Sage Tutorial

Versión 10.4.rc1

The Sage Development Team

27 de junio de 2024

Índice general

Sage es un software matemático libre y de código abierto que apoya la investigación y la enseñanza en álgebra, geometría, teoría de números, criptografía, computación numérica y áreas relacionadas. Tanto el modelo de desarrollo como la tecnología en Sage se distinguen por un énfasis extremadamente fuerte en la apertura, comunidad, cooperación y colaboración: estamos construyendo el carro, no reinventando la rueda. La meta de Sage en su conjunto es la de crear una alternativa viable, libre y de código abierto a Maple, Mathematica, Magma y MATLAB.

Este tutorial es la mejor manera de familiarizarse con Sage en unas cuantas horas. Puedes leerlo en versión HTML o PDF, o desde el notebook de Sage (haz click en `Help`, luego haz click en `Tutorial` para trabajar interactivamente en el tutorial desde dentro de Sage).

CAPÍTULO 1

Introducción

Completar este tutorial debería llevarte unas 3 o 4 horas. Puedes leerlo en versión HTML o PDF, o desde el notebook (interfaz interactiva vía web) de Sage (haz click en `Help`, luego haz click en `Tutorial` para trabajar interactivamente en el tutorial desde dentro de Sage).

Aunque gran parte de Sage está implementada usando el lenguaje de programación Python, no es necesario ningún conocimiento previo de Python para poder leer este tutorial. En algún punto seguramente querrás aprender Python (¡un lenguaje muy divertido!), y hay muchos recursos gratuitos excelentes para hacerlo, La Guía Para Principiantes De Python [PyB] enumera muchas opciones. Si tan solo quieres experimentar ligeramente con Sage, este tutorial es el lugar justo para empezar. Por ejemplo:

```
sage: 2 + 2
4
sage: factor(-2007)
-1 * 3^2 * 223

sage: A = matrix(4,4, range(16)); A
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
[12 13 14 15]

sage: factor(A.charpoly())
x^2 * (x^2 - 30*x - 80)

sage: m = matrix(ZZ,2, range(4))
sage: m[0,0] = m[0,0] - 3
sage: m
[-3  1]
[ 2  3]

sage: E = EllipticCurve([1,2,3,4,5]);
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5
over Rational Field
```

(continúe en la próxima página)

(proviene de la página anterior)

```
sage: E.anlist(10)
[0, 1, 1, 0, -1, -3, 0, -1, -3, -3, -3]
sage: E.rank()
1

sage: k = 1/(sqrt(3)*I + 3/4 + sqrt(73)*5/9); k
36/(20*sqrt(73) + 36*I*sqrt(3) + 27)
sage: N(k)
0.165495678130644 - 0.0521492082074256*I
sage: N(k,30)       # 30 "bits"
0.16549568 - 0.052149208*I
sage: latex(k)
\frac{36}{20 \, \sqrt{73} + 36 i \, \sqrt{3} + 27}
```

1.1 Instalación

Si no tienes instalado Sage en tu computador y sólo quieres probar algunos comandos, usa la versión en línea en http://sagecell.sagemath.org.

Mira la Guía De Instalación Para Sage en la sección de documentación de la página web principal de [Sage] para obtener instrucciones sobre cómo instalar Sage en tu computador. Aquí hacemos simplemente dos comentarios:

1. El archivo de descarga de Sage viene con «baterías incluidas». En otras palabras, aunque Sage utiliza Python, IPython, PARI, GAP, Singular, Maxima, NTL, GMP, etc., no necesitas instalarlos por separado pues ya están incluidos con la distribución de Sage. Sin embargo, para utilizar ciertas características de Sage, por ejemplo, Macaulay o KASH, debes tener los programas pertinentes ya instalados en tu computador.
2. La versión binaria precompilada de Sage (que se encuentra en el sitio web de Sage) puede ser más rápida y fácil de instalar que la versión en código fuente. Sólo desempaqueta el archivo y ejecuta `sage`.
3. Si quieres utilizar el paquete SageTeX (el cual te permite insertar los resultados de tus cálculos con Sage en un archivo LaTeX), necesitarás hacerle conocer SageTeX a tu distribución de TeX. Para hacer esto, consulta la sección «Haciendo que TeX conozca a SageTeX» en la guía de instalación de Sage Sage installation guide (Este enlace debería llevarte a tu copia local de la guía de instalación). Es bastante sencillo: sólo necesitas establecer una variable de entorno o copiar un solo archivo en un directorio en el que TeX va a buscar.

   La documentación para usar SageTeX se encuentra en `$SAGE_ROOT/venv/share/texmf/tex/latex/sagetex/`, donde «`$SAGE_ROOT`» se refiere al directorio donde Sage está instalado – por ejemplo, `/opt/sage-9.6`.

1.2 Formas de usar Sage

Puedes usar Sage de varias maneras.

- **Interfaz gráfico del Notebook:** iniciar `sage -n jupyter`; leer Jupyter documentation on-line,
- **Línea de comandos interactiva:**,
- **Programas:** Escribiendo programas compilados e interpretados en Sage y
- **Scripts:** Escribiendo scripts (archivos de órdenes) independientes en Python que utilizan la biblioteca Sage.

1.3 Metas a largo plazo de Sage

- **Útil**: La audiencia a la que está destinado Sage son los estudiantes de matemáticas (desde la secundaria hasta la universidad), profesores y matemáticos (para la investigación). El objetivo es proveer un software que pueda usarse para explorar y experimentar con construcciones matemáticas en álgebra, geometría, teoría de números, cálculo, computación numérica, etc. Sage facilita la experimentación interactiva con objetos matemáticos.
- **Eficiente:** Queremos que sea rápido. Sage utiliza software maduro y altamente optimizado: GMP, PARI, GAP y NTL, por lo que es muy rápido en ciertas operaciones.
- **Libre y de código abierto:** El código fuente debe ser legible y libremente disponible, de modo que los usuarios puedan entender qué está haciendo realmente el sistema y así poder extenderlo fácilmente. Tal como los matemáticos logran un entendimiento más profundo de un teorema al leerlo cuidadosamente o, por lo menos, al echarle una ojeada a la prueba, la gente que efectúa cálculos debe ser capaz de comprender cómo funcionan los cálculos leyendo el código fuente documentado. Si utilizas Sage para hacer cálculos en un artículo que vas a publicar, puedes estar seguro que tus lectores siempre tendrán libre acceso a Sage y a todo su código fuente, y hasta se te permite archivar y re-distribuir la versión de Sage que usaste.
- **Fácil de compilar:** Sage tiene que ser fácil de compilar desde el código fuente para los usuarios de Linux, OS X y Windows. Esto provee a los usuarios de una mayor flexibilidad para que modifiquen el sistema.
- **Cooperación con otros programas:** Sage debe proveer interfaces robustos a la mayoría de sistemas algebraicos de cómputo, incluyendo PARI, GAP, Singular, Maxima, KASH, Magma, Maple y Mathematica. Sage pretende unificar y extender el software matemático existente.
- **Bien documentado:** Debemos proveer un tutorial, una guía de programación, un manual de referencia y documentos sobre cómo hacer cosas específicas, con numerosos ejemplos y discusiones de las bases matemáticas.
- **Extensible:** Debe ser posible definir nuevos tipos de datos o derivar de tipos incorporados y utilizar código escrito en una amplia gama de lenguajes.
- **Fácil de usar**: Debe de ser fácil comprender cual funcionalidad se ha provisto para un objeto dado y examinar la documentación y el código fuente, así como alcanzar un alto nivel de soporte al usuario.

CAPÍTULO 2

Un Tour Guiado

Esta sección es un tour guiado de las opciones disponibles en Sage. Para muchos más ejemplos, véase «Construcciones En Sage», con la cual se intenta responder a la pregunta general «¿Cómo hago para construir …?». Véase también el «Manual De Referencia De Sage», el cual tiene miles de ejemplos. Observa también que puedes trabajar interactivamente con este tour en el notebook de Sage haciendo click en el enlace `Help`.

(Si estás viendo el tutorial en el notebook de Sage, presiona `shift-enter` para evaluar cualquier celda de entrada de datos. Incluso puedes editar la entrada de datos antes de presionar `shift-enter`. En algunas Macs tendrías que presionar `shift-return` en lugar de `shift-enter`.)

2.1 Asignación, Igualdad y Aritmética

Con algunas excepciones menores, Sage utiliza el lenguaje de programación Python, de modo que la mayoría de los libros introductorios sobre Python te ayudarán a aprender Sage.

Sage utiliza = para la asignación. Utiliza ==, <=, >=, < y > para la comparación:

```
sage: a = 5
sage: a
5
sage: 2 == 2
True
sage: 2 == 3
False
sage: 2 < 3
True
sage: a == 5
True
```

Sage provee todo lo relacionado con las operaciones matemáticas básicas:

```
sage: 2**3    #  ** significa exponente
8
```

(continúe en la próxima página)

(proviene de la página anterior)

```
sage: 2^3     #  ^ es un sinónimo de ** (diferente de Python)
8
sage: 10 % 3  #  para argumentos enteros, % significa mod, es decir, resíduo
1
sage: 10/4
5/2
sage: 10//4   #  para argumentos enteros, // devuelve el cociente de enteros
2
sage: 4 * (10 // 4) + 10 % 4 == 10
True
sage: 3^2*4 + 2%5
38
```

El cálculo de una expresión tal como `3^2*4 + 2%5` depende de el órden en que las operaciones son aplicadas.

Sage también provee muchas funciones matemáticas conocidas; he aquí solo unos cuantos ejemplos:

```
sage: sqrt(3.4)
1.84390889145858
sage: sin(5.135)
-0.912021158525540
sage: sin(pi/3)
1/2*sqrt(3)
```

Como demuestra el último ejemplo, algunas expresiones matemáticas devuelven valores "exactos", en lugar de aproximaciones numéricas. Para obtener una aproximación numérica, utilice la función `n` o el método `n` (ámbas tienen un nombre más largo, `numerical_approx`, y la función `N` es la misma que `n`)). Éstas toman argumentos opcionales `prec`, que es el número requerido de bits de precisión, y `digits`, que es el número requerido de digitos decimales de precisión; el número predeterminado es de 53 bits de precisión.

```
sage: exp(2)
e^2
sage: n(exp(2))
7.38905609893065
sage: sqrt(pi).numerical_approx()
1.77245385090552
sage: sin(10).n(digits=5)
-0.54402
sage: N(sin(10),digits=10)
-0.5440211109
sage: numerical_approx(pi, prec=200)
3.1415926535897932384626433832795028841971693993751058209749
```

Python es un lenguaje de tipado dinámico, de modo que el valor referido por cada variable tiene un tipo asociado. Pero una variable dada puede contener valores de cualquier tipo Python dentro de un ámbito dado:

```
sage: a = 5   # a es un entero
sage: type(a)
<class 'sage.rings.integer.Integer'>
sage: a = 5/3  # ahora es un número racional
sage: type(a)
<class 'sage.rings.rational.Rational'>
sage: a = 'hello'  # ahora es una cadena
sage: type(a)
<... 'str'>
```

El lenguaje de programación C, que es un lenguaje de tipado estático, es muy diferente; una variable declarada como int

solo puede contener un int en su ámbito.

2.2 Obteniendo Ayuda

Sage posee una extensa documentación incorporada, accesible con solo teclear el nombre de una función o una constante (por ejemplo), seguido de un signo de interrogación:

```
sage: tan?
Type:        <class 'sage.calculus.calculus.Function_tan'>
Definition:  tan( [noargspec] )
Docstring:

    The tangent function

    EXAMPLES:
        sage: tan(pi)
        0
        sage: tan(3.1415)
        -0.0000926535900581913
        sage: tan(3.1415/4)
        0.999953674278156
        sage: tan(pi/4)
        1
        sage: tan(1/2)
        tan(1/2)
        sage: RR(tan(1/2))
        0.546302489843790
sage: log2?
Type:        <class 'sage.functions.constants.Log2'>
Definition:  log2( [noargspec] )
Docstring:

    The natural logarithm of the real number 2.

    EXAMPLES:
        sage: log2
        log2
        sage: float(log2)
        0.69314718055994529
        sage: RR(log2)
        0.693147180559945
        sage: R = RealField(200); R
        Real Field with 200 bits of precision
        sage: R(log2)
        0.69314718055994530941723212145817656807550013436025525412068
        sage: l = (1-log2)/(1+log2); l
        (1 - log(2))/(log(2) + 1)
        sage: R(l)
        0.18123221829928249948761381864650311423330609774776013488056
        sage: maxima(log2)
        log(2)
        sage: maxima(log2).float()
        .6931471805599453
        sage: gp(log2)
        0.69314718055994530941723212145817656807   # 64-bit
        0.69314718055994530941723212145817656807   # 64-bit
```

(continúe en la próxima página)

(proviene de la página anterior)

```
sage: sudoku?
File:        sage/local/lib/python2.5/site-packages/sage/games/sudoku.py
Type:        <... 'function'>
Definition:  sudoku(A)
Docstring:

    Solve the 9x9 Sudoku puzzle defined by the matrix A.

    EXAMPLE:
        sage: A = matrix(ZZ,9,[5,0,0, 0,8,0, 0,4,9, 0,0,0, 5,0,0,
    0,3,0, 0,6,7, 3,0,0, 0,0,1, 1,5,0, 0,0,0, 0,0,0, 0,0,0, 2,0,8, 0,0,0,
    0,0,0, 0,0,0, 0,1,8, 7,0,0, 0,0,4, 1,5,0,   0,3,0, 0,0,2,
    0,0,0, 4,9,0, 0,5,0, 0,0,3])
        sage: A
        [5 0 0 0 8 0 0 4 9]
        [0 0 0 5 0 0 0 3 0]
        [0 6 7 3 0 0 0 0 1]
        [1 5 0 0 0 0 0 0 0]
        [0 0 0 2 0 8 0 0 0]
        [0 0 0 0 0 0 0 1 8]
        [7 0 0 0 0 4 1 5 0]
        [0 3 0 0 0 2 0 0 0]
        [4 9 0 0 5 0 0 0 3]
        sage: sudoku(A)
        [5 1 3 6 8 7 2 4 9]
        [8 4 9 5 2 1 6 3 7]
        [2 6 7 3 4 9 5 8 1]
        [1 5 8 4 6 3 9 7 2]
        [9 7 4 2 1 8 3 6 5]
        [3 2 6 7 9 5 4 1 8]
        [7 8 2 9 3 4 1 5 6]
        [6 3 5 1 7 2 8 9 4]
        [4 9 1 8 5 6 7 2 3]
```

Sage también provee "Autocompletado con el tabulador": teclea las primeras letras de una función y luego oprime la tecla del tabulador. Por ejemplo, si tecleas `ta` seguido por `TAB`, Sage imprimirá `tachyon, tan, tanh, taylor`. Esto proporciona una buena manera de encontrar los nombres de funciones y otras estructuras en Sage.

2.3 Funciones, Indentación Y Conteo

Para definir una nueva función en Sage, utilice el comando `def` y el signo de dos puntos después de la lista de nombres de variable. Por ejemplo:

```
sage: def is_even(n):
....:     return n % 2 == 0
....:
sage: is_even(2)
True
sage: is_even(3)
False
```

Nota: Dependiendo de la versión del tutorial que estás leyendo, puede que veas puntos `....:` en la segunda línea de este ejemplo. No los incluyas; son solo para enfatizar que el código está indentado. Siempre que este sea el caso, presiona [Return/Enter] una vez al final del bloque para insertar una línea en blanco y concluir la definición de la función.

No tienes que especificar los tipos de ninguno de los argumentos de entrada. Puedes especificar múltiples entradas, cada una de las cuales puede tener un valor predeterminado opcional. Por ejemplo, la función de abajo tiene un valor predeterminado `divisor=2` si no se especifica el valor de `divisor`.

```
sage: def is_divisible_by(number, divisor=2):
....:     return number % divisor == 0
sage: is_divisible_by(6,2)
True
sage: is_divisible_by(6)
True
sage: is_divisible_by(6, 5)
False
```

También puedes especificar explícitamente una o ambas de las entradas cuando llames a la función; si especificas las entradas explícitamente, puedes darlas en cualquier órden:

```
sage: is_divisible_by(6, divisor=5)
False
sage: is_divisible_by(divisor=2, number=6)
True
```

En Python, los bloques de código no se encierran entre llaves o bloques begin…end como en muchos otros lenguajes. En vez de ello, los bloques de código se indican por medio de la indentación, la cual se debe agrupar con exactitud. Por ejemplo, el siguiente es un error de sintáxis porque la declaración `return` no está indentada al mismo nivel que las otras líneas por encima de ella.

```
sage: def even(n):
....:     v = []
....:     for i in range(3,n):
....:         if i % 2 == 0:
....:             v.append(i)
....:    return v
Syntax Error:
       return v
```

Si arreglas la indentación, la función se ejecutará:

```
sage: def even(n):
....:     v = []
....:     for i in range(3,n):
....:         if i % 2 == 0:
....:             v.append(i)
....:     return v
sage: even(10)
[4, 6, 8]
```

El punto y coma no es necesario al final de las líneas. Una línea termina, en muchos casos, por un carácter de nueva línea. Sin embargo, puedes poner múltiples declaraciones en una línea, separadas por punto y coma:

```
sage: a = 5; b = a + 3; c = b^2; c
64
```

Si quisieras que una simple línea de código abarque multiples líneas, utiliza una barra invertida como terminación:

```
sage: 2 + \
....: 3
5
```

En Sage, se cuenta iterando sobre un rango de enteros. Por ejemplo, la primer línea de abajo es exactamente igual a `for(i=0; i<3; i++)` en C++ o Java:

```
sage: for i in range(3):
....:     print(i)
0
1
2
```

La primer línea de abajo es igual a `for(i=2;i<5;i++)`.

```
sage: for i in range(2,5):
....:     print(i)
2
3
4
```

El tercer argumento controla el incremento, de modo que lo siguiente es igual a `for(i=1;i<6;i+=2)`.

```
sage: for i in range(1,6,2):
....:     print(i)
1
3
5
```

A menudo, querrás crear una tabla para presentar números que has calculado utilizando Sage. Una manera sencilla de hacer esto es usando el formateado de cadenas. Abajo, creamos tres columnas, cada una con un ancho exácto de 6 caracteres y hacemos una tabla de cuadrados y cubos.

```
sage: for i in range(5):
....:     print('%6s %6s %6s' % (i, i^2, i^3))
     0      0      0
     1      1      1
     2      4      8
     3      9     27
     4     16     64
```

La estructura de datos más básica en Sage es la lista, la cual es – como sugiere su nombre – solo una lista de objetos arbitrarios. Por ejemplo, el comando `range` que hemos usado crea una lista:

```
sage: list(range(2,10))
[2, 3, 4, 5, 6, 7, 8, 9]
```

He aquí una lista más complicada:

```
sage: v = [1, "hello", 2/3, sin(x^3)]
sage: v
[1, 'hello', 2/3, sin(x^3)]
```

El indexado de una lista comienza en el cero, como en muchos lenguajes de programación.

```
sage: v[0]
1
sage: v[3]
sin(x^3)
```

La función `len(v)` devuelve la longitud de `v`. Utiliza `v.append(obj)` para añadir un nuevo objeto al final de `v`, y utiliza `del v[i]` para borrar el $i-simo$ elemento de `v`:

```
sage: len(v)
4
sage: v.append(1.5)
sage: v
[1, 'hello', 2/3, sin(x^3), 1.50000000000000]
sage: del v[1]
sage: v
[1, 2/3, sin(x^3), 1.50000000000000]
```

Otra estructura de datos importante es el diccionario (o array asociativo). Funciona como una lista, excepto que puede ser indexado con casi cualquier objeto (los índices deben ser immutables):

```
sage: d = {'hi':-2,  3/8:pi,   e:pi}
sage: d['hi']
-2
sage: d[e]
pi
```

También puedes definir nuevos tipos de datos usando clases. El encapsulado de objetos matemáticos con clases es una técnica potente que puede ayudar a simplificar y organizar tus programas en Sage. Abajo, definimos una clase que representa la lista de enteros positivos pares hasta `n`; se deriva de el tipo básico `list`.

```
sage: class Evens(list):
....:     def __init__(self, n):
....:         self.n = n
....:         list.__init__(self, range(2, n+1, 2))
....:     def __repr__(self):
....:         return "Even positive numbers up to n."
```

El método `__init__` se llama para inicializar al objeto cuando es creado; el método `__repr__` imprime el objeto. Llamamos al método constructor de listas en la segunda línea del método `__init__`. A continuación, creamos un objeto de clase `Evens`:

```
sage: e = Evens(10)
sage: e
Even positive numbers up to n.
```

Observe que `e` se imprime usando el método `__repr__` que hemos definido. Para ver la lista subyacente de números, utilice la función `list`:

```
sage: list(e)
[2, 4, 6, 8, 10]
```

También podemos acceder al atributo `n` o tratar a `e` como una lista.

```
sage: e.n
10
sage: e[2]
6
```

2.4 Álgebra Y Cálculo Básicos

Sage puede efectuar cómputos relacionados al algebra y cálculo básicos: por ejemplo, encontrar soluciones de ecuaciones, diferenciación, integración y transformadas de Laplace. Véa la documentación «Construcciones En Sage» para más ejemplos.

2.4.1 Resolviendo Ecuaciones

Resolviendo Ecuaciones De Manera Exacta

La función `solve` resuelve ecuaciones. Para usarla, primero no olvides especificar algunas variables. Los argumentos de `solve` son una ecuación (o un sistema de ecuaciones), junto con las variables a resolver:

```
sage: x = var('x')
sage: solve(x^2 + 3*x + 2, x)
[x == -2, x == -1]
```

Puedes resolver ecuaciones en una variable respecto de las demás:

```
sage: x, b, c = var('x b c')
sage: solve([x^2 + b*x + c == 0],x)
[x == -1/2*b - 1/2*sqrt(b^2 - 4*c), x == -1/2*b + 1/2*sqrt(b^2 - 4*c)]
```

Puedes también resolver ecuaciones en varias variables:

```
sage: x, y = var('x, y')
sage: solve([x+y==6, x-y==4], x, y)
[[x == 5, y == 1]]
```

El siguiente ejemplo del uso de Sage para resolver un sistema de ecuaciones no-lineales fue proporcionado por Jason Grout: primero, resolvemos el sistema simbólicamente:

```
sage: var('x y p q')
(x, y, p, q)
sage: eq1 = p+q==9
sage: eq2 = q*y+p*x==-6
sage: eq3 = q*y^2+p*x^2==24
sage: solve([eq1,eq2,eq3,p==1],p,q,x,y)
[[p == 1, q == 8, x == -4/3*sqrt(10) - 2/3, y == 1/6*sqrt(10) - 2/3], [p == 1, q == 8,
↪ x == 4/3*sqrt(10) - 2/3, y == -1/6*sqrt(10) - 2/3]]
```

Si queremos aproximaciones numéricas de las soluciones, podemos usar lo siguiente:

```
sage: solns = solve([eq1,eq2,eq3,p==1],p,q,x,y, solution_dict=True)
sage: [[s[p].n(30), s[q].n(30), s[x].n(30), s[y].n(30)] for s in solns]
[[1.0000000, 8.0000000, -4.8830369, -0.13962039],
 [1.0000000, 8.0000000, 3.5497035, -1.1937129]]
```

(La función `n` imprime una aproximación numérica, y el argumento es el número de bits de precisión.)

Resolviendo Ecuaciones Numéricamente

A menudo, `solve` no podrá encontrar una solución exacta para la ecuación o ecuaciones especificadas. Cuando falla, puedes usar `find_root` para encontrar una solución numérica. Por ejemplo, `solve` no devuelve nada interesante para la siguiente ecuación:

```
sage: theta = var('theta')
sage: solve(cos(theta)==sin(theta), theta)
[sin(theta) == cos(theta)]
```

Por otro lado, podemos usar `find_root` para encontrar una solución a la ecuación de arriba en el rango $0 < \theta < \pi/2$:

```
sage: phi = var('phi')
sage: find_root(cos(phi)==sin(phi),0,pi/2)
0.785398163397448...
```

2.4.2 Diferenciación, Integración, etc.

Sage sabe cómo diferenciar e integrar muchas funciones. Por ejemplo, para diferenciar $\sin(u)$ con respecto a u, haz lo siguiente:

```
sage: u = var('u')
sage: diff(sin(u), u)
cos(u)
```

Para calcular la cuarta derivada de $\sin(x^2)$:

```
sage: diff(sin(x^2), x, 4)
16*x^4*sin(x^2) - 48*x^2*cos(x^2) - 12*sin(x^2)
```

Para calcular las derivadas parciales de $x^2 + 17y^2$ con respecto a x e y, respectivamente:

```
sage: x, y = var('x,y')
sage: f = x^2 + 17*y^2
sage: f.diff(x)
2*x
sage: f.diff(y)
34*y
```

También podemos calcular integrales, tanto indefinidas como definidas. Para calcular $\int x \sin(x^2)\,dx$ y $\int_0^1 \frac{x}{x^2+1}\,dx$

```
sage: integral(x*sin(x^2), x)
-1/2*cos(x^2)
sage: integral(x/(x^2+1), x, 0, 1)
1/2*log(2)
```

Para calcular la descomposición en fracciones simples de $\frac{1}{x^2-1}$:

```
sage: f = 1/((1+x)*(x-1))
sage: f.partial_fraction(x)
-1/2/(x + 1) + 1/2/(x - 1)
```

2.4.3 Resolviendo Ecuaciones Diferenciales

Puedes usar a Sage para investigar ecuaciones diferenciales ordinarias. Para resolver la ecuación $x' + x - 1 = 0$:

```
sage: t = var('t')     # defina una variable t
sage: x = function('x')(t)    # defina x como una función de esa variable
sage: DE = diff(x, t) + x - 1
sage: desolve(DE, [x,t])
(_C + e^t)*e^(-t)
```

Esto utiliza el interfaz a Maxima de Sage [Max], por lo que el resultado puede diferir de otros resultados de Sage. En este caso, la salida nos dice que la solución general a la ecuación diferencial es $x(t) = e^{-t}(e^t + c)$.

También puedes calcular transformadas de Laplace; la transformada de Laplace de $t^2e^t - \sin(t)$ se calcula como sigue:

```
sage: s = var("s")
sage: t = var("t")
sage: f = t^2*exp(t) - sin(t)
sage: f.laplace(t,s)
-1/(s^2 + 1) + 2/(s - 1)^3
```

Veamos un ejemplo más complicado. El desplazamiento desde el punto de equilibrio de dos resortes acoplados, sujetos a una pared a la izquierda

```
|------\/\/\/\/\---|masa1|----\/\/\/\/\/----|masa2|
         resorte1                resorte2
```

está modelado por el sistema de ecuaciones diferenciales de segundo órden

$$m_1x_1'' + (k_1 + k_2)x_1 - k_2x_2 = 0$$
$$m_2x_2'' + k_2(x_2 - x_1) = 0,$$

donde m_i es la masa del objeto *i*, x_i es el desplazamiento desde el equilibrio de la masa *i*, y k_i es la constante de elasticidad del resorte *i*.

Ejemplo: Utiliza Sage para resolver el problema de arriba con $m_1 = 2, m_2 = 1, k_1 = 4, k_2 = 2, x_1(0) = 3, x_1'(0) = 0$, $x_2(0) = 3, x_2'(0) = 0$.

Solución: Toma la transformada de Laplace de la primera ecuación (con la notación $x = x_1$, $y = x_2$):

```
sage: de1 = maxima("2*diff(x(t),t, 2) + 6*x(t) - 2*y(t)")
sage: lde1 = de1.laplace("t","s"); lde1.sage()
2*s^2*laplace(x(t), t, s) - 2*s*x(0) + 6*laplace(x(t), t, s) - 2*laplace(y(t), t, s) -
↪ 2*D[0](x)(0)
```

El resultado puede ser difícil de leer, pero significa que

$$-2x'(0) + 2s^2 * X(s) - 2sx(0) - 2Y(s) + 6X(s) = 0$$

(donde la transformada de Laplace de una función en letra minúscula como $x(t)$ es la función en letra mayúscula $X(s)$). Toma la transformada de Laplace de la segunda ecuación:

```
sage: t,s = SR.var('t,s')
sage: x = function('x')
sage: y = function('y')
sage: f = 2*x(t).diff(t,2) + 6*x(t) - 2*y(t)
sage: f.laplace(t,s)
2*s^2*laplace(x(t), t, s) - 2*s*x(0) + 6*laplace(x(t), t, s) - 2*laplace(y(t), t, s) -
↪ 2*D[0](x)(0)
```

Esto dice

$$-Y'(0) + s^2Y(s) + 2Y(s) - 2X(s) - sy(0) = 0.$$

Introduce las condiciones iniciales para $x(0)$, $x'(0)$, $y(0)$ y $y'(0)$ y resuelve las dos ecuaciones resultantes:

```
sage: var('s X Y')
(s, X, Y)
sage: eqns = [(2*s^2+6)*X-2*Y == 6*s, -2*X +(s^2+2)*Y == 3*s]
sage: solve(eqns, X,Y)
[[X == 3*(s^3 + 3*s)/(s^4 + 5*s^2 + 4),
  Y == 3*(s^3 + 5*s)/(s^4 + 5*s^2 + 4)]]
```

Ahora toma la transformada inversa de Laplace para obtener la respuesta:

```
sage: var('s t')
(s, t)
sage: inverse_laplace((3*s^3 + 9*s)/(s^4 + 5*s^2 + 4),s,t)
cos(2*t) + 2*cos(t)
sage: inverse_laplace((3*s^3 + 15*s)/(s^4 + 5*s^2 + 4),s,t)
-cos(2*t) + 4*cos(t)
```

Por tanto, la solución es

$$x_1(t) = \cos(2t) + 2\cos(t), \quad x_2(t) = 4\cos(t) - \cos(2t).$$

La solución puede dibujarse paramétricamente usando

```
sage: t = var('t')
sage: P = parametric_plot((cos(2*t) + 2*cos(t), 4*cos(t) - cos(2*t) ),\
....: (0, 2*pi), rgbcolor=hue(0.9))
sage: show(P)
```

Los componentes individuales pueden dibujarse usando

```
sage: t = var('t')
sage: p1 = plot(cos(2*t) + 2*cos(t), 0, 2*pi, rgbcolor=hue(0.3))
sage: p2 = plot(4*cos(t) - cos(2*t), 0, 2*pi, rgbcolor=hue(0.6))
sage: show(p1 + p2)
```

REFERENCIAS: Nagle, Saff, Snider, Fundamentos De Ecuaciones Diferenciales, 6a ed, Addison-Wesley, 2004. (véase § 5.5).

2.4.4 Método De Euler Para Sistemas De Ecuaciones Diferenciales

En el siguiente ejemplo, ilustraremos el método de Euler para EDOs de primer y segundo órden. Primero, recordemos la idea básica para ecuaciones de primer órden. Dado un problema con valor inicial de la forma

$$y' = f(x, y)y(a) = c$$

queremos encontrar el valor aproximado de la solución en $x = b$ con $b > a$.

Recuerda de la definición de derivada que

$$y'(x) \approx \frac{y(x+h) - y(x)}{h},$$

donde $h > 0$ está dado y es pequeño. Esto, junto con la ED, dan $f(x, y(x)) \approx \frac{y(x+h)-y(x)}{h}$. Ahora resuelve para $y(x + h)$:

$$y(x + h) \approx y(x) + h * f(x, y(x)).$$

Si llamamos a $hf(x, y(x))$ el «término de corrección» (a falta de algo mejor), llamamos a $y(x)$ «el valor viejo de y», y llamamos a $y(x + h)$ el «nuevo valor de y», entonces, esta aproximación puede re-expresarse como

$$y_{nuevo} \approx y_{viejo} + h * f(x, y_{viejo}).$$

Si descomponemos el intervalo desde a a b en n pasos, de modo que $h = \frac{b-a}{n}$, podemos guardar la información dada por este método en una tabla.

x	y	$hf(x, y)$
a	c	$hf(a, c)$
$a + h$	$c + hf(a, c)$	$\ldots$
$a + 2h$	$\ldots$	
$\ldots$		
$b = a + nh$	???	$\ldots$

La meta es llenar todos los espacios de la tabla, una fila cada la vez, hasta que lleguemos a la casilla ???, que será la aproximación del método de Euler para $y(b)$.

La idea para los sistemas de EDOs es similar.

Ejemplo: Aproxima numéricamente $z(t)$ en $t = 1$ usando 4 pasos del método de Euler, donde $z'' + tz' + z = 0$, $z(0) = 1$, $z'(0) = 0$.

Debemos reducir la EDO de segundo órden a un sistema de dos EDs de primer órden (usando $x = z$, $y = z'$) y aplicar el método de Euler:

```
sage: t,x,y = PolynomialRing(RealField(10),3,"txy").gens()
sage: f = y; g = -x - y * t
sage: eulers_method_2x2(f,g, 0, 1, 0, 1/4, 1)
      t                x            h*f(t,x,y)                y        h*g(t,x,y)
      0                1                  0.00                0           -0.25
    1/4              1.0                -0.062            -0.25           -0.23
    1/2             0.94                 -0.12            -0.48           -0.17
    3/4             0.82                 -0.16            -0.66          -0.081
      1             0.65                 -0.18            -0.74           0.022
```

Por tanto, $z(1) \approx 0.75$.

También podemos dibujar los puntos (x, y) para obtener una representación aproximada de la curva. La función que hace esto es `eulers_method_2x2_plot`. Para poder usarla, necesitamos definir las funciones *f* y *g* que toman un argumento con tres coordenadas: (*t*, *x*,*y*).

```
sage: f = lambda z: z[2]         # f(t,x,y) = y
sage: g = lambda z: -sin(z[1])   # g(t,x,y) = -sin(x)
sage: P = eulers_method_2x2_plot(f,g, 0.0, 0.75, 0.0, 0.1, 1.0)
```

A estas alturas, `P` está guardando dos gráficas: `P[0]`, el gráfico de *x* vs. *t*, y `P[1]`, el gráfico de *y* vs. *t*. Podemos mostrar ámbas como sigue:

```
sage: show(P[0] + P[1])
```

2.4.5 Funciones Especiales

Se han implementado varios polinomios ortogonales y funciones especiales, utilizando tanto PARI [GAP] como Maxima [Max]. Estas funciones están documentadas en las secciones apropiadas («Polinomios Ortogonales» y «Funciones Especiales», respectivamente) del manual de referencia de Sage.

```
sage: x = polygen(QQ, 'x')
sage: chebyshev_U(2,x)
4*x^2 - 1
sage: bessel_I(1,1).n(250)
0.56515910399248502720769602760986330732889962162109200948029448947925564096
sage: bessel_I(1,1).n()
0.565159103992485
sage: bessel_I(2,1.1).n()  # los últimos digitos son al azar
0.16708949925104...
```

Hasta este punto, Sage únicamente ha encapsulado estas funciones para uso numérico. Para uso simbólico, por favor utiliza directamente la interfaz a Maxima, como en el siguiente ejemplo:

```
sage: maxima.eval("f:bessel_y(v, w)")
'bessel_y(v,w)'
sage: maxima.eval("diff(f,w)")
'(bessel_y(v-1,w)-bessel_y(v+1,w))/2'
```

2.5 Anillos Elementales

Cuando definimos matrices, vectores o polinomios, a veces es útil, incluso necesario, especificar el «anillo» sobre el que están definidos. Un *anillo* es una construcción matemática consistente en un conjunto de elementos sobre los que está bien definidas las operaciones de suma y producto; si la noción de anillo no te resulta familiar, probablemente sólo necesitas conocer estos cuatro anillos:

- los enteros $\{..., -1, 0, 1, 2, ...\}$, a los que nos referimos en Sage por `ZZ`.
- los números racionales – e.g., fracciones, o cocientes de números enteros –, `QQ` en Sage.
- los números reales, `RR` en Sage.
- los números complejos, `CC` en Sage.

Es importante conocer estas distinciones porque el mismo polinomio, por ejemplo, puede ser tratado de forma diferente dependiendo del anillo sobre el que se ha definido. Por ejemplo, el polinomio $x^2 - 2$ tiene dos raíces, $\pm\sqrt{2}$. Estas raíces no son racionales, así que si trabajamos con polinomios con coeficientes racionales, el polinomio es irreducible. Sin embargo, si los coeficientes son números reales, el polinomio factoriza como producto de dos factores lineales. En el siguiente ejemplo, los conjuntos de polinomios se llaman «ratpoly» y «realpoly», aunque no usaremos estos nombres; observa sin embargo que las cadenas «.<t>» y «.<z>» sirven para dar nombre a las variables usadas en cada caso.

```
sage: ratpoly.<t> = PolynomialRing(QQ)
sage: realpoly.<z> = PolynomialRing(RR)
```

Veamos el punto que hicimos antes sobre factorizar $x^2 - 2$:

```
sage: factor(t^2-2)
t^2 - 2
sage: factor(z^2-2)
(z - 1.41421356237310) * (z + 1.41421356237310)
```

Comentarios similares se aplican a las matrices: la forma reducida por filas de una matriz puede depender del anillo en que está definida, al igual que sus autovalores y autofunciones. Hay más construcciones con polinomios en la sección *Polinomios*, y más construcciones con matrices en *Álgebra Lineal*.

El símbolo `I` representa la raíz cuadrada de -1; `i` es un sinónimo de `I`. Por supuesto, no es un número racional:

```
sage: i  # raíz cuadrada de -1
I
sage: i in QQ
False
```

Nota: El código siguiente puede no funcionar como esperas si hemos asignado otro valor a la variable `i`, por ejemplo si la hemos usado como variable interna de un bucle. En este caso, podemos devolver `i` a su valor original:

```
sage: reset('i')
```

Hay una sutileza al definir números complejos: el símbolo `i` representa una raíz cuadrada de -1, pero es una raíz *formal* como elemento de un cuerpo de números algebráicos. Ejecutando `CC(i)` o `CC.0` o `CC.gen(0)` obtenemos el número *complejo* de coma flotante que es la raíz cuadrada de -1.

```
sage: i = CC(i)        # número complejo de coma flotante
sage: i == CC.0
True
sage: a, b = 4/3, 2/3
sage: z = a + b*i
sage: z
1.33333333333333 + 0.666666666666667*I
sage: z.imag()         # parte imaginaria
0.666666666666667
sage: z.real() == a    # conversión automática antes de la comparación
True
sage: a + b
2
sage: 2*b == a
True
sage: parent(2/3)
Rational Field
sage: parent(4/2)
Rational Field
sage: 2/3 + 0.1        # conversión automática antes de la suma
0.766666666666667
sage: 0.1 + 2/3        # las reglas de conversión son simétricas en Sage
0.766666666666667
```

Veamos más ejemplos de anillos elementales en Sage. Como mencionamos antes, nos podemos referir al anillo de números racionales usando `QQ`, o también `RationalField()` (*field*, o *cuerpo*, se refiere a un anillo en el que el producto es conmutativo y todo elemento excepto el cero tiene un inverso para la multiplicación. De este modo, los racionales son un cuerpo, pero los enteros no:

```
sage: RationalField()
Rational Field
sage: QQ
Rational Field
sage: 1/2 in QQ
True
```

El número decimal `1.2` se considera que está en `QQ`: los números decimales, que también son racionales, se pueden convertir a racionales de forma automática. Sin embargo, los números π y $\sqrt{2}$ no son racionales:

```
sage: 1.2 in QQ
True
sage: pi in QQ
False
sage: pi in RR
True
sage: sqrt(2) in QQ
False
sage: sqrt(2) in CC
True
```

En Sage también podemos trabajar con otros anillos, como cuerpos finitos, enteros p-ádicos, el anillo de los números algebraicos, anillos de polinomios y anillos de matrices. Veamos algunos de estos anillos:

```
sage: GF(3)
Finite Field of size 3
sage:                  # es necesario dar un nombre al generador si el número
sage: GF(27, 'a')      # de elementos no es primo
Finite Field in a of size 3^3
sage: Zp(5)
5-adic Ring with capped relative precision 20
sage: sqrt(3) in QQbar # clausura algebraica de QQ
True
```

2.6 Álgebra Lineal

Sage soporta construcciones estándar de álgebra lineal, como el polinomio característico, la forma escalonada, la traza, descomposición, etcétera de una matriz.

La creación de matrices y la multiplicación es sencilla y natural:

```
sage: A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: w = vector([1,1,-4])
sage: w*A
(0, 0, 0)
sage: A*w
(-9, 1, -2)
sage: kernel(A)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1  1 -4]
```

La descripción de `kernel(A)` indica que se trata de un subespacio de dimensión 1 («rank 1») de un espacio de dimensión 3 («degree 3»). Por el momento, tanto `kernel(A)` como el espacio ambiente admiten coeficientes enteros («over Integer Ring»). Finalmente, Sage nos muestra una base escalonada («Echelon basis»).

Observa que en Sage, el núcleo de la matriz A es el «núcleo por la izquierda», e.g. el subespacio formado por los vectores w tales que $wA = 0$.

Resolver ecuaciones matriciales es sencillo, usando el método `solve_right` (resolver por la derecha). Al evaluar `A.solve_right(Y)` obtenemos una matriz (o un vector) X tal que $AX = Y$:

```
sage: Y = vector([0, -4, -1])
sage: X = A.solve_right(Y)
sage: X
```

(continúe en la próxima página)

(proviene de la página anterior)

```
(-2, 1, 0)
sage: A * X   # comprobando la solución...
(0, -4, -1)
```

Si no hay solución, Sage lanza un error:

```
sage: A.solve_right(w)
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

De forma similar, usamos `A.solve_left(Y)` para despejar X de la ecuación $XA = Y$.

Sage también puede calcular autovalores («eigenvalues») y autovectores («eigenvectors»):

```
sage: A = matrix([[0, 4], [-1, 0]])
sage: A.eigenvalues ()
[-2*I, 2*I]
sage: B = matrix([[1, 3], [3, 1]])
sage: B.eigenvectors_left()
[(4, [
(1, 1)
], 1), (-2, [
(1, -1)
], 1)]
```

(La sintaxis de la salida de `eigenvectors_left` es una lista de tuplas: (autovalor, autovector, multiplicidad).) Los autovalores y autovectores sobre `QQ` o `RR` también se pueden calcular usando Maxima.

Como ya indicamos en *Anillos Elementales*, el anillo sobre el que se define una matriz afecta algunas de sus propiedades. En las líneas que siguen, el primer argumento al comando `matrix` le dice a Sage que considere la matriz como una matriz de enteros (si el argumento es `ZZ`), de números racionales (si es `QQ`), o de números reales (si es `RR`):

```
sage: AZ = matrix(ZZ, [[2,0], [0,1]])
sage: AQ = matrix(QQ, [[2,0], [0,1]])
sage: AR = matrix(RR, [[2,0], [0,1]])
sage: AZ.echelon_form()
[2 0]
[0 1]
sage: AQ.echelon_form()
[1 0]
[0 1]
sage: AR.echelon_form()
[ 1.00000000000000 0.000000000000000]
[0.000000000000000  1.00000000000000]
```

(El comando `echelon_form` devuelve una forma escalonada de la matriz)

2.6.1 Espacios de matrices

Creamos el espacio $\mathrm{Mat}_{3\times 3}(\mathbf{Q})$ matrices 3×3 con coeficientes racionales:

```
sage: M = MatrixSpace(QQ,3)
sage: M
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
```

(Para especificar el espacio de matrices 3 por 4, usaríamos `MatrixSpace(QQ,3,4)`. Si se omite el número de columnas, se adopta por defecto el número de filas, de modo que `MatrixSpace(QQ,3)` es un sinónimo de `MatrixSpace(QQ,3,3)`.) El espacio de matrices está equipado con su base canónica:

```
sage: B = M.basis()
sage: len(B)
9
sage: B[0,1]
[0 1 0]
[0 0 0]
[0 0 0]
```

Creamos una matriz como un elemento de `M`.

```
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
```

Calculamos su forma escalonada por filas y su núcleo.

```
sage: A.echelon_form()
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
sage: A.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

Ilustramos un cálculo de matrices definidas sobre cuerpos finitos:

```
sage: M = MatrixSpace(GF(2),4,8)
sage: A = M([1,1,0,0, 1,1,1,1, 0,1,0,0, 1,0,1,1,
....:        0,0,1,0, 1,1,0,1, 0,0,1,1, 1,1,1,0])
sage: A
[1 1 0 0 1 1 1 1]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 1 1 1 1 1 0]
sage: rows = A.rows()
sage: A.columns()
[(1, 0, 0, 0), (1, 1, 0, 0), (0, 0, 1, 1), (0, 0, 0, 1),
 (1, 1, 1, 1), (1, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1, 0)]
sage: rows
[(1, 1, 0, 0, 1, 1, 1, 1), (0, 1, 0, 0, 1, 0, 1, 1),
 (0, 0, 1, 0, 1, 1, 0, 1), (0, 0, 1, 1, 1, 1, 1, 0)]
```

Construimos el subespacio sobre $\mathbf{F}_2$ engendrado por las filas de arriba.

```
sage: V = VectorSpace(GF(2),8)
sage: S = V.subspace(rows)
sage: S
Vector space of degree 8 and dimension 4 over Finite Field of size 2
Basis matrix:
[1 0 0 0 0 1 0 0]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 0 1 0 0 1 1]
sage: A.echelon_form()
[1 0 0 0 0 1 0 0]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 0 1 0 0 1 1]
```

La base de S usada por Sage se obtiene de las filas no nulas de la forma escalonada reducida de la matriz compuesta por los generadores de S.

2.6.2 Álgebra Lineal Dispersa

Sage soporta espacios de matrices sobre DIPs almacenados de forma dispersa.

```
sage: M = MatrixSpace(QQ, 100, sparse=True)
sage: A = M.random_element(density = 0.05)
sage: E = A.echelon_form()
```

El algoritmo multi-modular de Sage es bueno para matrices cuadradas (pero no tan bueno para matrices no cuadradas):

```
sage: M = MatrixSpace(QQ, 50, 100, sparse=True)
sage: A = M.random_element(density = 0.05)
sage: E = A.echelon_form()
sage: M = MatrixSpace(GF(2), 20, 40, sparse=True)
sage: A = M.random_element()
sage: E = A.echelon_form()
```

2.7 Polinomios

En esta sección mostraremos cómo crear y usar polinomios en Sage.

2.7.1 Polinomios en una variable

Hay tres formas de construir anillos de polinomios.

```
sage: R = PolynomialRing(QQ, 't')
sage: R
Univariate Polynomial Ring in t over Rational Field
```

De esta forma creamos un anillo de polinomios en una variable, y pedimos que esta variable se muestre por pantalla como `t`. Sin embargo, de esta forma no se define `t` como variable simbólica en Sage, y no se puede usar este símbolo para escribr polinomios de `R` como por ejemplo $t^2 + 1$.

Otra forma es:

```
sage: S = QQ['t']
sage: S == R
True
```

Los mismos comentarios sobre la variable `t` se aplican a esta forma.

Una tercera forma, muy práctica es

```
sage: R.<t> = PolynomialRing(QQ)
```

o

```
sage: R.<t> = QQ['t']
```

o incluso

```
sage: R.<t> = QQ[]
```

Todas estas formas tienen el efecto añadido de definir la variable `t` como la indeterminada del anillo de polinomios, lo que hace más sencillo definir elementos de `R`. (Esta tercera forma es similar a la notación de Magma [MAGMA] , y al igual que en Magma se puede usar para una amplia variedad de objetos.)

```
sage: poly = (t+1) * (t+2); poly
t^2 + 3*t + 2
sage: poly in R
True
```

Independientemente de la forma usada para definir un anillo de polinomios, podemos recuperar la indeterminada mediante el generador 0-ésimo.

```
sage: R = PolynomialRing(QQ, 't')
sage: t = R.0
sage: t in R
True
```

Observa que una construcción similar funciona con los números complejos, que pueden ser vistos como el conjunto generado por los números reales y el símbolo `i`:

```
sage: CC
Complex Field with 53 bits of precision
sage: CC.0  # 0th generator of CC
1.00000000000000*I
```

También podemos obtener tanto el anillo como el generador, o sólo el generador, en el momento de crear un anillo de polinomios, del modo siguiente:

```
sage: R, t = QQ['t'].objgen()
sage: t    = QQ['t'].gen()
sage: R, t = objgen(QQ['t'])
sage: t    = gen(QQ['t'])
```

Finalmente hacemos un poco de aritmética en $\mathbf{Q}[t]$.

```
sage: R, t = QQ['t'].objgen()
sage: f = 2*t^7 + 3*t^2 - 15/19
sage: f^2
4*t^14 + 12*t^9 - 60/19*t^7 + 9*t^4 - 90/19*t^2 + 225/361
```

(continúe en la próxima página)

(proviene de la página anterior)

```
sage: cyclo = R.cyclotomic_polynomial(7); cyclo
t^6 + t^5 + t^4 + t^3 + t^2 + t + 1
sage: g = 7 * cyclo * t^5 * (t^5 + 10*t + 2)
sage: g
7*t^16 + 7*t^15 + 7*t^14 + 7*t^13 + 77*t^12 + 91*t^11 + 91*t^10 + 84*t^9
       + 84*t^8 + 84*t^7 + 84*t^6 + 14*t^5
sage: F = factor(g); F
(7) * t^5 * (t^5 + 10*t + 2) * (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1)
sage: F.unit()
7
sage: list(F)
[(t, 5), (t^5 + 10*t + 2, 1), (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1, 1)]
```

Observamos que la factorización tiene en cuenta la unidad que multiplica a los factores irreducibles.

Si en el curso de nuestra investigación usásemos mucho, por ejemplo, la función `R.cyclotomic_polynomial`, sería recomendable citar, además de a Sage, a la componente de Sage que realiza el cálculo en última instancia. En este caso, ejecutando `R.cyclotomic_polynomial??` para ver el código fuente, observamos la línea `f = pari.polcyclo(n)` , lo que significa que para este cálculo se usa PARI, y deberíamos citarlo además de Sage.

Al dividir dos polinomios, construimos un elemento del cuerpo de fracciones (que Sage crea automáticamente).

```
sage: x = QQ['x'].0
sage: f = x^3 + 1; g = x^2 - 17
sage: h = f/g;  h
(x^3 + 1)/(x^2 - 17)
sage: h.parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

Usando series de Laurent, podemos calcular expansiones en serie de potencias de elementos del cuerpo de fracciones de `QQ[x]`:

```
sage: R.<x> = LaurentSeriesRing(QQ); R
Laurent Series Ring in x over Rational Field
sage: 1/(1-x) + O(x^10)
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + O(x^10)
```

Si usamos otro nombre para la variable, obtenemos un anillo diferente.

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<y> = PolynomialRing(QQ)
sage: x == y
False
sage: R == S
False
sage: R(y)
x
sage: R(y^2 - 17)
x^2 - 17
```

El anillo de polinomios está determinado por el anillo de coeficientes y la variable. Observamos que construir otro anillo con una variable de nombre `x` no devuelve un anillo distinto.

```
sage: R = PolynomialRing(QQ, "x")
sage: T = PolynomialRing(QQ, "x")
sage: R == T
True
```

(continúe en la próxima página)

(proviene de la página anterior)

```
sage: R is T
True
sage: R.0 == T.0
True
```

Sage soporta los anillos de series de potencias y de series de Laurent sobre cualquier anillo base. En el ejemplo siguiente, creamos un elemento de $\mathbf{F}_7[[T]]$ y calculamos su inverso para crear un elemento de $\mathbf{F}_7((T))$.

```
sage: R.<T> = PowerSeriesRing(GF(7)); R
Power Series Ring in T over Finite Field of size 7
sage: f = T  + 3*T^2 + T^3 + O(T^4)
sage: f^3
T^3 + 2*T^4 + 2*T^5 + O(T^6)
sage: 1/f
T^-1 + 4 + T + O(T^2)
sage: parent(1/f)
Laurent Series Ring in T over Finite Field of size 7
```

También podemos crear anillos de series de potencias usando dobles corchetes:

```
sage: GF(7)[['T']]
Power Series Ring in T over Finite Field of size 7
```

2.7.2 Polinomios en varias variables

Para trabajar con polinomios de varias variables, comenzamos por declarar el anillo de polinomios y las variables.

```
sage: R = PolynomialRing(GF(5),3,"z") # here, 3 = number of variables
sage: R
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
```

Al igual que al definir anillos de polinomios en una variable, hay varias formas:

```
sage: GF(5)['z0, z1, z2']
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
sage: R.<z0,z1,z2> = GF(5)[]; R
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
```

Es posible usar una letra distinta para cada variable usando la notación:

```
sage: PolynomialRing(GF(5), 3, 'xyz')
Multivariate Polynomial Ring in x, y, z over Finite Field of size 5
```

Veamos un poco de aritmética:

```
sage: z = GF(5)['z0, z1, z2'].gens()
sage: z
(z0, z1, z2)
sage: (z[0]+z[1]+z[2])^2
z0^2 + 2*z0*z1 + z1^2 + 2*z0*z2 + 2*z1*z2 + z2^2
```

Es posible usar una notación más parecida a la convención usual en matemáticas para definir el anillo.

```
sage: R = GF(5)['x,y,z']
sage: x,y,z = R.gens()
sage: QQ['x']
Univariate Polynomial Ring in x over Rational Field
sage: QQ['x,y'].gens()
(x, y)
sage: QQ['x'].objgens()
(Univariate Polynomial Ring in x over Rational Field, (x,))
```

Los polinomios en varias variables están implementados en Sage usando diccionarios de Python y la «representación distributiva» de un polinomio. Sage usa en parte Singular [Si], por ejemplo para el cálculo del mcd de dos polinomios y la base de Gröbner de un ideal.

```
sage: R, (x, y) = PolynomialRing(RationalField(), 2, 'xy').objgens()
sage: f = (x^3 + 2*y^2*x)^2
sage: g = x^2*y^2
sage: f.gcd(g)
x^2
```

A continuación creamos el ideal (f, g) generado por f y g, simplemente multiplicando la tupla `(f,g)` por `R` (también podemos escribir `ideal([f,g])` o `ideal(f,g)`).

```
sage: I = (f, g)*R; I
Ideal (x^6 + 4*x^4*y^2 + 4*x^2*y^4, x^2*y^2) of Multivariate Polynomial
Ring in x, y over Rational Field
sage: B = I.groebner_basis(); B
[x^6, x^2*y^2]
sage: x^2 in I
False
```

La base de Gröbner de arriba no es una lista, sino una secuencia inmutable. Esto implica que tiene un universo y un padre, y que no se puede cambiar (lo cual es importante porque otras rutinas usarán esta base de Gröbner).

```
sage: B.parent()
<class 'sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic'>
sage: B.universe()
Multivariate Polynomial Ring in x, y over Rational Field
sage: B[1] = x
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Sage incluye código basado en la librería Singular que permite hacer algo de álgebra conmutativa (entiéndase: no tanta como nos gustaría). Por ejemplo, podemos calcular la descomposición primaria y los primos asociados a I:

```
sage: I.primary_decomposition()
[Ideal (x^2) of Multivariate Polynomial Ring in x, y over Rational Field,
 Ideal (y^2, x^6) of Multivariate Polynomial Ring in x, y over Rational Field]
sage: I.associated_primes()
[Ideal (x) of Multivariate Polynomial Ring in x, y over Rational Field,
 Ideal (y, x) of Multivariate Polynomial Ring in x, y over Rational Field]
```

2.8 Grupos Finitos y Grupos Abelianos

Sage ofrece algunas posibilidades para trabajar con grupos de permutaciones, grupos finitos clásicos (como $SU(n, q)$), grupos finitos de matrices (con generadores definidos por el usuario), y grupos abelianos (incluso infinitos). La mayor parte de esta funcionalidad está implementada por medio de una interfaz con GAP.

Por ejemplo, para crear un grupo de permutaciones, introducimos una lista de generadores:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(3,4)'])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
sage: G.order()
120
sage: G.is_abelian()
False
sage: G.derived_series()           # random output (resultado aleatorio)
Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by␣
↪[(3,4), (1,2,3)(4,5)],
 Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by␣
↪[(1,5,3), (1,5)(3,4), (1,5)(2,4)]]
sage: G.center()
Subgroup generated by [()] of (Permutation Group with generators [(3,4), (1,2,3)(4,
↪5)])
sage: G.random_element()           # random output (resultado aleatorio)
(1,5,3)(2,4)
sage: print(latex(G))
\langle (3,4), (1,2,3)(4,5) \rangle
```

También podemos obtener la tabla de caracteres en formato LaTeX (usa `show(G.character_table())` para ver directamente el resultado de compilar el código LaTeX):

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: latex(G.character_table())
\left(\begin{array}{rrrr}
1 & 1 & 1 & 1 \\
1 & -\zeta_{3} - 1 & \zeta_{3} & 1 \\
1 & \zeta_{3} & -\zeta_{3} - 1 & 1 \\
3 & 0 & 0 & -1
\end{array}\right)
```

Sage también incluye los grupos clásicos y los grupos de matrices sobre cuerpos finitos:

```
sage: MS = MatrixSpace(GF(7), 2)
sage: gens = [MS([[1,0],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.conjugacy_classes_representatives()
(
[1 0]  [0 6]  [0 4]  [6 0]  [0 6]  [0 4]  [0 6]  [0 6]  [0 6]  [4 0]
[0 1], [1 5], [5 5], [0 6], [1 2], [5 2], [1 0], [1 4], [1 3], [0 2],

[5 0]
[0 3]
)
sage: G = Sp(4,GF(7))
sage: G._gap_init_()
'Symplectic Group of degree 4 over Finite Field of size 7'
sage: G
```

(continúe en la próxima página)

(proviene de la página anterior)

```
Symplectic Group of degree 4 over Finite Field of size 7
sage: G.random_element()             # random output (resultado aleatorio)
[5 5 5 1]
[0 2 6 3]
[5 0 1 0]
[4 6 3 4]
sage: G.order()
276595200
```

También podemos hacer cálculos con grupos abelianos (finitos o infinitos):

```
sage: F = AbelianGroup(5, [5,5,7,8,9], names='abcde')
sage: (a, b, c, d, e) = F.gens()
sage: d * b**2 * c**3
b^2*c^3*d
sage: F = AbelianGroup(3,[2]*3); F
Multiplicative Abelian group isomorphic to C2 x C2 x C2
sage: H = AbelianGroup([2,3], names="xy"); H
Multiplicative Abelian group isomorphic to C2 x C3
sage: AbelianGroup(5)
Multiplicative Abelian group isomorphic to Z x Z x Z x Z x Z
sage: AbelianGroup(5).order()
+Infinity
```

2.9 Teoría de Números

Sage tiene bastante funcionalidad para teoría de números. Por ejemplo, podemos hacer aritmética en $\mathbf{Z}/N\mathbf{Z}$ del modo siguiente:

```
sage: R = IntegerModRing(97)
sage: a = R(2) / R(3)
sage: a
33
sage: a.rational_reconstruction()
2/3
sage: b = R(47)
sage: b^20052005
50
sage: b.modulus()
97
sage: b.is_square()
True
```

Sage contiene las funciones estándar de teoría de números. Por ejemplo:

```
sage: gcd(515,2005)
5
sage: factor(2005)
5 * 401
sage: c = factorial(25); c
15511210043330985984000000
sage: [valuation(c,p) for p in prime_range(2,23)]
[22, 10, 6, 3, 2, 1, 1, 1]
sage: next_prime(2005)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
2011
sage: previous_prime(2005)
2003
sage: divisors(28); sum(divisors(28)); 2*28
[1, 2, 4, 7, 14, 28]
56
56
```

¡Un número perfecto!

La función `sigma(n,k)` suma las potencias k-ésimas de los divisores de `n`:

```
sage: sigma(28,0); sigma(28,1); sigma(28,2)
6
56
1050
```

A continuación ilustramos el algoritmo de Euclides extendido, la función ϕ de Euler, y la función `prime_to_m_part(n, m)`, que devuelve el mayor divisor de `n` que es primo relativo a `m`:

```
sage: d,u,v = xgcd(12,15)
sage: d == u*12 + v*15
True
sage: n = 2005
sage: inverse_mod(3,n)
1337
sage: 3 * 1337
4011
sage: prime_divisors(n)
[5, 401]
sage: phi = n*prod([1 - 1/p for p in prime_divisors(n)]); phi
1600
sage: euler_phi(n)
1600
sage: prime_to_m_part(n, 5)
401
```

Seguimos con un ejemplo sobre el problema $3n + 1$:

```
sage: n = 2005
sage: for i in range(1000):
....:     n = 3*odd_part(n) + 1
....:     if odd_part(n)==1:
....:         print(i)
....:         break
38
```

Finalmente ilustramos el teorema chino del resto:

```
sage: x = crt(2, 1, 3, 5); x
11
sage: x % 3  # x mod 3 = 2
2
sage: x % 5  # x mod 5 = 1
1
sage: [binomial(13,m) for m in range(14)]
[1, 13, 78, 286, 715, 1287, 1716, 1716, 1287, 715, 286, 78, 13, 1]
```

(continúe en la próxima página)

(proviene de la página anterior)

```
sage: [binomial(13,m)%2 for m in range(14)]
[1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
sage: [kronecker(m,13) for m in range(1,13)]
[1, -1, 1, 1, -1, -1, -1, -1, 1, 1, -1, 1]
sage: n = 10000; sum([moebius(m) for m in range(1,n)])
-23
sage: Partitions(4).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

2.9.1 Números p-ádicos

El cuerpo de números p-ádicos está implementado en Sage. Observa que una vez creamos un cuerpo p-ádico, no podemos cambiar su precisión:

```
sage: K = Qp(11); K
11-adic Field with capped relative precision 20
sage: a = K(211/17); a
4 + 4*11 + 11^2 + 7*11^3 + 9*11^5 + 5*11^6 + 4*11^7 + 8*11^8 + 7*11^9
+ 9*11^10 + 3*11^11 + 10*11^12 + 11^13 + 5*11^14 + 6*11^15 + 2*11^16
+ 3*11^17 + 11^18 + 7*11^19 + O(11^20)
sage: b = K(3211/11^2); b
10*11^-2 + 5*11^-1 + 4 + 2*11 + O(11^18)
```

Se ha hecho mucho trabajo para implementar otros anillos de enteros sobre cuerpos p-ádicos. El lector interesado está invitado a pedir más detalles a los expertos en el grupo de Google `sage-support`.

Varios métodos relacionados están implementados en la clase `NumberField`:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K = NumberField(x^3 + x^2 - 2*x + 8, 'a')
sage: K.integral_basis()
[1, 1/2*a^2 + 1/2*a, a^2]
```

```
sage: K.galois_group()
Galois group 3T2 (S3) with order 6 of x^3 + x^2 - 2*x + 8
```

```
sage: K.polynomial_quotient_ring()
Univariate Quotient Polynomial Ring in a over Rational Field with modulus
x^3 + x^2 - 2*x + 8
sage: K.units()
(-3*a^2 - 13*a - 13,)
sage: K.discriminant()
-503
sage: K.class_group()
Class group of order 1 of Number Field in a with defining polynomial x^3 + x^2 - 2*x␣
↪+ 8
sage: K.class_number()
1
```

CAPÍTULO 3

Indices and tables

- genindex
- modindex
- search

Bibliografía

[PyT] El Tutorial De Python, https://docs.python.org/es/3/tutorial/

[PyB] La Guía Para Principiantes De Python, https://wiki.python.org/moin/BeginnersGuide

[Sage] Sage, https://www.sagemath.org

[GAP] El Grupo GAP, `GAP – Grupos, Algorítmos y Programación`, https://www.gap-system.org

[Max] Maxima, http://maxima.sf.net/

[Si] Singular es un sistema de álgebra computerizado para cálculos con polinomios, http://www.singular.uni-kl.de

[MAGMA] Sistema de algebra computacional, http://magma.maths.usyd.edu.au/magma/