
```

aunque también se puede utilizar la función

```
type()
```

La diferencia entre método y función es su forma de utilización. Para usar una función escribimos el nombre de la función y entre paréntesis el objeto al que se le aplica la función. Sin embargo para utilizar un método, primero escribimos el objeto sobre el que va a actuar, después escribimos un punto y finalmente el método.

Para obtener aproximaciones decimales utilizamos la función (y a la vez método)

```
n()
```

Esta función admite como opción `digits` para obtener las aproximaciones con distinto número de cifras.

Para extraer el denominador y el numerador de un número racional

`.denominator(), .numerator()`

Los números reales se pueden aproximar a los enteros de distintos modos. La forma más habitual es el redondeo, que devuelve el número entero más próximo al número decimal en cuestion. Pero también podemos redondear por defecto y por exceso. Los comandos para realizar todos estos tipos de redondeo son

`.round(), .floor(), .ceil()`

```
sage: a = 2
sage: a.base_ring()
Integer Ring
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: # El comando ``in'' se utiliza como el simbolo pertenece
sage: a in ZZ, a in QQ, a in RR, a in CC
(True, True, True, True)
sage: a = 3/7
sage: a.base_ring()
Rational Field
sage: type(a)
<type 'sage.rings.rational.Rational'>
sage: a in ZZ, a in QQ, a in RR, a in CC
(False, True, True, True)
sage: a = 1.53876
sage: a.base_ring()
Real Field with 53 bits of precision
sage: type(a)
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: a in ZZ, a in QQ, a in RR, a in CC
(False, True, True, True)
sage: # El numero decimal tambien es un numero racional
```

```
sage: n(sqrt(2))   # Precision por defecto
1.41421356237310
sage: n(sqrt(2), digits=50)  # Mayor precision
1.4142135623730950488016887242096980785696718753769
sage: n(sqrt(2), digits=5)
1.4142
sage: sqrt(2).n()  # Actuando como metodo
1.41421356237310
sage: sqrt(2).n(digits=30)  # El modificador digits como metodo
1.4142135623730950488016887242l0
```

Estas 5 instrucciones se pueden utilizar como funciones o como métodos. El método `.round()` admite una opción, que nos sirve para especificar con cuantos decimales queremos el redondeo. Incluso admite números negativos, si queremos redondear a las decenas, centenas,.... Sin embargo, y por razones no muy bien conocidas, esto solamente funciona si utilizamos la función y no sirve con el método.

```
sage: a = 77/23
sage: denominator(a)
23
sage: a.denominator().
23
sage: a.numerator()
77
sage: a = -74/21
sage: a.numerator(), a.denominator()
(-74, 21)
sage: a = 3.67
sage: a.round(), a.floor(), a.ceil()
(4, 3, 4)
sage: a = 3.234
sage: a.round(), a.floor(), a.ceil()
(3, 3, 4)
sage: a = -3.898
sage: a.round(), a.floor(), a.ceil()
```

```
(-4, -4, -3)
sage: a = 3456.876543
sage: round(a), round(a,2), round(a,5)
(3457, 3456.88, 3456.87654)
sage: round(a,-1), round(a,-2), round(a,-4)
(3460.0, 3500.0, 0.0)
```

## 1.3. La ayuda en Sage

La notación de función es más intuitiva y más próxima al hacer matemático. Sin embargo los métodos tienen una ventaja adicional: como primero debemos escribir el objeto, Sage ya «sabe» qué métodos se pueden aplicar a dicho objeto. Si escribimos un objeto y el punto y pulsamos el tabulador, el programa nos muestra un menú con los métodos que se pueden aplicar a dicho objeto. Si hemos empezado a escribir el método y pulsamos el tabulador, el programa completa la escritura, o en caso de que haya varios métodos que comiencen por dichas letras nos da las alternativas. Esto ayuda mucho en el estudio y hace que muchas veces no tengamos que retener en la memoria la escritura exacta del método. Por esta razón nosotros potenciaremos la utilización de métodos.

Para obtener ayuda sobre un método o función, en vez de escribir los paréntesis, escribimos el signo de interrogación «?» y pulsamos el tabulador. Si la ayuda ocupa más de una pantalla la barra espaciadora nos permite avanzar página. Para salir de la ayuda debemos pulsar la tecla «q» (del inglés *quit*). Puede ocurrir que gran parte de la ayuda sea incomprensible para nosotros, pero hay una parte de texto (eso si, en inglés) y ejemplos de uso que nos pueden ser de utilidad. El uso de la ayuda es imprescindible, así como la completación con el tabulador, a pesar de que el texto quede con peor presentación y más difícil de leer. Otra cosa, la completación automática también sirve para funciones, pero aquí el problema es que alguna de las funciones puede no ser aplicable al objeto con el que estamos trabajando.

Si en vez de escribir el signo de interrogación escribimos dos signos de interrogación, el programa nos informa del código fuente de la función. En principio esto sólo tiene utilidad para una utilización más avanzada de Sage que la que proponemos en estas notas.

```
sage: # Veamos que metodos se pueden aplicar a un numero racional
sage: a = 7/3
sage: a.
a.abs                    a.inverse_mod           a.period
a.additive_order         a.is_integral           a.py
a.base_extend            a.is_nilpotent          a.quo_rem
a.base_ring              a.is_one                a.rename
a.category               a.is_square             a.reset_name
a.ceil                   a.is_unit               a.round
a.charpoly               a.is_zero               a.save
a.conjugate              a.lcm                   a.sobj
a.copy                   a.list                  a.sqrt
a.db                     a.minpoly               a.sqrt_approx
a.denom                  a.mod                   a.squarefree_part
a.denominator            a.mod_ui                a.str
a.divides                a.multiplicative_order  a.subs
a.dump                   a.n                     a.substitute
a.dumps                  a.norm                  a.support
a.factor                 a.nth_root              a.trace
a.floor                  a.numer                 a.val_unit
a.gamma                  a.numerator             a.valuation
a.gcd                    a.order                 a.version
a.height                 a.parent
sage: # Ahora los metodos que empiezan por d
sage: a.d
a.db           a.denominator  a.dump
a.denom        a.divides      a.dumps
sage: # Pedimos ayuda sobre metodo denominator
sage: a.denominator?
Type:                builtin_function_or_method
Base Class:        <type 'builtin_function_or_method'>
String Form:        <built-in method denominator of sage.rings.rational.Rational
object at 0x9df295c>
Namespace:        Interactive
```

```
Docstring:

            self.denominator(): Return the denominator of this rational number.

            EXAMPLES:
                sage: x = -5/11
                sage: x.denominator()
                11
                sage: x = 9/3
                sage: x.denominator()
                1

Class Docstring:
    <attribute '__doc__' of 'builtin_function_or_method' objects>

sage: # Ahora las funciones que empiezan por j
j_invariant_qexp  jacobi_P          join              jsmath
jacobi            jobs              jordan_block
```

## 1.4. Comparación de números

Los operadores de orden son

```
<, >, <=, >=
```

Cuando comparamos dos números con uno de estos operadores, la respuesta es `True`, si la desigualdad es verdadera y `False` en caso contrario.

Además de los operadores de orden, tenemos los operadores

```
==, !=
```

El primero devuelve `True` cuando los dos números que comparamos son iguales. En cambio el segundo operador devuelve `True` cuando ambos son distintos.

```
sage: 5 < 6
True
sage: -5 > 6
False
sage: 78 < 78, 78 <= 78
(False, True)
sage: # El signo de comparacion es == y no =
sage: 45 == 9 * 5
True
sage: 78 != 78
False
sage: # En vez del simbolo != tambien se puede emplear <>
sage: 78 <> 78, 78 <> 123
(False, True)
```

## 1.5. Primos y factorización

Un número natural es primo si sus únicos divisores (positivos) son la unidad y él mismo. Los números que no son primos se denominan compuestos. Para ellos rige el Teorema Fundamental de la Aritmética, que afirma que todo número compuesto admite una factorización (única salvo signos) en producto de primos.

La comprobación de que un número es primo es un problema computacionalmente costoso. Por ello muchas veces (sobre todo para enteros «muy» grandes) se hace un análisis probabilístico para verificar si un número es primo. Aún peor, desde el punto de vista computacional, es el problema de la factorización. Por ello solo debemos esperar de Sage resultados para enteros «razonables».

Para saber si un número es primo debemos emplear el método

```
.is_prime()
```

Sage también puede informarnos si un número es una potencia de un número primo

```
.is_prime_power()
```

Dado un número entero $n$, el primer número mayor que $n$ y primo

.next_prime()

Análogamente el primo inmediatamente anterior a $n$ se calcula con la función (en la versión 3.2.3 de Sage no es un método)

previous_prime()

Las siguientes funciones se expresan por si mismas (del mismo modo que la anterior, no son métodos, solamente funciones)

next_prime_power(), previous_prime_power()

```
sage: a = 89; a.is_prime()
True
sage: 81.is_prime_power()  # Sabemos que 81 = 3^4
True
sage: 100.is_prime_power()  # Sin embargo 100 = 2^2*5^2
False
sage: 9.next_prime()
11
sage: previous_prime(16)
13
sage: next_prime_power(14)
16
sage: previous_prime_power(90)
89
sage: # Efectivamente, puesto que 89 = 89^1
```

Cuando el número es compuesto podemos estar interesados en obtener su descomposición en factores primos

.factor()

Muy relacionado con el problema de la factorización, se encuentra el problema de hallar todos los divisores de un número dado. Multiplicando cualesquiera factores de la descomposición de un número obtenemos un divisor de dicho número. Es más, cualquier divisor se puede obtener de ese modo. Para obtener una lista con todos los divisores de un número

```
.divisors()
```

```
sage: 360.factor()
2^3 * 3^2 * 5
sage: # Comprobemos que es correcto
sage: 2^3 * 3^2 * 5
360
sage: 180.divisors()
[1, 2, 3, 4, 5, 6, 9, 10, 12, 15, 18, 20, 30, 36, 45, 60, 90, 180]
```

## 1.6. División euclídea

En el conjunto de los números enteros existe el concepto de división euclídea. Dados dos números enteros arbitrarios $a$, $b$ podemos encontrar dos números, únicos, $c$ y $r$ que verifican:

- $a = bc + r$.
- $r$ es un entero positivo o nulo, menor que $|b|$.

Para obtener el cociente de la división de $n$ entre $m$ se emplean el comando

```
//
```

Para obtener el resto se pueden emplear dos formas alternativas

```
.mod(), %
```

```
sage: 23 // 5
4
sage: 23 % 5
3
sage: # Comprobemos
sage: 4 * 5 + 3
23
sage: # Tambien funciona con numeros negativos
sage: -26 // 5
-6
sage: -26 % 5
4
sage: -6 * 5 + 4
-26
sage: # Lo mismo lo podemos realizar con el metodo mod
sage: (-26).mod(5)
4
sage: # El quinto numero de Fermat es
sage: 2^(2^5) + 1
4294967297
sage: # Fermat pensaba que era primo. Sin embargo Euler comprobo que
sage: 4294967297 % 641
0
```

## 1.7. Máximo común divisor y Mínimo común múltiplo

Dado un entero $n$ consideramos el conjunto de sus divisores positivos. Ciertamente este es un conjunto finito contenido en $\mathbb{N}$. Dado otro número $m$ consideramos nuevamente el conjunto de sus divisores. Hacemos la intersección de ambos conjuntos, que es necesariamente no vacía puesto que la unidad divide a todo número. El mayor de los números

de dicha intersección es, por definición, el máximo común divisor de ambos números. Esta es la definición clásica de este concepto.

Ahora consideramos los múltiplos positivos de $n$ y los de $m$. Hacemos la intersección de dichos conjuntos. Dicha intersección es no vacía, puesto que $nm$ está en ambos conjuntos. De todos los elementos de dicha intersección debe existir uno que sea el menor de todos (necesariamente es menor o igual que $nm$). Dicho número es el mínimo común múltiplo.

Para calcular el máximo común divisor de dos enteros

```
.gcd()
```

El mínimo común múltiplo

```
.lcm()
```

```
sage: (78).gcd(24)
6
sage: (78).lcm(24)
312
sage: # El producto del minimo por el maximo da el producto de los numeros
sage: 6 * 312, 78 * 24
(1872, 1872)
```

## 1.8. Números complejos

Los números complejos son expresiones de la forma $a + bi$ donde $a$ y $b$ son números reales e $i$ es una raíz cuadrada de $-1$ (o lo que es lo mismo, $i^2 = -1$). Por defecto Sage reconoce a la letra `i` como un objeto simbólico con la propiedad $i^2 = -1$. Sin embargo para el programa dicha letra no es todavía un número complejo. Para decirle al programa que trate a la letra `i` como número complejo debemos escribir (no es necesario comprender el funcionamiento de esta orden).

```
i = CC(i)
```

```
sage: type(i)
<class 'sage.functions.constants.I_class'>
sage: type(3 + 4*i)
<class 'sage.calculus.calculus.SymbolicArithmetic'>
sage: i^2
-1
sage: i = CC(i)
sage: type(i)
<type 'sage.rings.complex_number.ComplexNumber'>
sage: type(3 + 4*i)
<type 'sage.rings.complex_number.ComplexNumber'>
sage: i^2
-1.00000000000000
```

Las operaciones aritméticas se realizan nuevamente con los operadores habituales. En las respuestas de Sage la unidad imaginaria se escribe en mayúsculas. Las operaciones con números complejos se realizan en aritmética aproximada, por lo que son susceptibles a errores. Además el número de decimales que da el programa puede hacer difícil la lectura del resultado.

```
sage: i = CC(i)
sage: a = 5 + 8*i
sage: b = 3.4 + 9*i
sage: a + b
8.40000000000000 + 17.0000000000000*I
sage: 3*a + 8.9*b
45.2600000000000 + 104.100000000000*I
sage: a - b
1.60000000000000 - 1.00000000000000*I
sage: a * b
-55.0000000000000 + 72.2000000000000*I
sage: a / b
0.961538461538461 - 0.192307692307692*I
sage: a^5
```

```
25525.0000000000 - 70232.0000000000*I
```

Dado el número complejo $z = a + bi$ su parte real es $a$ y su parte imaginaria es $b$. Para obtenerlas utilizamos los métodos

```
.real(), .imag()
```

El módulo o valor absoluto es $\sqrt{a^2 + b^2}$

```
.abs()
```

El conjugado es $a - bi$

```
.conjugate()
```

La norma de un número complejo se obtiene multiplicando dicho número por su complejo conjugado. Es siempre un número real, que resulta coincidir con $a^2 + b^2$

```
.norm()
```

Si imaginamos el número complejo $z = a+bi$ como el vector $(a, b)$, dicho vector forma con la parte positiva del eje de las $x$ un cierto ángulo. Es el denominado argumento del número complejo (Sage lo mide en radianes)

```
.arg(), argument()
```

```
sage: i = CC(i)
sage: a = 5 + 8*i
sage: a.real()
5.00000000000000
sage: a.imag()
8.00000000000000
sage: a.abs()
9.43398113205660
sage: a.norm()
89.0000000000000
sage: sqrt(89.0)
```

```
9.43398113205660
sage: a.conjugate()
5.00000000000000 - 8.00000000000000*I
sage: a * a.conjugate()
89.0000000000000
sage: a.arg()
1.01219701145133
```

Como consecuencia del Teorema Fundamental del Álgebra todo número complejo tiene exactamente $n$ raíces $n$-ésimas (pues el polinomio $x^n - a$ debe tener $n$ soluciones). Para conocerlas utilizamos el comando

`.nth_root(n)`

donde debemos indicar como argumento el orden de la raíz deseada. Por defecto Sage solamente nos devuelve una de las raíces. Si utilizamos la opción `all = True` nos devuelve una lista con todas las raíces. Para el caso de la raíz cuadrada se puede utilizar `sqrt()`.

```
sage: i = CC(i)
sage: a = 4.6 + 9.68*i
sage: b = a.nth_root(5); b
1.56634459784419 + 0.359216283265026*I
sage: b^5
4.59999999999999 + 9.68000000000000*I
sage: a.nth_root(5, all = True)

[1.56634459784419 + 0.359216283265026*I,
 0.142392112822719 + 1.60068617272853*I,
 -1.47834143238984 + 0.630062176803190*I,
 -1.05605736501685 - 1.21128633243841*I,
 0.825662086739775 - 1.37867830035833*I]
sage: sqrt(a)
2.76743452871272 + 1.74891219639849*I
sage: sqrt(a, all = True)
[2.76743452871272 + 1.74891219639849*I,
 -2.76743452871272 - 1.74891219639849*I]
```

## 1.9. Miscelánea

Normalmente los números se escriben en base 10, pero se pueden expresar en otras bases. Sage está especialmente preparado para esta circunstancia. Si escribimos un número entero que comience por 0, Sage entenderá que es un número escrito en el sistema octal. Si el número empieza por 0x entiende que es un número escrito en hexadecimal. Para transformar un número a otra base $b$ (necesariamente menor que 36, que es la suma de las diez cifras y las 26 letras) se emplea el método

```
.str(b)
```

La factorial de un número entero positivo es el producto de todos los números enteros positivos menores o iguales que el dado

```
.factorial()
```

La sucesión de Fibonacci es $1, 1, 2, 3, 5, 8, \ldots$. Los dos primeros términos son la unidad. Los siguientes se obtienen sumando los dos anteriores. Para conocer el término $n$-ésimo de la sucesión

```
fibonacci()
```

En una sección anterior hemos dado la definición más elemental del concepto de máximo común divisor. Sin embargo muchas veces es más útil utilizar la siguiente. Dados dos enteros $p$ y $q$, consideremos todos los números naturales que se pueden expresar en la forma $np + mq$, siendo $n$ y $m$ números enteros arbitrarios. El elemento mínimo de dicho conjunto coincide con el máximo común divisor. Además del máximo común divisor $d$, muchas veces es necesario conocer una pareja de números enteros $n, m$ que cumplan $np + mq = d$. Para obtener todo empleamos el comando

```
.xgcd()
```

```
sage: 010, 011, 012
(8, 9, 10)
sage: 0x10, 0x11, 0x12
(16, 17, 18)
```

```
sage: 03456 + 0x23a
2408
sage: 348.str(2)
'101011100'
sage: 348.str(23)
'f3'
sage: factorial(4), factorial(5), 5*factorial(4)
(24, 120, 120)
sage: fibonacci(6), fibonacci(7), fibonacci(8)
(8, 13, 21)
sage: xgcd(78,24)
(6, 9, -29)
sage: 9*78 + (-24)*29
6
```

# Capítulo 2

# Ecuaciones y sistemas

Sage puede resolver muchos tipos de ecuaciones. Algunas las resuelve de un modo exacto y otras de un modo aproximado. Además de esto también resuelve ecuaciones en modo simbólico. En este caso los coeficientes de la ecuación son letras.

## 2.1. Resolución de ecuaciones elementales en modo exacto

En esta sección llamamos ecuaciones elementales a las polinómicas de grados bajos, para las que existe un fórmula (utilizando operaciones racionales y extracción de raíces) para su resolución. Los matemáticos italianos del Renacimiento encontraron las fórmulas de resolución para las ecuaciones de grado menor que 5. A principios de siglo 19 Abel y Ruffini demostraron que no existía ninguna fórmula general para la resolución de las ecuaciones de grado igual o superior al quinto. Finalmente, también en el siglo 19, Galois dió las condiciones necesarias y suficientes para que una ecuación de grado superior al cuarto fuese resoluble (por radicales).

En Sage las ecuaciones están separadas por dos signos igual. Si las separamos solamente por un signo igual nos dará un error. Si no escribimos el doble igual el programa supone que la ecuación está igualada a cero. También es capaz de resolver algunas ecuaciones de grado superior al cuarto.

El comando para resolver ecuaciones es

```
solve(ecuacion, x)
```

Este comando no tiene en cuenta la multiplicidad de las soluciones.

```
sage: solve(x^2 - 5*x +6 == 0, x)  # Una ecuacion tipica de segundo grado
[x == 3, x == 2]
sage: solve(x^2 - 5*x + 6, x)  # Ahora sin el signo igual
[x == 3, x == 2]
sage: solve(x^2 - 4*x + 4 == 0, x)  # Una con solucion doble
[x == 2]
sage: solve(x^3 - 3*x + 2 == 0, x)  # Una ecuacion de tercer grado facil
[x == -2, x == 1]
sage: solve((x + 1)/9 +(3*x - 4)/45 == x + (5*x - 54)/12, x)
[x == 814/223]
sage: solve(x^8 - 10*x^6 + 37*x^4 - 60*x^2 + 36, x)
[x == -sqrt(2), x == sqrt(2), x == -sqrt(3), x == sqrt(3)]
sage: # Esta ecuacion de octavo grado la ha resuelto
sage: # Sin embargo una arbitraria quinto grado no la resuelve
sage: solve(x^5 -4*x^4 - 13*x^2 + 2*x -4, x)
[0 == x^5 - 4*x^4 - 13*x^2 + 2*x - 4]
```

Hasta ahora en todas las ecuaciones que hemos resuelto la incógnita (o varible) es $x$. Si intentamos repetir esto con otra letra, el programa nos informará de un error. Naturalmente se pueden resolver ecuaciones con otro nombre de variable, pero para ello, antes de intentar resolverla debemos escribir

```
var('letra')
```

La letra debe ir necesariamente entre comillas simples. Se pueden «inicializar» más letras escribiendo varias separadas por comas (o simplemente por espacios).

```
sage: var('a')
a
sage: solve(a^2 - 5*a + 6, a)
[a == 3, a == 2]
sage: var('b,c,d,e')  # Ya podemos utilizar todas estas letras
```

```
(b, c, d, e)
sage: solve(d^2 - 5*d + 6, d)
[d == 3, d == 2]
```

Es muy común que al resolver ecuaciones aparezcan soluciones complejas. El símbolo que usa el programa para la unidad imaginaria es `I`.

```
sage: solve(x^2 +4,x)  # Solucion compleja
[x == -2*I, x == 2*I]
sage: solve(3*x^3 + 5*x^2 - 4*x +5,x)
[x == 61*(sqrt(3)*I/2 - 1/2)/(81*(sqrt(1423)/(18*sqrt(3)) - 2005/1458)^(1/3)) +
(sqrt(1423)/(18*sqrt(3)) - 2005/1458)^(1/3)*(-sqrt(3)*I/2 - 1/2) - 5/9, x ==
(sqrt(1423)/(18*sqrt(3)) - 2005/1458)^(1/3)*(sqrt(3)*I/2 - 1/2) +
61*(-sqrt(3)*I/2 - 1/2)/(81*(sqrt(1423)/(18*sqrt(3)) - 2005/1458)^(1/3)) - 5/9,
x == (sqrt(1423)/(18*sqrt(3)) - 2005/1458)^(1/3) +
61/(81*(sqrt(1423)/(18*sqrt(3)) - 2005/1458)^(1/3)) - 5/9]
```

## 2.2. Resolución de sistemas

Un sistema es simplemente un conjunto de ecuaciones. En Sage se escriben formando lo que técnicamente se conoce como una lista. Debemos escribir entre corchetes cada una de las ecuaciones y separarlas por comas. Además, en el comando `solve`, después de escribir la lista, debemos escribir los nombres de las variables que queramos resolver.

```
sage: var('x,y')
(x, y)
sage: solve([x + y == 2, x - y == 0], x, y)  # Un sistema lineal
[[x == 1, y == 1]]
sage: solve(x^2 + y == 2, y)  # Despejamos una letra
[y == 2 - x^2]
sage: solve([x + y == 2, 2*x + 2*y == 4], x, y)  # Un sistema indeterminado
[[x == 2 - r1, y == r1]]
sage: # Sage denomina r1 al parametro
```

```
sage: solve([x^2 + y^2 == 2, 2*x + 2*y == 3], x, y)  # Un sistema no lineal
[[x == (3 - sqrt(7))/4, y == (sqrt(7) + 3)/4],
[x == (sqrt(7) + 3)/4, y == (3 - sqrt(7))/4]]
```

## 2.3. Manipulación de ecuaciones

Cuando trabajamos con las ecuaciones en secundaria, el profesor nos indica que sigamos una serie de pasos. El primero sueler ser desarrollar totalmente ambos miembros de la ecuación. Después podemos mover términos de un miembro a otro de la ecuación empleando las reglas «lo que esta sumando pasa restando»,... En realidad lo que hacemos con esa regla es sumar (o restar) la misma cantidad a ambos miembros de la ecuación. Lo mismo sucede con las otras reglas.

Cuando aplicamos el método de reducción (o de Gauss) decimos que sumamos dos ecuaciones y lo que hacemos es sumar las parte izquierda y derecha por separado. Aunque no es muy habitual, también podemos multiplicar dos ecuaciones, multiplicando por separado las partes izquierda y derecha. Todo es lo hace Sage de un modo muy visual. También es habitual «elevar al cuadrado» ambos miembros de una ecuación. Esto no lo hace el programa directamente, pero podemos hacerlo multiplicando la ecuación por si misma.

```
sage: # Aunque no es necesario damos nombre a las ecuaciones
sage: f = (x^2 + 2 == x^3 + x)
sage: g = (3*x^2 - 5*x == 8)
sage: # Restamos 2 a la ecuacion f
sage: f - 2
x^2 == x^3 + x - 2
sage: # Tambien podriamos haber restado x^3 + x
sage: f - (x^3 + x)
-x^3 + x^2 - x + 2 == 0
sage: # Multiplicamos la ecuacion g por 9
sage: 9 * g
9*(3*x^2 - 5*x) == 72
sage: g / 88   # Ahora la dividimos por 88
```

```
(3*x^2 - 5*x)/88 == 1/11
sage: f + g  # "Sumamos" las dos ecuaciones
4*x^2 - 5*x + 2 == x^3 + x + 8
sage: f * g  # "Multiplicamos" las dos ecuaciones
(x^2 + 2)*(3*x^2 - 5*x) == 8*(x^3 + x)
sage: f * f  # Elevamos al cuadrado ambos miembros
(x^2 + 2)^2 == (x^3 + x)^2
```

Para desarrollar (expandir) ambos miembros de una ecuación se emplea el método

```
.expand()
```

Si queremos quedarnos con el primer miembro de la ecuacion

```
.left()
```

y para la parte derecha

```
.right()
```

Para sustituir la variable por un valor particular

```
.subs(x = valor)
```

```
sage: f = (x-3)^2 + (3*x-5)/89 == (2*x - 5)*(x-2)
sage: f
(3*x - 5)/89 + (x - 3)^2 == (x - 2)*(2*x - 5)
sage: f.expand()
x^2 - 531*x/89 + 796/89 == 2*x^2 - 9*x + 10
sage: f.left()
(3*x - 5)/89 + (x - 3)^2
sage: f.right()
(x - 2)*(2*x - 5)
sage: # Sustituimos x por el valor 7
sage: f.subs(x=7)
1440/89 == 45
```

## 2.4. Resolución numérica de ecuaciones

Cuando empezamos a estudiar las ecuaciones parece que todas tienen solución exacta y que basta seguir un método (más o menos complicado) para conseguir resolver de modo exacto la ecuación. Ello es debido, o bien a que las ecuaciones son muy sencillas, o bien a que están «preparadas» para que funcionen los métodos. Pero la realidad nos dice que en general una ecuación no es resoluble de modo exacto. Si queremos conseguir soluciones debemos recurrir a métodos númericos que nos porporcionarán las soluciones con el grado de aproximación deseado. Debido a los algoritmos empleados para los métodos númericos, debemos especificar el intervalo donde el algoritmo debe «buscar» la solución. Normalmente aunque haya varias soluciones en el intervalo dado, el algoritmo nos «encontrará» solamente una. En el caso de ecuaciones polinómicas hay métodos que nos permiten encontrar todas las soluciones, pero eso lo veremos en una sección posterior.

El método para encontrar la solución númerica de una ecuación es

```
.find_root(intervalo)
```

```
sage: f = (x^2 -5*x + 6 == 0)
sage: # Encontrar solucion en el intervalo (-20,5)
sage: f.find_root(-20,5)
3.0
sage: f.find_root(0,2.5)
2.0
sage: g = (sin(x^2)+3 == exp(x)) # Una ecuacion mas complicada
sage: g.find_root(-10,10)
1.3737974476331929
sage: g.subs(x=_)
3.950323394682415 == 3.950323394682415
sage: # La variable _ guarda siempre el ultimo valor calculado
sage: # El problema de este metodo es fijar un intervalo adecuado
sage: h = x^6 - 5*x^4 - 3*x^2 + 5*x - 12   # Una de sexto grado
sage: h.find_root(-10,20)
2.3554150099757294
```

# Capítulo 3

# Polinomios

A nivel elemental, un polinomio es simplemente una expresión algebraica formada con números, letras y operaciones de suma, resta, multiplicación y potenciación. Sin embargo a un nivel más avanzado debemos indicar, de manera explícita, que tipo de números van a ser los coeficientes y la letra con la que vamos a trabajar. Nosotros adoptaremos este último punto de vista para extraer todo el potencial de Sage.

En general, de ahora en adelante, los coeficientes serán números racionales o enteros, puesto que el programa trabaja con ellos utilizando aritmética exacta. Para ello, al comienzo del trabajo con Sage debemos escribir lo siguiente (no es necesario entender lo que significa esta instrucción).

```
R.<x> = PolynomialRing(QQ)
```

En este caso empleamos la letra `x` y `QQ` es la forma en que Sage denota el cuerpo de los números racionales. Existen otras formas de construir el anillo de polinomios con coeficientes en $\mathbb{Q}$. La siguiente tiene el mismo efecto y es más corta

```
R.<x> = QQ[]
```

## 3.1. Operaciones con polinomios

La suma, la resta, la multiplicación y la potenciación se realizan con los operadores habituales. El cociente de la división euclídea se consigue con el operador `//` y el resto con `%`. El uso de paréntesis es similar al utilizado en matemáticas.

Cada polinomio con coeficientes racionales induce una función real de variable real. Todo lo referente a funciones que veremos en los capítulos siguientes se puede aplicar a los polinomios.

Si trabajamos con polinomios sobre varios cuerpos el método

```
.base_ring()
```

nos indica sobre que cuerpo (o anillo) está definido el polinomio. La orden

```
.parent()
```

nos informa prácticamente de lo mismo que la anterior, pero es más completa.

```
sage: R.<x> = QQ[]
sage: # Aunque no es obligatorio es conveniente nombrar los polinomios
sage: f = 3*x^4 + 5*x^3 + 3*x + 2
sage: type(f)
<class 'sage.rings.polynomial.polynomial_element_generic.
Polynomial_rational_dense'>
sage: # Si hemos nombrado al polinomio, evaluarlo se hace como en Matematicas
sage: f(0), f(1), f(89)
(2, 13, 191751837)
sage: g = x^2 - 4*x + 1
sage: f + g
3*x^4 + 5*x^3 + x^2 - x + 3
sage: f - g
3*x^4 + 5*x^3 - x^2 + 7*x + 1
sage: f * g
3*x^6 - 7*x^5 - 17*x^4 + 8*x^3 - 10*x^2 - 5*x + 2
sage: g^3
x^6 - 12*x^5 + 51*x^4 - 88*x^3 + 51*x^2 - 12*x + 1
sage: # Hallamos el cociente
sage: f // g
3*x^2 + 17*x + 65
sage: # Ahora el resto
sage: f % g
246*x - 63
sage: f.base_ring()
Rational Field
sage: f.parent()
Univariate Polynomial Ring in x over Rational Field
```

## 3.2. Factorización

Decimos que un polinomio $g$ divide a $f$ si al realizar la división euclídea de $f$ entre $g$ el resto es nulo. Es claro que todo polinomio $f$ no nulo es divisible por cualquier constante

$k$. Asimismo es divisible por todo polinomio de la forma $kf$, siendo $k$ una constante no nula. Estos divisores se denominan **triviales**. Si un polinomio posee algún divisor no trivial decimos que es **reducible**. En caso contrario, esto es, si sus únicos divisores son los triviales, decimos que el polinomio es **irreducible**. La cuestión de la irreducibilidad es delicada, pues depende en gran medida del cuerpo base. Por ejemplo, $x^2+1$ es irreducible como polinomio con coeficientes racionales, pero entendido como polinomio con coeficientes complejos es reducible, pues es divisible entre $x - i$. El mismo comentario es válido en la cuestión de la factorización de un polinomio en factores irreducibles.

La comprobación de que un polinomio es irreducible (sobre el cuerpo de definición)

```
.is_irreducible()
```

Para factorizar un polinomio (sobre el cuerpo de definición)

```
.factor()
```

Para obtener el máximo común divisor de dos polinomios (que es independiente del cuerpo base)

```
.gcd()
```

y para el mínimo común múltiplo

```
.lcm()
```

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = (x-1)^2 * (x-6)^2 * (x^2 + 1)
sage: g = (x-1)^3 * (x-6) * (3*x - 10)
sage: f.factor()
(x - 6)^2 * (x - 1)^2 * (x^2 + 1)
sage: g.factor()
(3) * (x - 6) * (x - 10/3) * (x - 1)^3
sage: f.gcd(g)
x^3 - 8*x^2 + 13*x - 6
sage: _.factor()
(x - 6) * (x - 1)^2
```

```
sage: f.lcm(g)
3*x^8 - 55*x^7 + 378*x^6 - 1240*x^5 + 2185*x^4 - 2493*x^3 + 2170*x^2 -
1308*x + 360
sage: _.factor()
(3) * (x - 10/3) * (x - 6)^2 * (x - 1)^3 * (x^2 + 1)
sage: f.is_irreducible()
False
sage: (x^2 - 2).is_irreducible()
True
sage: # En Q este polinomio es irreducible.  Sobre R es claro que no,
sage: # pues tiene raices, sqrt(2) y -sqrt(2)
sage: R.<x> = RR[]
sage: f = x^2 - 2
sage: f.is_irreducible()
False
sage: f.factor()
(1.00000000000000*x - 1.41421356237310)*(1.00000000000000*x + 1.41421356237310)
```

## 3.3. Raíces de polinomios

Cuando hablamos de raíces de un polinomio debemos tener en cuenta en que cuerpo se encuentran las raíces. Por ejemplo, el polinomio $x^2 - 2$ no tiene raices en $\mathbb{Q}$, pero si que las tiene en $\mathbb{R}$. Por ello existen distintos comandos para hallar las raíces. Si queremos hallar las raíces (con su multiplicidad) que existen en el cuerpo base (en nuestro caso $\mathbb{Q}$)

```
.roots()
```

Si queremos las raíces reales (calculadas por métodos aproximados)

```
.real_roots()
```

y si queremos las complejas (también por métodos aproximados)

```
.complex_roots()
```

```
sage: R.<x> = QQ[]
sage: f = (x^2 + 1) * (x - 3)^2 * (x^2 - 2) * (5*x -13)
sage: f.roots()
[(13/5, 1), (3, 2)]
sage: # Tiene la raiz 13/5 simple y la raiz 3 doble. No tiene mas raices en Q
sage: f.real_roots()

[-1.41421356237310,
 1.41421356237310,
 2.60000000000012,
 2.99999974717083,
 3.00000025282904]
sage: # Han aparecido dos soluciones mas, que son las raices cuadradas de 2
sage: # Ademas observamos que la raiz doble 3 ahora son dos soluciones
sage: # muy proximas a 3. Esto es debido a que se trabaja en modo aproximado
sage: f.complex_roots()

[-1.41421356237310,
 1.41421356237310,
 2.60000000000012,
 2.99999974717083,
 3.00000025282904,
 3.06978292612115e-16 + 1.00000000000000*I,
 3.06978292612115e-16 - 1.00000000000000*I]
sage: # Las soluciones complejas, que son i y -i, se dan de modo aproximado
```

## 3.4. Fracciones algebraicas

Con dos polinomios podemos formar una fracción, que en general no será un polinomio. Las fracciones de polinomios se suelen denominar fracciones algebraicas o también funciones racionales. Sobre ellas se aplican gran parte de los comandos estudiados para polinomios y también para fracciones (númericas).

```
sage: R.<x> = QQ[x]
sage: r = (x^3 - 4)/(3*x^2 - 5*x + 1); s = (6*x^2 - 4*x)/(2*x-5)
sage: r + s
(20*x^4 - 47*x^3 + 26*x^2 - 12*x + 20)/(6*x^3 - 25*x^2 + 27*x - 5)
sage: s^3
(216*x^6 - 432*x^5 + 288*x^4 - 64*x^3)/(8*x^3 - 60*x^2 + 150*x - 125)
sage: r(4), r(-2), r(100)
(60/29, -12/23, 999996/29501)
sage: r.denominator()
3*x^2 - 5*x + 1
sage: r.numerator()
x^3 - 4
sage: type(r)
<class 'sage.rings.fraction_field_element.FractionFieldElement'>
sage: r.base_ring()
Rational Field
sage: r.parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

## 3.5. Miscelánea

Muchos de estos comandos pueden parecer inútiles, y ciertamente lo son, si trabajamos con Sage en modo interactivo (esto es, tecleando comandos y esperando la contestación). Sin embargo pueden ser muy útiles cuando se programa en Sage.

El grado de un polinomio

```
.degree()
```

Para obtener una lista con todos los coeficientes, ordenados de grado menor a grado mayor

```
.coeffs()
```

Para derivar e integrar polinomios

.derivative(), .integral()

La gráfica del polinomio se obtiene con

.plot()

Se pueden pasar varias opciones a este último método para mejorar la visualización de la gráfica (ver el capítulo sobre representación gráfica).

```
sage: R.<x> = QQ[x]
sage: f = x^4 + 3*x^3 - 2*x^2 + 9
sage: f.degree()
4
sage: f.coeffs()
[9, 0, -2, 3, 1]
sage: f.derivative()
4*x^3 + 9*x^2 - 4*x
sage: f.integral()
1/5*x^5 + 3/4*x^4 - 2/3*x^3 + 9*x
sage: f.plot()
sage: # Por defecto lo dibuja en el intervalo (0,1) y en color azul
sage: f.plot(-3,2, color = 'red')
sage: # Ahora se dibuja en el intervalo (-3,2) y en color rojo
```

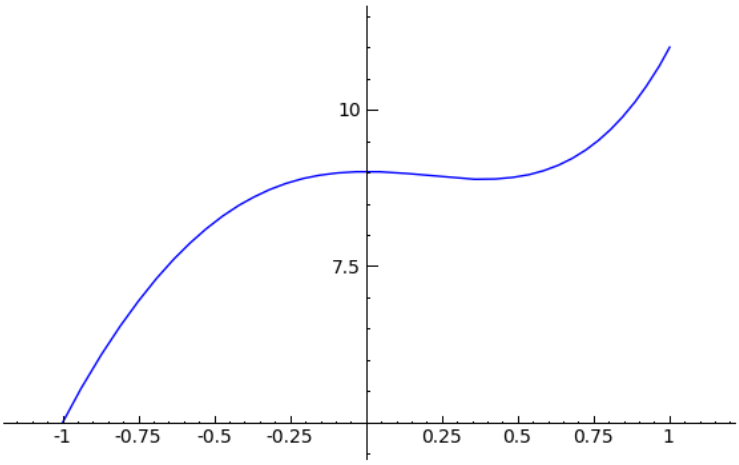


Figura 3.1: Gráfica construida con f.plot()

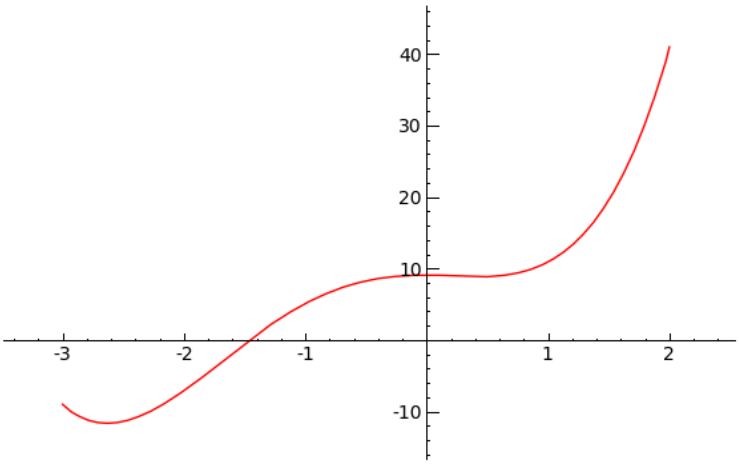


Figura 3.2: Gráfica construida con la orden f.plot(-3,2, color = 'red')

# Capítulo 4

# Gráficas y objetos gráficos

Sage es capaz de realizar gráficas de funciones reales de variable real dadas en varias formas: explícita (que es la forma habitual de presentar una función), paramétrica, implícita,... Asimismo puede realizar gráficas de funciones de dos variables, cuyo dibujo es una superficie. Analizaremos únicamente el dibujo de las funciones dadas en forma explícita. Además puede realizar el dibujo de ciertos objetos geométricos, como puntos, rectas, circunferencias,...

El formato gráfico por defecto de las imágenes generadas por Sage es png, que es un formato libre, de alta calidad y de pequeño tamaño (en bytes).

## 4.1. Gráficas de funciones

La instrucción básica para representar gráficas de funciones es la orden

```
plot(f, xmin, xmax, opciones)
```

siendo obligatorio únicamente el primer argumento. Si simplemente escribimos la función entre paréntesis, el programa realizará el dibujo en el intervalo $[-1, 1]$. El eje $y$ lo adaptará a la función. Normalmente la escala no será la misma en el eje $x$ y en el eje $y$.

```
sage: f(x) = x^2 -2*x +3
sage: f.plot()
```

Después de la función, podemos añadir, separados por comas, los extremos del intervalo donde queremos dibujar la gráfica.

```
sage: f(x) = x^2 -2*x +3
sage: f.plot(-2,5)  # Dibujo en el intervalo [-2,5]
```

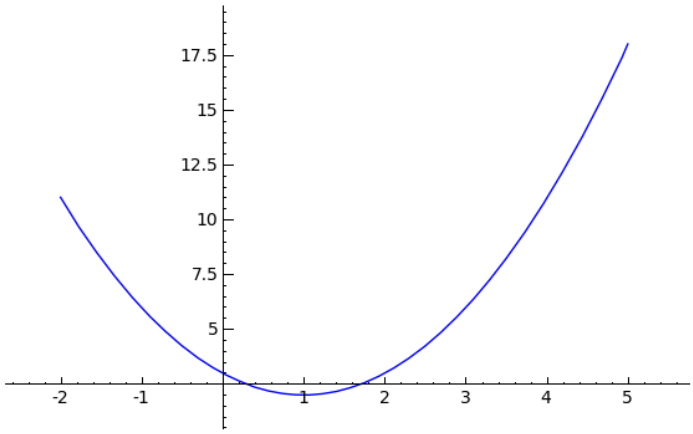


Figura 4.1: plot(f, -2, 5)

Podemos cambiar el color de gráfica añadiendo la opción `color`. El nombre del color debe figurar entre comillas simples. Existen otras opciones como `thickness` que afecta al grosor de la gráfica. Para ver todas las opciones del comando (y sus valores por defecto) tecleamos `plot.options` (sin paréntesis).

```
sage: f = x^2 - 2*x + 3
sage: plot(f, -3, 4, color = 'red', thickness = 2)
```

## 4.2. Funciones a trozos y varias funciones

Podemos dibujar varias gráficas en los mismos ejes. Para ello no debemos más que «sumar» las distintas ordenes `plot`. Con esto podremos «simular» una función definida a trozos, pero en realidad estamos haciendo trampas.

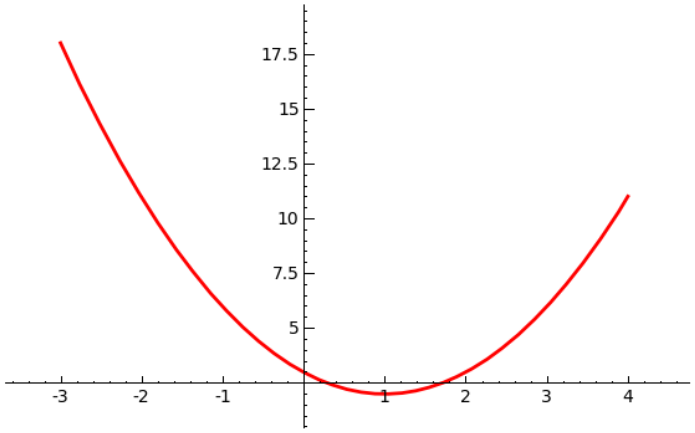


Figura 4.2: plot(f, -3, 4, color = 'red', thickness = 2)

```
plot(sin(x),-5,-1) + plot(x^2,-1,2) + plot(x,2,4)
```

Sin embargo la mejor opción es construir una verdadera función definida a trozos y después representarla. A efectos gráficos no tiene mucha importancia, pero en otros temas si la tendrá. La orden para definir una función con varios trozos es

```
Piecewise()
```

```
sage: f = Piecewise([[(-5,-1),sin(x)],[(-1,2),x^2],[(2,4),x]]); f
Piecewise defined function with 3 parts, [[(-5, -1), sin(x)],
[(-1, 2), x^2], [(2, 4), x]]
sage: f.plot()
```

## 4.3. Diversos objetos gráficos

Los diversos objetos gráficos de esta sección se pueden dibujar directamente con los comandos dados. Sin embargo los ejes se adaptan al dibujo, lo que conlleva que la apariencia de los dibujos sea muy distinta a la real (una circunferencia parece una elipse, la pendiente de una recta no es la que corresponde,...). Por ello lo mejor es guardar cada

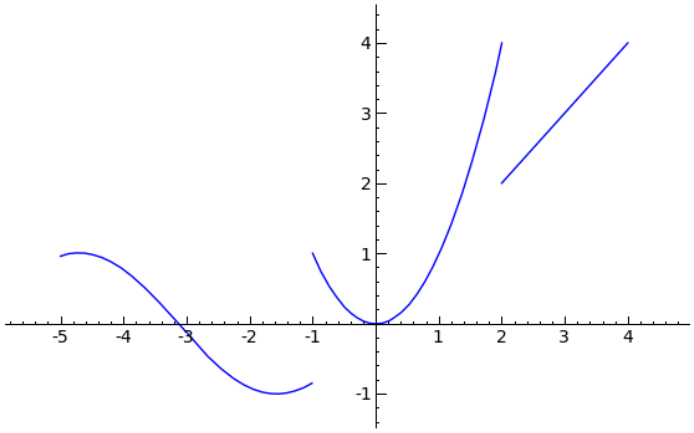


Figura 4.3: La gráfica de una función compuesta por 3 trozos

uno de los objetos gráficos en una variable y después utilizar el comando `show()` con la opción `aspect_ratio = 1` que hace que los ejes tengan la misma escala.

Para dibujar un punto

```
point2d(p)
```

donde `p` es un punto. Para dar las coordenadas de un punto, escribimos ambas entre paréntesis. Además de esto se le pueden añadir varias opciones. Para conocerlas todas (y sus valores por defecto) escribimos `point2d.options` (sin paréntesis)

Para dibujar una recta

```
line2d([p,q])
```

donde `p` es un punto extremo de la recta y `q` el otro. Ambos puntos deben ir colocados dentro de corchetes. Para ver las opciones se hace lo mismo que en el caso anterior.

Para dibujar una circunferencia

```
circle(centro, radio)
```

donde `centro` es un punto y `radio` un número positivo.

```
sage: point2d.options
{'alpha': 1, 'faceted': False, 'pointsize': 10, 'rgbcolor': (0, 0, 1)}
```

```
sage: line2d.options
{'alpha': 1, 'rgbcolor': (0, 0, 1), 'thickness': 1}
sage: circle.options
{'alpha': 1, 'fill': False, 'rgbcolor': (0, 0, 1), 'thickness': 1}
sage: a = point2d((2,3))
sage: b = line2d([(-1,-2),(4,1)])  # Una linea (o mejor un segmento)
sage: c = circle((2,1), 1.5)
sage: type(a)
<class 'sage.plot.plot.Graphics'>
sage: # El comando show muestra los objetos graficos. La suma de objetos
sage: # graficos significa que los muestra todos
sage: show(a + b + c, aspect_ratio = 1)  # Ejes con la misma escala
```

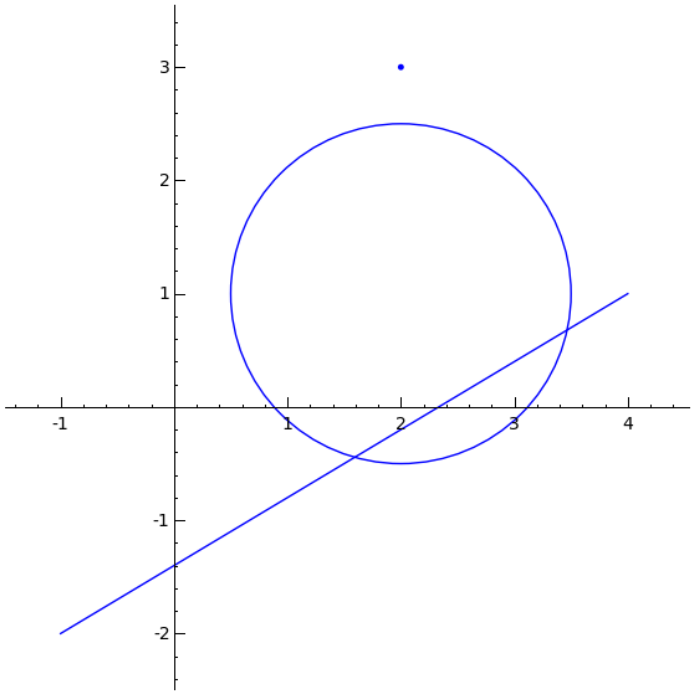


Figura 4.4: Un montón de objetos en un solo gráfico

# Capítulo 5

# Análisis

Puede definirse el Análisis, de una manera un tanto tosca, como el estudio de las funciones. Las funciones más elementales son sin lugar a duda las polinómicas, construidas con una variable y las operaciones de suma, resta y multiplicación. A un nivel un poco más elevado están las funciones algebraicas, formadas por cocientes de polinomios. En estas encontramos ya los primeros problemas con el dominio de definición. Sin embargo pronto se descubre que con estas funciones no es suficiente. A pesar de que el abanico de posibilidades es ilimitado, la siguiente tabla muestra algunas de las funciones predefinidas en Sage. Su uso es similar al que se realiza en Matemáticas.

| | |
|---|---|
| `sin(x)` | Seno de $x$ medido en radianes |
| `cos(x)` | Coseno de $x$ medido en radianes |
| `tan(x)` | Tangente de $x$ medido en radianes |
| `asin(x)` | ArcoSeno de $x$ |
| `acos(x)` | ArcoCoseno de $x$ |
| `atan(x)` | ArcoTangente de $x$ |
| `abs(x)` | Valor absoluto de $x$ |
| `sinh(x)` | Seno hiperbólico de $x$ |
| `cosh(x)` | Coseno hiperbólico de $x$ |
| `tanh(x)` | Tangente hiperbólica de $x$ |
| `exp(x)` | Exponencial de base el número $e$ |
| `log(x, base)` | Logaritmo. Por defecto la base es $e$ |
| `ln(x)` | Logaritmo natural (en base $e$) |

## 5.1. Límites

El primer concepto necesario para desarrollar el Análisis es el de límite. Para calcular un limite necesitamos al menos dos datos: una función y un punto en el que calcular el límite. Para la existencia de límite los dos límites laterales deben existir y coincidir. Si queremos especificar un límite lateral necesitamos informar al programa de otro dato, lo que haremos con la opción `dir`, que puede adoptar los valores (siempre entre comillas simples) `plus` (por la derecha) o `minus` (por la izquierda). También es muy habitual calcular límites en el infinito. La notación de Sage para el infinito es `oo`. El comando para calcular límites es

```
limit(f, x = a)
```

Este comando tiene una opción, llamada `Taylor`, que nos permite resolver más límites. Un comentario sobre el polinomio de Taylor se da en la última sección de este capítulo.

```
sage: f = x^2 + 3
sage: limit(f, x = 7)  # Un limite sencillo
52
sage: f.subs(x = 7)   # Se podria haber hecho por sustitucion
52
sage: limit(sin(x)/x, x = 0)  # Este da lugar a una indeterminacion
1
sage: limit(ln(x), x = 0, dir = 'plus')   # Por la izquierda no existe
-Infinity
sage: limit(exp(x), x = oo), limit(exp(x), x = -oo)
(+Infinity, 0)
sage: var('t')  # Se puede utilizar cualquier variable
t
sage: limit(sin(t)/t, t = 0)
1
```

## 5.2. Derivadas

Las derivadas de funciones se calculan con los comandos (ambos comando son «alias», o sea, son exactamente iguales)

```
diff(), derivative()
```

Si la función a derivar es de una sola variable se pueden usar con un único argumento.El segundo argumento, que por defecto es 1, indica el orden de derivación (derivada segunda, tercera, etc).

```
sage: f(x) = x^4 + 3*x^3 + 2*x + sin(x); f
x |--> sin(x) + x^4 + 3*x^3 + 2*x
sage: f.diff()
x |--> cos(x) + 4*x^3 + 9*x^2 + 2
sage: f.diff(3)  # Derivada tercera
x |--> -cos(x) + 24*x + 18
sage: f.derivative()  # El otro comando
x |--> cos(x) + 4*x^3 + 9*x^2 + 2
```

Si la función es de varias variables, tenemos la derivada parcial. Para ello es necesario indicar, como segundo argumento, la variable sobre la que se deriva. Por lo demás el funcionamiento es similar al caso de una variable.

```
sage: var('x,y')
(x, y)
sage: f(x,y) = x^4*y^5
sage: f.diff(x)  # Derivada parcial respecto a x
(x, y) |--> 4*x^3*y^5
sage: f.diff(x,3)  # Derivada parcial tercera respecto a x
(x, y) |--> 24*x*y^5
sage: f.diff(x,y)  # Derivada parcial respecto a x y a y
(x, y) |--> 20*x^3*y^4
sage: f.diff(x,2,y,3)  # Segunda respecto a x y tercera respecto a y
(x, y) |--> 720*x^2*y^2
```

## 5.3. Integrales

Para realizar integrales empleamos los comandos (alias)

`integral(), integrate()`

Dicha función tiene cuatro argumentos, siendo los tres ultimos opcionales. El primero es la funcion a integrar. Si la funcion tiene una sola variable con esto llega para realizar integrales indefinidas. El segundo argumento es la letra sobre la que se realiza la integración. Si la función tiene más de una variable es imprescindible informar de ello al programa. La integración en cualquiera de los dos casos es indefinida. Los dos últimos argumentos son los límites inferior y superior, por lo que si los añadimos estaremos realizando una integración definida.

```
sage: f(x) = x^2
sage: f.integral()
x |--> x^3/3
sage: f.integral(x,0,10)  # Esta ya es una integral definida
1000/3
sage: var('a')
a
sage: f(a,x) = a * sin(x)  # Una funcion con dos variables
sage: f.integral(x)
(a, x) |--> -a*cos(x)
sage: f.integral(a)
(a, x) |--> a^2*sin(x)/2
```

## 5.4. Polinomio de Taylor

Consideremos una función $f(x)$ y un punto $x = a$ en el dominio de $f$. Sea $n$ un número natural. Consideramos ahora el conjunto de todos los polinomios reales de grado menor que $n$. Entre todos ellos, existe uno que, en cierto sentido, es el «más» próximo a $f(x)$ en un entorno del punto $x = a$. Para calcular dicho polinomio utilizamos la orden

taylor(f, x, a, n)

```
sage: f = exp(x) * sin(x)
sage: t3 = taylor(f, x, 1, 3); t3
sin(1)*e + (sin(1) + cos(1))*e*(x - 1) + cos(1)*e*(x - 1)^2 -
(sin(1) - cos(1))*e*(x - 1)^3/3
sage: expand(t3)
-1*e*sin(1)*x^3/3 + e*cos(1)*x^3/3 + e*sin(1)*x^2 + e*sin(1)/3 - e*cos(1)/3
sage: # Observese que es un polinomio de grado 3. Los senos, cosenos
sage: # y exponenciales que aparecen son numeros
sage: t7 = taylor(f, x, 1, 7)
sage: plot(f, -2, 4) + plot(t3, -2, 4, color = 'red')

sage: plot(f, -2, 4) + plot(t7, -2, 4, color = 'green')
```

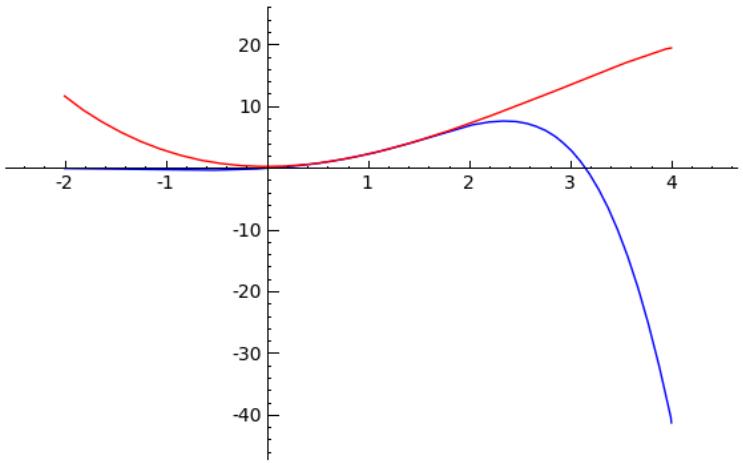


Figura 5.1: El polinomio de grado 3 se «pega» a la función cerca del 1

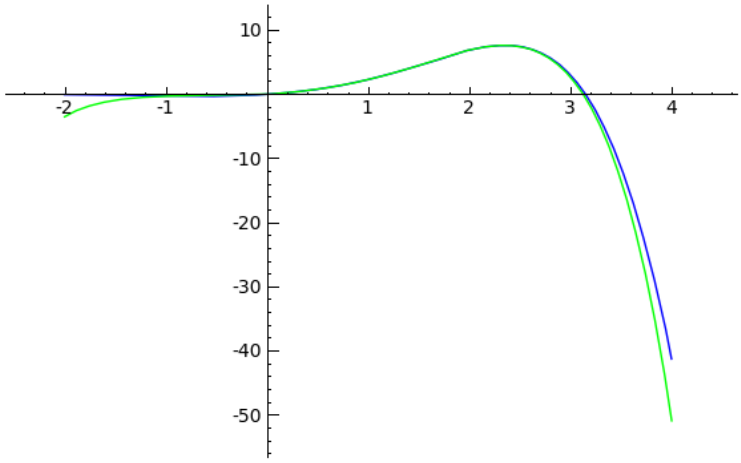


Figura 5.2: El polinomio de grado 7 se aproxima todavía más

# Capítulo 6

# Álgebra Lineal Elemental

## 6.1. Vectores y operaciones

Los objetos más básicos del Álgebra Lineal son los vectores. A nivel elemental un vector es simplemente una colección ordenada de números. Si tenemos dos números decimos que el vector es de dos dimensiones, si tenemos una terna hablamos de vectores tridimensionales, ...

Para construir un vector se utiliza la orden

```
vector(R, lista)
```

Las listas en Sage siempre van entre corchetes. El tamaño de la lista coincide con la dimensión del vector. Matemáticamente todo vector es un elemento de cierto $A^n$, donde $A$ es anillo. Para indicarle a Sage el anillo o cuerpo que contiene a los coeficientes, lo introducimos como opción en el comando que crea los vectores. Si no introducimos ninguno, Sage asocia a cada vector el mínimo anillo que contiene a los coeficientes que forman parte del vector. Para saber cual es el conjunto que considera Sage que contiene a los coeficientes

```
.base_ring()
```

```
sage: u = vector([2, 5, 6])
sage: type(u)
<type 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

```
sage: u.base_ring()
Integer Ring
sage: v = vector([4, 3/5, 8/3])
sage: type(v)
<type 'sage.modules.vector_rational_dense.Vector_rational_dense'>
sage: v.base_ring()
Rational Field
sage: w = vector([3, 6, 1.56])
sage: type(w)
<type 'sage.modules.free_module_element.FreeModuleElement_generic_dense'>
sage: w.base_ring()
Real Field with 53 bits of precision
sage: u + v
(6, 28/5, 26/3)
sage: 4*u + 6*v - 1.09*w
(28.7300000000000, 17.0600000000000, 38.2996000000000)
sage: # Podemos cambiar el anillo asociado a un vector
sage: a = vector(QQ, [2, 5, 6]);  a.base_ring()
Rational Field
```

Para realizar el producto escalar de dos vectores se pueden emplear indistintamente los siguientes métodos (aunque también se puede utilizar el asterisco)

`.dot_product(), .inner_product()`

La norma de un vector (que es la raíz cuadrada del producto escalar del vector consigo mismo)

`.norm()`

```
sage: u = vector([2, 5, 6]);  v = vector([4, 3/5, 8/3])
sage: u.dot_product(v)
27
sage: v.inner_product(u)
27
sage: u * v  # La multiplicacion de vectores es la escalar
```

```
27
sage: u.norm(), u * u
(sqrt(65), 65)
```

Para el producto vectorial (definido solamente para vectores tridimensionales)

```
.cross_product()
```

Aunque no es muy habitual, también a veces es necesario multiplicar dos vectores coordenada a coordenada, siendo el resultado un vector de la misma dimensión

```
.pairwise_product()
```

Para transponer un vector y «colocarlo» vertical

```
.transpose()
```

El resultado de transponer un vector, no es un vector sino una matriz.

Sage puede dibujar vectores de dos y tres dimensiones. El método para hacerlo es

```
.plot()
```

```
sage: u = vector([2, 5, 6])
sage: v = vector([4, 3/5, 8/3])
sage: # El producto vectorial no conmuta, sino que anticonmuta
sage: u.cross_product(v); v.cross_product(u)
(146/15, 56/3, -94/5)
(-146/15, -56/3, 94/5)
sage: u.pairwise_product(v)
(8, 3, 16)
sage: u.transpose()

[2]
[5]
[6]
sage: type(_)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: u = vector([2,4])
sage: u.plot()
```

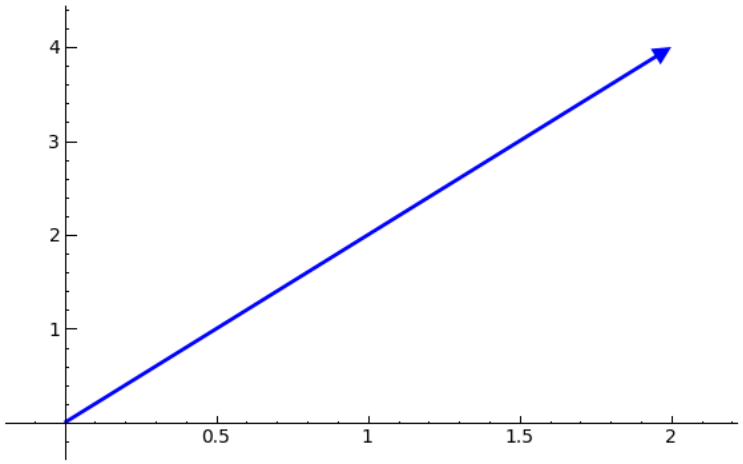


Figura 6.1: Vector bidimensional construido con u.plot()

## 6.2. Matrices y operaciones

A nivel operativo, puede decirse que la parte fundamental del Álgebra Lineal es la teoría de matrices. A nivel matemático un vector es simplemente una matriz de una sola fila. Sin embargo para Sage son distintos y se definen con distintos comandos.

Para construir una matriz utilizamos el comando

```
matrix(R, [f1, f2, ... , fn])
```

Una matriz es una «lista de listas». Por lo tanto dentro del paréntesis debe ir situado siempre un corchete. Dentro de dicho corchete deben figurar las filas de la matriz en cuestión. Dichas filas son ellas mismas listas, por lo que también debemos colocarlas entre corchetes. De la misma forma que en los vectores, podemos cambiar el anillo asociado, añadiéndolo como opción.

Las operaciones con matrices y vectores se realizan con los operadores habituales. El producto matricial se realiza con el asterisco y las potencias con el circunflejo.

```
sage: A = matrix([[1,4,6],[3,6,8],[-3,8,1]]); A

[ 1  4  6]
[ 3  6  8]
[-3  8  1]
```

```
sage: B = matrix([[-3,7,9],[8,-3,-7],[-2,0,9]]); B

[-3  7  9]
[ 8 -3 -7]
[-2  0  9]
sage: type(A)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: A.base_ring()
Integer Ring
sage: A + B

[-2 11 15]
[11  3  1]
[-5  8 10]
sage: 3*A - 5*B

[ 18 -23 -27]
[-31  33  59]
[  1  24 -42]
sage: # La multiplicación no es conmutativa
sage: A * B

[ 17  -5  35]
[ 23   3  57]
[ 71 -45 -74]
sage: B * A

[ -9 102  47]
[ 20 -42  17]
[-29  64  -3]
sage: A^3

[  91  788  622]
[ 111 1252  952]
```

```
[   9  712  507]
```

Escribir matrices de ordenes altos suele resultar tedioso. Sage es capaz de «inventarse» matrices por nosotros. Para ello debemos indicarle de que tipo van a ser los elementos de la matriz, el tamaño de la matriz y el máximo de los números aleatorios que puede generar el programa. La orden es

```
random_matrix(ZZ, n, m, x = max)
```

Aquí `n` y `m` indican el tamaño de la matriz. Si se omite alguno de ellos, el programa supone que la matriz es cuadrada.

Para constuir la matriz identidad de orden $n$

```
identity_matrix(n)
```

Para construir una matriz diagonal

```
diagonal_matrix(lista)
```

donde `lista` es una lista formada por los elementos de la diagonal principal.

```
sage: A = random_matrix(ZZ, 3, 4, x = 10); A
[7 4 7 7]
[2 2 0 3]
[5 6 6 4]
sage: I = identity_matrix(3)
sage: I * A == A
True
sage: D = diagonal_matrix([3,6,-2,7]); D
[ 3  0  0  0]
[ 0  6  0  0]
[ 0  0 -2  0]
[ 0  0  0  7]
```

## 6.3. Determinantes

Los determinantes son números asociados a las matrices cuadradas. En ellos está resumido cierta información algebraica y geométrica relacionada con la matriz. Es conocido que una matriz es invertible si y solamente si su determinante es distinto de cero. Si la matriz no es cuadrada, «tachando» filas y columnas podemos conseguir matrices cuadradas. Los determinantes de dichas matrices son los llamados menores. Ligado a estos menores está el concepto de rango, que es el orden del mayor menor no nulo (aunque lo parezca no es un juego de palabras). Los menores también sirven para construir la llamada matriz adjunta, muy relacionada con la inversa de la matriz.

Para calcular el determinante hay dos órdenes sinónimas

```
.det(), .determinant()
```

Para transponer una matriz

```
.transpose()
```

```
sage: A = random_matrix(ZZ, 4, x = 10); A

[4 2 3 7]
[6 9 3 2]
[4 9 2 5]
[1 6 0 4]
sage: B = A.transpose()
sage: A.det(); B.det()
62
62
sage: # Veamos la potencia de calculo de Sage
sage: A = random_matrix(ZZ, 500, x = 100)  # Una matriz de buen tamaño
sage: # Si escribimos time delante de un comando, nos dice lo que tarda
sage: # nuestro ordenador en realizar la operacion
sage: time A.det()
CPU times: user 9.21 s, sys: 0.07 s, total: 9.28 s
Wall time: 9.47 s
-513121187660621782157902879359103778048155214452767958444879925272897 90
```

```
185754714277249296185253535819264346574960388348158150444222215717575271
582620208094273237654671407368449045494839625212568472820719182539369433
864144720619663533251447469207291330153663448885671161958743048804994401
033369198349588668211157092638507512238595926560582643712759086633633708
482592874208145460666299045405171350050208266234159017778314185118760333
122454222253557536031505255901630478649166521589863768860741764828469001
683955795874513743653697706392383461862894573146608266064281142813682113
343342262901515285739793161679021156636778269697375440293345428819927467
796075365373944915716559812112994273303001261081017214968312830561064174
742102238656132613385336089794969093158673579033166743398037770518036090
200513908945937873038712551913007436660951653274622106974218118890880317
868291047134215148396261992465767292143426822492602358671070588167633756
452096207163642405442038989272613126779682918059872287314653467587309168
493206248897710248054823898698472410039075480209369247191548256509067865
446925519063716764816895631173310097276903861710660521969905714397564184
207334741277542639830742489806961602424027410160496053353915810429096033
824699653357329390587994669534905897466587232310063644315336894711907545
5330
```

Si la matriz es invertible, su inversa se calcula elevando a -1 o con la orden

```
.inverse()
```

Los menores de una matriz

```
.minors(n)
```

donde `n` es el tamaño de los menores. Debemos tener cuidado pues la colocación de los menores puede resultar inesperado para muchos, puesto que no es el mismo que el que se utiliza en la matriz adjunta.

El rango de la matriz

```
.rank()
```

La matriz adjunta

.adjoint()

En España se suele denominar matriz adjunta a la formada por los menores. Si queremos construir la matriz inversa, esta matriz adjunta, la debemos transponer y después dividir por un número. Sin embargo en los países anglosagones denominan matriz adjunta a lo que para nosotros sería la matriz adjunta y transpuesta. Naturalmente Sage sigue la convención anglosagona.

```
sage: A = random_matrix(ZZ, 4, x = 10)
sage: A.det()
-906
sage: # Determinante no nulo, implica rango maximo
sage: A.rank()
4
sage: # Dos formas de calcular la inversa
sage: A^(-1)

[  38/453   61/453 -104/453   -8/453]
[ -22/151  -15/302   33/302   49/302]
[   7/151  -57/302   65/302    5/302]
[  -2/151   81/302    3/302  -23/302]
sage: A.inverse()

[  38/453   61/453 -104/453   -8/453]
[ -22/151  -15/302   33/302   49/302]
[   7/151  -57/302   65/302    5/302]
[  -2/151   81/302    3/302  -23/302]
sage: # Observese la colocacion de los menores y el signo
sage: A.adjoint()

[ -76 -122  208   16]
[ 132   45  -99 -147]
[ -42  171 -195  -15]
[  12 -243   -9   69]
sage: A.minors(3)
```

```
[69, 15, -147, -16, 9, -195, 99, 208, -243, -171, 45, 122, -12, -42, -132, -76]
```

## 6.4. Operaciones elementales con matrices

Llamamos operación elemental a la suma de un múltiplo de una columna a otra columna (también con filas). Las operaciones elementales tienen la propiedad de que no varían el rango de la matriz (la explicación de este hecho en el capítulo siguiente). Las operaciones con filas también útiles en el método de Gauss, que es uno de los métodos elementales para resolver sistemas de ecuaciones lineales.

Las operaciones elementales con Sage presentan el problema de que la numeración empieza en cero. Para definir una operación elemental debemos dar 3 datos: la fila a la que vamos a multiplicar, el escalar por el que vamos a multiplicar y la fila a la que vamos a sumarle el resultado anterior. El método en el programa es

```
.add_multiple_of_row(f1, f2, lambda)
```

Esta operación sustituye la fila `f1` por `f1 + lambda * f2`.

Si queremos hacer operaciones elementales con columnas el comando es

```
.add_multiple_of_row(f1, f2, lambda)
```

Debemos tener en cuenta que estos comandos cambian a la matriz sobre la que se ejecutan.

Normalmente se utilizan las operaciones elementales para triangularizar la matriz. En Sage hay dos comandos para triangularizar matrices

```
.echelon_form(), .echelonize()
```

El primero nos muestra como quedaría la matriz triangulada, sin embargo el segundo cambia la matriz sobre la que actua y la triangulariza (es un detalle técnico, pero importante).

```
sage: A = identity_matrix(4)
sage: A.add_multiple_of_row(0, 3, 100); A  # Le estamos sumando a la fila 1
[  1   0   0 100]
[  0   1   0   0]
[  0   0   1   0]
[  0   0   0   1]
sage: # Observamos que la matriz A ha cambiado
sage: A = random_matrix(ZZ, 4, x = 10)
sage: A.echelon_form()  # La matriz A no cambia
[   1    0    0  670]
[   0    1    0  222]
[   0    0    1   45]
[   0    0    0 1668]
sage: A
[2 8 5 5]
[3 5 5 9]
[7 0 7 1]
[2 9 0 2]
sage: A.echelonize(); A  # Ahora si que cambiamos la matriz A

[   1    0    0  670]
[   0    1    0  222]
[   0    0    1   45]
[   0    0    0 1668]
```

## 6.5. Sistemas de ecuaciones

Muchas ecuaciones matriciales son del tipo $A \cdot X = B$, donde $A$ y $B$ son matrices dadas y $X$ es la matriz incógnita. Sage es capaz de resolver sistemas de este tipo con el método

```
A.solve_right(B)
```

Del mismo modo las ecuaciones de la forma $X \cdot A = B$ se resuelve con el método

```
A.solve_left(B)
```

El primer caso incluye de modo natural los sistemas de ecuaciones lineales, que es el caso en que $B$ es una matriz columna. Aunque $B$ teóricamente debe ser una matriz columna, si en su lugar colocamos un vector, el programa lo interpreta como una columna.

Si el sistema es compatible determinado ($X$ existe y es única) entonces el programa calcula dicha solución. Si el sistema es incompatible ($X$ no existe) Sage produce un error. Si el sistema es compatible indeterminado (existen varias $X$ que cumplen la ecuación) el programa únicamente calcula una de ellas.

```
sage: A = random_matrix(ZZ, 4); A

[-4 -5  1  1]
[ 1 -7  2 -1]
[-6  1  1  4]
[ 1 -3  1 -4]
sage: B = random_matrix(ZZ, 4, 2); B

[ 1  0]
[-4  0]
[ 1 -1]
[ 2 -1]
sage: X = A.solve_right(B); X

[-27/23   4/23]
[ 17/69 -17/69]
[-82/69 -56/69]
[-88/69  19/69]
sage: A * X

[ 1  0]
[-4  0]
[ 1 -1]
```

```
[ 2 -1]
sage: # Si B es un vector, Sage comprende lo que queremos hacer
sage: b = vector([1,4,7, 90])
sage: X = A.solve_right(b); X
(-265/23, 2039/207, 6116/207, -5254/207)
sage: A * X
(1, 4, 7, 90)
```

# Capítulo 7

# Álgebra Lineal

El concepto fundamental del Álgebra Lineal es el de espacio vectorial. Un espacio vectorial es un grupo abeliano donde se ha definido una multiplicación por escalares. Los escalares forman siempre un cuerpo (si solamente forman un anillo, la estructura se denomina módulo). Fijado un cuerpo $k$, para cada número entero $n$ existe un único (salvo isomorfismos) $k$-espacio vectorial de dimensión $n$. También existen espacios vectoriales de dimensión infinita, aunque no trataremos ahora de ellos. El espacio coordenado $k^n$ con las estructuras canónicas es un modelo de espacio vectorial de dimensión $n$.

## 7.1. Espacios vectoriales

Una cualquiera de las órdenes

```
VectorSpace(k, n), k^n
```

crea el espacio vectorial $k^n$. Es lo que denominaremos espacio vectorial ambiente. Para construir un vector de un espacio vectorial `V` empleamos la orden

```
V(lista)
```

Naturalmente la lista debe tener el tamaño adecuado a la dimensión del espacio vectorial.

Para saber cual es la dimensión de un espacio vectorial `V` simplemente escribimos su nombre y el programa nos da todos los datos del espacio vectorial. También podemos utilizar el método

.dimension()

Todo espacio vectorial de dimensión $n$ posee una base, formada por $n$ vectores linealmente independientes. Aunque existan muchas bases en un espacio vectorial, Sage realiza una elección y asocia a cada espacio vectorial una base. En el caso de los espacios vectoriales canónicos $k^n$ existe una base también canónica y coincide con la que le asocia el programa. Para obtener una base de un espacio vectorial se emplea uno de los comandos

.basis(), basis_matrix()

La diferencia es puramente técnica. En el primer caso el resultado es una lista y en el segundo es una matriz, donde las filas forman los vectores de la base.

```
sage: V = VectorSpace(QQ, 5); V
Vector space of dimension 5 over Rational Field
sage: W = QQ^5; W
Vector space of dimension 5 over Rational Field
sage: V == W
True
sage: a = V([3,7,9,2/7,1])
sage: a in V
True
sage: V.dimension()
5
sage: V.basis()

[
(1, 0, 0, 0, 0),
(0, 1, 0, 0, 0),
(0, 0, 1, 0, 0),
(0, 0, 0, 1, 0),
(0, 0, 0, 0, 1)
]
sage: V.basis_matrix()

[1 0 0 0 0]
```

```
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

Dentro del espacio vectorial ambiente existen subconjuntos que tienen estructura de espacio vectorial. Son los denominados subespacios vectoriales. Para especificar un subespacio vectorial debemos dar un conjunto de vectores que lo generen, utilizando la orden

```
.subspace(lista)
```

Los subespacios son ellos mismos espacios vectoriales, por ello todo lo dicho para espacios vectoriales vale para subespacios. El problema que tiene este comando es que nosotros damos los vectores que generan y Sage «elige» una base del subespacio. Posiblemente no sea la base que a nosotros nos interese. Si queremos fijar una base del subespacio empleamos el comando

```
.subspace_with_basis(lista)
```

En este caso la lista deben ser vectores linealmente independientes y formarán la base asociada a dicho subespacio.

```
sage: V = QQ^5
sage: a = V([2,5,7,2,7]); b = V([2,0,8,2,5]); c = V([1,8,3,2,-90])
sage: # El subespacio generado por a y b es
sage: W = V.subspace([a,b]
sage: W.basis_matrix()

[   1    0    4    1  5/2]
[   0    1 -1/5    0  2/5]
sage: # Ahora construimos un subespacio vectorial donde a y b sean base
sage: Z = V.subspace_with_basis([a,b])
sage: Z.basis_matrix()

[2 5 7 2 7]
```

```
[2 0 8 2 5]
sage: Z == W
True
```

Si W y Z son subespacios de un mismo espacio vectorial (el llamado espacio vectorial ambiente) podemos realizar la intersección conjuntista de ambos subespacios. Resulta que el conjunto resultante es nuevamente un subespacio. La orden es

```
W.intersection(Z)
```

El mínimo subespacio que contiene a la vez a W y a Z se denota habitualmente W + Z y se dice que es la suma de dos subespacios. En Sage se utiliza la misma notación.

Todo espacio vectorial (en particular los subespacios) en Sage tienen vectores de $n$ coordenadas (aunque el espacio vectorial puede tener dimensión menor). Se denomina grado de un espacio vectorial precisamente al tamaño de los vectores que lo contienen. Si un espacio vectorial tiene grado $n$, es de modo natural un subespacio de $k^n$. Este último conjunto es lo que se denomina espacio vectorial ambiente. Las órdenes para extraer la información son

```
.degree(), .ambient_vector_space()
```

```
sage: V = QQ^5
sage: a = V([2,5,7,2,7]); b = V([2,0,8,2,5]); c = V([1,8,3,2,-90])
sage: W = V.subspace([a,b])
sage: Z = V.subspace([b,c])
sage: W.degree()
5
sage: W.dimension()
2
sage: W.intersection(Z)

Vector space of degree 5 and dimension 1 over Rational Field
Basis matrix:
[  1   0   4   1 5/2]
sage: W + Z
```

```
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[     1      0      0  -17/3 1281/2]
[     0      1      0    1/3  -63/2]
[     0      0      1    5/3 -319/2]
```

## 7.2. Las matrices y los espacios vectoriales

Las matrices aparecen en prácticamente cualquier campo de las Matemáticas. Sin embargo su origen histórico (relacionado con la resolución de sistemas lineales) hacen que su relación con el Álgebra Lineal sea muy estrecha. Veamos algunos puntos donde las matrices se aplican al Álgebra Lineal.

Fijadas bases en dos espacios vectoriales, cada aplicación lineal entre los espacios vectoriales da lugar a una matriz. El tamaño de la matriz es el producto de las dimensiones de ambos espacios. La correspondencia inversa también es correcta: a cada matriz le corresponde una aplicación lineal (una vez fijadas las bases). Esta es la primera relación que guardan las matrices con los espacios vectoriales.

Si tomamos una matriz, cada una de sus filas puede considerarse como un vector. Por lo tanto una matriz «es» un conjunto de vectores. Lo mismo puede decirse de las columnas de la matriz.

Cada matriz simétrica induce un producto escalar (no necesariamente definido positivo) en un espacio vectorial (una vez fijada una base). Aunque Sage permite tratar este aspecto, no lo discutiremos en estas notas.

En todo lo que sigue supondremos que las matrices y vectores tienen los tamaños adecuados para poder realizar las multiplicaciones. Asimismo a veces debemos entender un vector como una columna y no como una fila.

Dada una matriz $A$, se dice que un vector $v$ pertenece al núcleo de $A$ si $vA = 0$. Es fácil ver que el núcleo de cualquier matriz es siempre un subespacio vectorial. Esta el la noción de Sage para el núcleo de una matriz. Sin embargo en Álgebra Lineal es más habitual considerar el núcleo de una matriz al conjunto $v$ de vectores que satisfacen $Av = 0$. Por ello Sage tiene varios comandos relacionados con matrices que comienza o

bien por `left` o bien por `right`. En particular para calcular los núcleos de una matriz tenemos los comandos

`.left_kernel(), .right_kernel()`

```
sage: A = random_matrix(QQ, 3, 5); A
[   0    1    0    0    0]
[  -1   -1 -1/2    1  1/2]
[   1    2    1    0   -2]
sage: A.kernel()
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
sage: A.left_kernel()
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
sage: A.right_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[  1   0   0 3/4 1/2]
[  0   0   1 1/4 1/2]
sage: # Comprobemos que el primer vector de la base pertenece al núcleo
sage: A * vector(QQ, [1, 0, 0, 3/4, 1/2]).transpose()
[0]
[0]
[0]
sage: # Veamos que kernel es lo mismo que left_kernel en Sage
sage: A = random_matrix(QQ, 5, 3)
sage: A.kernel() == A.left_kernel()
True
```

Dada una matriz `A`, entendemos que sus columnas son vectores de la dimensión adecuada. El subespacio que generan las columnas de la matriz se obtiene con el método

`.column_space()`

Un resultado de Álgebra Lineal dice que la dimensión de este subespacio vectorial coincide con el rango de la matriz (que es número de vectores columna linealmente independientes). Si queremos hacer lo mismo, pero con los vectores fila, la orden es

```
.row_space()
```

El espacio vectorial tiene como cuerpo asociado el mismo que la matriz.

## 7.3. Polinomio característico y autovectores

Consideremos un espacio vectorial $V$ y un endomorfismo $\phi$. Fijada una base de dicho espacio vectorial, al endomorfismo $\phi$ le asociamos la matriz $A$. Todo lo que vamos a decir se refiere a la matriz, pero una comprensión más adecuada se tiene entendiendo la matriz como un endomorfismo.

Dada una matriz cuadrada $A$, decimos que un vector $v$ es un autovector si existe un escalar $\lambda$ tal que $Av = \lambda v$. El escalar $\lambda$ se denomina autovalor asociado a dicho vector. Se tienen los siguientes resultados de Álgebra Lineal.

- Fijado un autovalor $\lambda$, el conjunto de vectores $V_\lambda$ que son autovectores con dicho autovalor, es un subespacio vectorial.
- Dados dos autovalores distintos $\lambda$ y $\mu$, sus subespacios vectoriales asociados $V_\lambda$ y $V_\mu$ tienen intersección vacía. También se dice que los subespacios están en posición de suma directa.
- Los autovalores de una matriz son las raíces del polinomios característico (también las soluciones del polinomio mínimo). Debido a esto el estudio de los autovalores y autovectores depende fuertemente del cuerpo base. La situación ideal se produce cuando el cuerpo es algebraicamente cerrado.
- En general no es cierto que la suma de todos los subespacios asociados a todos los autovalores sea el espacio total (ni aun en el caso de cuerpos algebraicamente cerrados). En el caso en que esto ocurra se dice que la matriz es diagonalizable.

El polinomio característico y el mínimo se obtienen con

`.charpoly(), .minpoly()`

En muchos casos estos dos polinomios coinciden. A nivel operativo suele ser más sencillo calcular el polinomios característico, puesto que basta desarrollar un determinante. Sin embargo a nivel teórico es más interesante el polinomio mínimo, que es el polinomio mónico y de menor grado que «anula» a la matriz (un polinomio $p(x)$ anula a una matriz $A$ si $p(A)$ es la matriz nula). El Teorema de Hamilton-Cayley nos dice que el polinomio mínimo es siempre un divisor del característico.

```
sage: # Construiremos una matriz C que tenga autovalores y no sea trivial
sage: A = diagonal_matrix(QQ,[4, 4, -5, 1, 12])
sage: B = random_matrix(ZZ, 5, x = 5)
sage: C = B * A * B^(-1); C

[-1089/268    662/67  -415/268  -411/134   267/134]
[-1807/134    731/67   707/134   -129/67    477/67]
[-1597/268    620/67   237/268  -207/134   111/134]
[-1831/134    736/67   263/134     79/67    381/67]
[-1573/134    967/67  -391/134   -147/67    475/67]
sage: f = C.charpoly()
sage: g = C.minpoly()
sage: # El polinomio minimo siempre es divisor del caracteristico
sage: g.divides(f)
True
sage: # Ademas tienen las mismas raices (aunque distinta multiplicidad)
sage: f.roots(); g.roots()
[(12, 1), (1, 1), (-5, 1), (4, 2)]
[(12, 1), (4, 1), (1, 1), (-5, 1)]
```

El conjunto de autovalores de una matriz se obtiene con el comando

`.eigenvalues()`

Para obtener los subespacios vectoriales $V_\lambda$, utilizamos el comando

`.eigenspaces_right()`

```
sage: # Seguimos con las definiciones anteriores
sage: C.eigenvalues()
[12, 1, -5, 4, 4]
sage: # El autovalor doble aparece dos veces
sage: factor(C.charpoly())
(x - 12) * (x - 1) * (x + 5) * (x - 4)^2
sage: C.eigenspaces_right()

[
(12, Vector space of degree 5 and dimension 1 over Rational Field
User basis matrix:
[1 2 1 2 2]),
(1, Vector space of degree 5 and dimension 1 over Rational Field
User basis matrix:
[  1 3/4 1/2   1 3/4]),
(-5, Vector space of degree 5 and dimension 1 over Rational Field
User basis matrix:
[  1   0   1 2/3 4/3]),
(4, Vector space of degree 5 and dimension 2 over Rational Field
User basis matrix:
[   1    0   -1 -1/2  5/2]
[   0    1    2  3/4 -9/4])
]
sage: # Comprobemos que el primero es efectivamente un autovector de valor 12
sage: C * vector([1,2,1,2,2])
(12, 24, 12, 24, 24)
```

# Capítulo 8

# Teoría elemental de números

## 8.1. Los anillos $\mathbb{Z}_n$

A comienzos del siglo 19 Gauss se dió cuenta de que en la Teoría de Números era fundamental el estudio de las congruencias. Desde un punto de vista moderno, esto equivale a comprender la estructura de los anillos $\mathbb{Z}_n$. El caso en que $n$ es primo es especial, ya que en este caso el anillo es un cuerpo.

Una vez construido el anillo en cuestión, todas las operaciones aritméticas se realizan con los operadores habituales. Es importante no confundir los elementos de los anillos cociente $\mathbb{Z}_n$ con los elementos del anillo de los enteros. Cuando tengamos dudas le podemos preguntar al programa.

La construcción del anillo $\mathbb{Z}_n$ se realiza con la instrucción

```
IntegerModRing(n)
```

```
sage: R = IntegerModRing(23)  # Creamos el anillo Z_23 y lo llamamos R
sage: R
Ring of integers modulo 23
sage: a = R(12)  # Asi se crean los elementos del anillo
sage: type(a)
<type 'sage.rings.integer_mod.IntegerMod_int'>
sage: parent(a)
Ring of integers modulo 23
sage: a.base_ring()
```

```
Ring of integers modulo 23
sage: b = R(14)
sage: a + b, a - b, a * b, a / b
(3, 21, 7, 14)
sage: a^(-1)  # El inverso de a, que en este caso existe
2
sage: a * R(2)
1
sage: a^2345
6
```

Si el anillo no tiene módulo primo, existen elementos invertibles (los primos con el módulo) y también elementos que no tienen inverso. Para averiguarlo tenemos el método

`.is_unit()`

Una vez que sabemos que es invertible podremos calcular el inverso (que es único).

De la misma forma, no todo elemento del anillo cociente es un cuadrado

`.is_square()`

Si un elemento es un cuadrado se puede extraer su raíz cuadrada con el método `.sqrt()`. Puede ocurrir que la raíz cuadrada no sea única. Si queremos obtenerlas todas debemos añadir la opción `all = True`

```
sage: R = IntegerModRing(88)
sage: a = R(4)
sage: a.is_unit()
False
sage: a.is_square()
True
sage: a.sqrt()   # Una raíz cuadrada
2
sage: a.sqrt(all = True)  # Todas las raices cuadradas
[2, 42, 46, 86]
sage: R(2)^2, R(42)^2, R(46)^2, R(86)^2
(4, 4, 4, 4)
```

## 8.2. Funciones multiplicativas

Las funciones multiplicativas se utilizan habitualmente en Teoría de Números y están caracterizadas por satisfacer la siguiente propiedad

$$f(nm) = f(n)f(m) \quad \text{si } n \text{ y } m \text{ son primos entre si}$$

Por ello, si conocemos $f(p)$ y $f(q)$, siendo $p$ y $q$ números primos, podemos conocer $f(pq)$. Por el teorema fundamental de la aritmética, si se conoce $f(p^k)$ para cualquier primo, podemos conocer $f(n)$ para cualquier $n$.

Dado un número $n$, consideramos todos sus divisores positivos, incluyendo la unidad y él mismo. Posteriormente sumamos todos los divisores. El número así obtenido es costumbre denotarlo $\sigma(n)$. Hemos construido así la función $\sigma : \mathbb{N} \to \mathbb{N}$. Esta función es multiplicativa y es sencillo calcular su valor para las potencias de primos.

sigma(n)

```
sage: sigma(8)
15
sage: divisors(8)
[1, 2, 4, 8]
sage: 1 + 2 + 4 + 8
15
sage: # Calculemos sigma sobre un numero primo
sage: sigma(7)
8
sage: # Calculemos sigma sobre potencias de primos
sage: sigma(7), sigma(7^2), sigma(7^3), sigma(7^4)
(8, 57, 400, 2801)
sage: # Para obtener estos numeros debemos sumar progresiones geometricas
sage: sigma(7^4), 1 + 7 + 7^2 + 7^3 + 7^4
(2801, 2801)
sage: # Comprobemos que es multiplicativa
sage: n = 2^5 * 3^6
sage: m = 5^4 * 7^3
sage: n,m
```

```
(23328, 214375)
sage: gcd(n,m)
1
sage: sigma(n*m), sigma(n) * sigma(m)
(21511551600, 21511551600)
sage: # Los numeros perfectos son aquellos que sigma(n) = 2n
sage: sigma(6), 2 * 6
(12, 12)
sage: sigma(28), 2 * 28
(56, 56)
```

La función $\tau$ cuenta el número de divisores positivos, incluyendo la unidad y el mismo número. Es también una función multiplicativa.

number_of_divisors(n)

```
sage: divisors(90)
[1, 2, 3, 5, 6, 9, 10, 15, 18, 30, 45, 90]
sage: number_of_divisors(90)
12
sage: # Sobre los numeros primos la funcion tau es sencilla
sage: number_of_divisors(23)
2
sage: # Tambien es sencilla sobre potencias de primos
sage: number_of_divisors(7^6), number_of_divisors(7^201)
(7, 202)
sage: # Veamos que es multiplicativa
sage: n = 2^4 * 3^7
sage: m = 5^3 * 11^5
sage: n, m, gcd(n,m)
(34992, 20131375, 1)
sage: number_of_divisors(n * m)
960
sage: number_of_divisors(n) * number_of_divisors(m)
960
```

La función $\mu$ de Möbius es un poco más complicada de definir. Tomamos un número natural $n$ y lo descomponemos en factores primos. Si algún factor primo aparece con exponente 2 o superior, entonces $\mu(n) = 0$. Si todos los factores que aparecen en $n$ son de grado 1 entonces los contamos. Si hay un número par de factores primos, entonces $\mu(n) = 1$. Si hay un número impar de factores entonces $\mu(n) = -1$. Es sencillo comprobar que si $\mu(n) = 0$ entonces existe un cuadrado $r^2$ que divide a $n$. El inverso también es correcto: Si algún cuadrado divide a $n$ entonces $\mu(n) = 0$. También es una función multiplicativa y en cierto sentido es la más importante de todas, debido a la fórmula de inversión de Möbius (ver cualquier libro de Teoría de Números).

moebius()

Dado un número natural $n$ consideramos todos los números menores que él. De estos algunos son primos con $n$ y otros no. Consideramos únicamente los que son primos con $n$ y los contamos. Dicho número se denota $\phi(n)$. El primero en utilizar esta función de un modo consciente fue Euler, y hoy en dia se llama la función phi de Euler. Como hemos visto en la primera sección de este capítulo, el número $\phi(n)$ coincide con el conjunto de unidades del anillo $\mathbb{Z}_n$.

euler_phi()

```
sage: moebius(2^2*3*5)  # Un factor repetido
0
sage: moebius(3*5*7)  # Un numero impar de factores, sin repeticion
-1
sage: moebius(3*5*11*23)  # Un numero par de factores sin repeticion
1
sage: euler_phi(7)  # Sobre los numeros primos es sencilla
6
sage: euler_phi(23)
22
sage: # Ahora sobre potencias de primos
sage: euler_phi(7^2)
42
sage: euler_phi(7^3)
```

```
294
sage: euler_phi(776556979878787888899999876)
384456325242775348203782 40
```

# Apéndice A

# Cuaterniones

El invento o descubrimiento de los cuaterniones se debe al matemático irlandés Hamilton. Una vez que fue asimilado por la comunidad matemática que los números complejos eran simplemente pares de números reales con unas reglas especiales de multiplicación, Hamilton intentó realizar lo mismo con ternas de números. Fracasó, naturalmente, puesto que hoy es conocido que no puede existir ningun cuerpo de grado 3 sobre los números reales. Sin embargo, en un momento de inspiración, descubrió las reglas para multiplicar cuaternas de números. Estas reglas cumplían todas las propiedades, salvo la propiedad conmutativa de la multiplicación. A día de hoy sabemos construir, a semejanza de lo hecho por Hamilton, extensiones de cuerpos (no necesariamente del cuerpo real) de grado 4 y que cumplen propiedades similares. Nosotros trabajaremos con el Álgebra cuaternionica más simple, que es aquella en la que los generadores $i, j, k$ cumplen $i^2 = -1$, $j^2 = -1$ e $ij = k$, construida sobre el cuerpo $\mathbb{Q}$.

La orden para crear el Álgebra cuaternionica es

```
A.<i,j,k> = QuaternionAlgebra(QQ, -1, -1)
```

Una vez hecho esto, cualquier cuaternión se puede escribir como una combinación lineal de la unidad y de las tres letras. Las operaciones, como siempre, se realizan con los operadores habituales. Debemos tener en cuenta que la multiplicación no es conmutativa.

```
sage: A.<i,j,k> = QuaternionAlgebra(QQ, -1, -1)
sage: a = 4 + 5*i + 8*j + 2*k
sage: type(a)
```

```
<class 'sage.algebras.quaternion_algebra_element.QuaternionAlgebraElement'>
sage: b = 8 + 2*i -9*j + 3*k
sage: a * b
88 + 90*i + 17*j - 33*k
sage: b * a
88 + 6*i + 39*j + 89*k
sage: a + 7*b
60 + 19*i - 55*j + 23*k
sage: b^4
-23164 - 1920*i + 8640*j - 2880*k
sage: c = 1 / b  # El inverso de b
sage: b * c
1
```

El conjugado de un cuaternión $a + bi + cj + dk$ es $a - bi - cj - dk$

`.conjugate()`

La norma de un cuaternión es su módulo en el espacio de cuatro dimensiones. Coincide con la multiplicación de un cuaternión por su conjugado (como en el caso complejo)

`.reduced_norm()`

La parte real (también llamada escalar) del cuaternión anterior es $a$ y la parte vectorial $bi + cj + dk$. Un cuaternión se dice que es imaginario puro (o simplemente que es un cuaternión puro), si su parte real es nula, por analogía con el caso complejo.

`.scalar_part(), .pure_part()`

```
sage: a = 4 + 5*i + 8*j + 2*k
sage: a.reduced_norm()
109
sage: a * a.conjugate()
109
sage: a.conjugate() * a  # Un cuaternion y su conjugado conmutan
109
```

```
sage: a.scalar_part()
4
sage: a.pure_part()
5*i + 8*j + 2*k
```

# Apéndice B

# Grupos abelianos finitos

Un grupo cíclico es aquel generado por un solo elemento. Para cualquier número natural $n$, existe un único (salvo isomorfismos) grupo cíclico de orden $n$, que denotaremos $C_n$. Dicho grupo es isomorfo al grupo aditivo $\mathbb{Z}_n$. Además de estos grupos finitos existe un grupo cíclico infinito, $C_\infty$, que es isomorfo a $\mathbb{Z}$. Naturalmente todo grupo cíclico es abeliano. También es claro que todo producto directo de grupos cíclicos es abeliano. El Teorema de estructura de los grupos abelianos finitos afirma esto agota el conjunto de grupos abelianos finitos: Todo grupo abeliano finito es isomorfo a un producto directo de grupos cíclicos.

El comando

```
G.<a,b,c> = AbelianGroup([n,m,k])
```

crea un grupo abeliano, de nombre `G`, isomorfo a $C_n \times C_m \times C_k$. El generador de $C_n$ es `a`, el de $C_m$ es `b` y el de $C_k$ es `c`. La misma construcción se puede realizar con cualquier número de grupos cíclicos. Todo elemento del grupo se puede escribir como producto de potencias de los generadores. La operación en el grupo se realiza con el asterisco. Escribiendo un único elemento en la lista construimos los grupos cíclicos.

Dado un grupo, el método

```
.list()
```

nos da una lista con todos los elementos del grupo.

Si solamente queremos saber cuantos elementos tiene el grupo podemos utilizar

```
.order()
```

El programa puede extraer, de modo aleatorio, un elemento del grupo

```
.random_element()
```

```
sage: G.<a,b> = AbelianGroup([2,3]); G
Multiplicative Abelian Group isomorphic to C2 x C3
sage: G.order()
6
sage: G.list()
[1, b, b^2, a, a*b, a*b^2]
sage: G.<a,b,c> = AbelianGroup([67,980,23]); G
Multiplicative Abelian Group isomorphic to C67 x C980 x C23
sage: h = G.random_element(); h
a^10*b^449*c^22
sage: g = G.random_element(); g
a^62*b^391*c^20
sage: h * g, g * h  # Deben conmutar
(a^5*b^840*c^19, a^5*b^840*c^19)
sage: i = g^(-1); i
a^5*b^589*c^3
sage: i * g
1
```

Para crear un subgrupo $H$ de un grupo $G$ basta con dar una lista de generadores del subgrupo. Esto es, si damos una lista de elementos del grupo, el subgrupo generado es el mínimo subgrupo que contiene a todos los elementos. En Sage se utiliza el método

```
.subgroup(lista)
```

donde `lista` es una lista (y por lo tanto debe ir entre corchetes, aunque tenga un solo elemento) de generadores del subgrupo.

Dado un elemento $g$ del grupo $G$, dicho elemento genera un subgrupo. Como estamos suponiendo que $G$ es finito, el subgrupo generado por $g$ también debe serlo. Al número de elementos de este subgrupo (también llamado orden del subgrupo) se le denomina orden

del elemento $g$. Coincide con el menor entero positivo que cumple la igualdad $g^n = 1$, que es la definición clásica de este concepto. El método para saber el orden de un elemento es

```
.order()
```

pero en este caso debe actuar sobre un elemento del grupo.

```
sage: G.<a,b,c,d> = AbelianGroup([45,78,23,678]); G
Multiplicative Abelian Group isomorphic to C45 x C78 x C23 x C678
sage: g = G.random_element()
sage: h = G.random_element()
sage: g, h
(a^4*b^76*c^21*d^548, a^22*b^75*c^17*d^129)
sage: H = G.subgroup([a,b]); H

Multiplicative Abelian Group isomorphic to C2 x C3 x C5 x C9 x C13,
which is the subgroup of
Multiplicative Abelian Group isomorphic to C45 x C78 x C23 x C678
generated by [a, b]
sage: Hg = G.subgroup([g])
sage: Hg.order(), g.order()
(1520415, 1520415)
```