Tutorial Sage

Release 8.6

The Sage Group

jan 21, 2019

Sumário

Sage é um software de matemática gratuito, de código aberto, para uso em ensino e pesquisa em álgebra, geometria, teoria de números, criptografia, computação numérica, e áreas relacionadas. Tanto o modelo de desenvolvimento como a tecnologia empregada no Sage se distinguem pela forte ênfase em transparência, cooperação, e colaboração: estamos desenvolvendo o carro, não reinventando a roda. O objetivo maior do Sage é criar uma alternativa viável, gratuita, e de código aberto aos programas Maple, Mathematica, Magma e MATLAB.

Este tutorial é a melhor forma de se familiarizar com o Sage em apenas algumas horas. Você pode lê-lo em versão HTML ou PDF, ou diretamente no Notebook Sage (clique em `Help`, e então clique em `Tutorial` para percorrer o tutorial de forma iterativa diretamente do Sage).

CAPÍTULO 1

Introdução

Este tutorial leva no máximo de 3 a 4 horas para ser percorrido. Você pode lê-lo em versão HTML ou PDF, ou a partir do Notebook Sage (clique em `Help`, então clique em `Tutorial` para percorrer o tutorial de forma interativa).

Embora grande parte do Sage seja implementado em Python, nenhum conhecimento de Python é necessário para a leitura deste tutorial. Você vai querer aprender Python (uma linguagem muito divertida!) em algum momento, e existem diversas opções gratuitas disponíveis para isso, entre elas *[PyT]* e *[Dive]* (em inglês). Se você quiser experimentar o Sage rapidamente, este tutorial é o lugar certo para começar. Por exemplo:

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

(continues on next page)

(continuação da página anterior)

```
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 Instalação

Se você não tem o Sage instalado em um computador e quer apenas experimentar alguns comandos, use o Sage através do site http://sagecell.sagemath.org.

Veja o guia de instalação do Sage na seção de documentação na página principal do Sage *[SA]* para instruções de como instalar o Sage no seu computador. Aqui faremos apenas alguns comentários.

1. O arquivo para instalação do Sage vem com "baterias incluídas". Em outras palavras, embora o Sage use o Python, IPython, PARI, GAP, Singular, Maxima, NTL, GMP, e uma série de outros programas, você não precisa instalá-los separadamente pois eles estão incluídos no Sage. Todavia, para usar alguns recursos, por exemplo, o Macaulay ou o KASH, você precisa instalar pacotes de software adicionais ou ter os programas necessários já instalados no seu computador. O Macaulay e o KASH estão disponíveis como pacotes adicionais do Sage (para uma lista de pacotes adicionais, digite `sage -optional`, ou visite a seção "Download" na página do Sage na internet).

2. A versão pré-compilada do Sage (disponível na página do Sage na internet) pode ser mais fácil e rápida para instalar do que a versão obtida compilando o código fonte.

3. Se você quiser usar o pacote SageTeX (que permite inserir cálculos do Sage em um arquivo LaTeX), você deve tornar o SageTex disponível para a sua distribuição TeX. Para fazer isso, consulte a seção "Make SageTex known to TeX" no Sage installation guide. O procedimento é bem simples; você precisa apenas definir algumas variáveis no seu sistema ou copiar um arquivo para um diretório onde o TeX poderá encontrá-lo.

   A documentação para usar o SageTex está disponível em `$SAGE_ROOT/local/share/texmf/tex/latex/sagetex/`, onde `$SAGE_ROOT` refere-se ao diretório onde você instalou o Sage – por exemplo, `/opt/sage-4.2.1`.

1.2 Formas de usar o Sage

Você pode usar o Sage de diversas formas.

- **Interface gráfica Notebook:** veja a seção sobre o Notebook em *A Interface do Notebook*,
- **Linha de comando interativa:** veja *A Linha de Comando Interativa*,
- **Programas:** escrevendo programas interpretados e compilados em Sage (veja *Carregando e Anexando Arquivos do Sage* e *Criando Código Compilado*), e
- **Scripts:** escrevendo scripts em Python que usam a biblioteca do Sage (veja *Scripts Independentes em Python/Sage*).

1.3 Objetivos do Sage a longo prazo

- **Útil**: O público alvo do Sage são estudantes de matemática (desde o ensino médio até a pós-graduação), professores, e pesquisadores em matemática. O objetivo é fornecer um software que possa ser usado para explorar e experimentar construções matemáticas em álgebra, geometria, teoria de números, cálculo, computação numérica, etc. O Sage torna mais fácil a experimentação com objetos matemáticos de forma interativa.
- **Eficiente:** Ser rápido. O Sage usa software bastante otimizado como o GMP, PARI, GAP, e NTL, e portanto é muito rápido em certas operações.
- **Gratuito e de código aberto:** O código fonte deve ser amplamente disponível e legível, de modo que os usuários possam entender o que o software realmente faz e possam facilmente estendê-lo. Da mesma forma que matemáticos ganham entendimento sobre um teorema lendo cuidadosamente a sua demonstração, as pessoas que fazem cálculos deveriam poder entender como os cálculos são feitos lendo o código fonte e seus comentários. Se você usar o Sage para fazer cálculos em um artigo que seja publicado, você pode ter certeza que os leitores sempre terão livre acesso ao Sage e seu código fonte, e você tem até mesmo permissão para arquivar e redistribuir a versão do Sage que você utilizou.
- **Fácil de compilar:** O Sage deve ser fácil de compilar a partir do código fonte para usuários de Linux, OS X e Windows. Isso fornece mais flexibilidade para os usuários modificarem o sistema.
- **Cooperação:** Fornecer uma interface robusta para outros sistemas computacionais, incluindo PARI, GAP, Singular, Maxima, KASH, Magma, Maple e Mathematica. O Sage foi concebido para unificar e estender outros softwares de matemática existentes.
- **Bem documentado:** Tutorial, guia de programação, manual de referência, e how-to, com inúmeros exemplos e discussão sobre conceitos matemáticos relacionados.
- **Estensível:** Ser capaz de definir novos tipos de dados ou derivá-los a partir dos tipos de dados existentes, e usar programas escritos em diversas outras linguagens.
- **Fácil de usar:** Deve ser fácil entender quais recursos estão disponíveis para um determinado objeto e consultar a documentação e o código fonte.

CAPÍTULO 2

Um passeio guiado

Esta seção é um passeio guiado pelo que está disponível no Sage. Para diversos outros exemplos, veja "Construções em Sage", que tem como objetivo responder à questão geral "Como eu construo . . . ?". Veja também o "Sage Reference Manual", que possui centenas de outros exemplos. Note que é também possível percorrer este tutorial no Sage Notebook clicando no link `Help`.

(Se você está acessando este tutorial no Sage Notebook, pressione `shift-enter` para processar qualquer célula de entrada. Você pode até editar a célula antes de pressionar `shift-enter`. Em alguns Macs pode ser necessário pressionar `shift-return` em vez de `shift-enter`.)

2.1 Atribuição, Igualdade, e Aritmética

Com pequenas exceções, o Sage utiliza a linguagem de programação Python, logo a maioria dos livros de introdução ao Python vão ajudá-lo a aprender Sage.

O Sage usa = para atribuição, e usa ==, <=, >=, < e > para comparação:

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

O Sage fornece todas as operações matemáticas básicas:

```
sage: 2**3    #  ** means exponent
8
```

(continues on next page)

(continuação da página anterior)

```
sage: 2^3      #  ^ is a synonym for ** (unlike in Python)
8
sage: 10 % 3  #  for integer arguments, % means mod, i.e., remainder
1
sage: 10/4
5/2
sage: 10//4   #  for integer arguments, // returns the integer quotient
2
sage: 4 * (10 // 4) + 10 % 4 == 10
True
sage: 3^2*4 + 2%5
38
```

O cálculo de uma expressão como `3^2*4 + 2%5` depende da ordem em que as operações são aplicadas; isso é especificado na "tabela de precedência" em *Precedência de operações aritméticas binárias*.

O Sage também fornece várias funções matemáticas básicas; aqui estão apenas alguns exemplos:

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

Como o último exemplo mostra, algumas expressões matemáticas retornam valores 'exatos' em vez de aproximações numéricas. Para obter uma aproximação numérica, use a função `n` ou o método `n` (ambos possuem um nome longo, `numerical_approx`, e a função `N` é o mesma que `n`). Essas funções aceitam o argumento opcional `prec`, que é o número de bits de precisão requisitado, e `digits`, que é o número de dígitos decimais de precisão requisitado; o padrão é 53 bits de precisão.

```
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
```

O Python é uma linguagem de tipagem dinâmica, portanto o valor referido por cada variável possui um tipo associado a ele, mas uma variável pode possuir valores de qualquer tipo em determinado escopo:

```
sage: a = 5   # a is an integer
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a = 5/3  # now a is a rational number
sage: type(a)
<type 'sage.rings.rational.Rational'>
sage: a = 'hello'  # now a is a string
sage: type(a)
<... 'str'>
```

A linguagem de programação C, que é de tipagem estática , é muito diferente; uma variável que foi declarada como int pode apenas armazenar um int em seu escopo.

2.2 Obtendo ajuda

O Sage possui vasta documentação, acessível digitando o nome de uma função ou constante (por exemplo), seguido pelo ponto de interrogação:

```
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)
        0.69314718055994530941723212215             # 32-bit
```

(continues on next page)

(continuação da página anterior)

```
        0.69314718055994530941723212145817656807   # 64-bit
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]
```

O Sage também fornece completamento tab: digite as primeiras letras de uma função e então pressione a tecla tab. Por exemplo, se você digitar `ta` seguido de `TAB`, o Sage vai imprimir `tachyon, tan, tanh, taylor`. Essa é uma boa forma de encontrar nomes de funções e outras estruturas no Sage.

2.3 Funções, Tabulação, e Contagem

Para definir uma nova função no Sage, use o comando `def` e dois pontos após a lista de nomes das variáveis. Por exemplo:

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

Observação: Dependendo da versão do tutorial que você está lendo, você pode ver três pontos `....:` na segunda linha desse exemplo. Não digite esses pontos; eles são apenas para enfatizar que o código está tabulado. Se for esse o

caso, pressione [Enter] uma vez após o fim do bloco de código para inserir uma linha em branco e concluir a definição da função.

Você não especifica o tipo de dado de nenhum dos argumentos da função. É possível especificar argumentos múltiplos, cada um dos quais pode ter um valor opcional padrão. Por exemplo, a função abaixo usa o valor padrão `divisor=2` se `divisor` não é especificado.

```
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
```

Você também pode especificar explicitamente um ou mais argumentos quando evocar uma função; se você especificar os argumentos explicitamente, você pode fazê-lo em qualquer ordem:

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

Em Python, blocos de código não são indicados por colchetes ou blocos de início e fim, como em outras linguagens. Em vez disso, blocos de código são indicados por tabulação, que devem estar alinhadas exatamente. Por exemplo, o seguinte código possui um erro de sintaxe porque o comando `return` não possui a mesma tabulação da linha que inicia o seu bloco de código.

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

Se você corrigir a tabulação, a função fica correta:

```
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]
```

Não é necessário inserir ponto-e-vírgula no final da linha. Todavia, você pode inserir múltiplos comandos em uma mesma linha separados por ponto-e-vírgula:

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

Se você quiser que uma única linha de comando seja escrita em mais de uma linha, use \ para quebrar a linha:

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

Em Sage, a contagem é feita iterando sobre um intervalo de inteiros. Por exemplo, a primeira linha abaixo é equivalente a `for(i=0; i<3; i++)` em C++ ou Java:

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

A primeira linha abaixo é equivalente a `for(i=2; i<5; i++)`.

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

O Terceiro argumento controla o passo. O comando abaixo é equivalente a `for(i=1; i<6; i+=2)`.

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

Frequentemente deseja-se criar uma tabela para visualizar resultados calculados com o Sage. Uma forma fácil de fazer isso é utilizando formatação de strings. Abaixo, criamos três colunas cada uma com largura exatamente 6, e fazemos uma tabela com quadrados e cubos de alguns números.

```
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
```

A estrutura de dados mais básica em Sage é a lista, que é – como o nome sugere – simplesmente uma lista de objetos arbitrários. Por exemplo, o comando `range` que usamos acima cria uma lista:

```
sage: range(2,10)    # py2
[2, 3, 4, 5, 6, 7, 8, 9]
sage: list(range(2,10))    # py3
[2, 3, 4, 5, 6, 7, 8, 9]
```

Abaixo segue uma lista mais complicada:

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

Listas são indexadas começando do 0, como em várias linguagens de programação.

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

Use `len(v)` para obter o comprimento de `v`, use `v.append(obj)` para inserir um novo objeto no final de `v`, e use `del v[i]` para remover o i-ésimo elemento de `v`:

```
sage: len(v)
4
sage: v.append(1.5)
sage: v
[1, 'hello', 2/3, sin(x^3), 1.500000000000000]
sage: del v[1]
sage: v
[1, 2/3, sin(x^3), 1.500000000000000]
```

Outra importante estrutura de dados é o dicionário (ou lista associativa). Ele funciona como uma lista, exceto que pode ser indexado por vários tipos de objeto (os índices devem ser imutáveis):

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

Você pode também definir novos tipos de dados usando classes. Encapsular objetos matemáticos usando classes é uma técnica poderosa que pode ajudar a simplificar e organizar os seus programas em Sage. Abaixo, definimos uma nova classe que representa a lista de inteiros pares positivos até n; essa classe é derivada do tipo `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."
```

O método `__init__` é evocado para inicializar o objeto quando ele é criado; o método `__repr__` imprime o objeto. Nós evocamos o construtor `__init__` do tipo `list` na segunda linha do método `__init__`. Criamos um objeto da classe `Evens` da seguinte forma:

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

Note que `e` imprime usando o método `__repr__` que nós definimos. Para ver a lista de números, use a função `list`:

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

Podemos também acessar o atributo `n` ou tratar `e` como uma lista.

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

2.4 Álgebra Elementar e Cálculo

O Sage pode realizar diversos cálculos em álgebra elementar e cálculo diferencial e integral: por exemplo, encontrar soluções de equações, diferenciar, integrar, e calcular a transformada de Laplace. Veja a documentação em Sage Constructions para mais exemplos.

2.4.1 Resolvendo equações

Resolvendo equações exatamente

A função `solve` resolve equações. Para usá-la, primeiro especifique algumas variáveis; então os argumentos de `solve` são uma equação (ou um sistema de equações), juntamente com as variáveis para as quais resolver:

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

Você pode resolver equações para uma variável em termos das outras:

```
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)]
```

Você pode resolver para diversas variáveis:

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

O seguinte exemplo, que mostra como usar o Sage para resolver um sistema de equações não-lineares, foi sugerido por Jason Grout: primeiro, resolvemos o sistemas simbolicamente:

```
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]]
```

Para obter soluções numéricas aproximadas, podemos usar:

```
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]]
```

(A função `n` imprime uma aproximação numérica, e o argumento é o número de bits de precisão.)

Resolvendo Equações Numericamente

Frequentemente, `solve` não será capaz de encontrar uma solução exata para uma equação ou sistema de equações. Nesse caso, você pode usar `find_root` para encontrar uma solução numérica. Por exemplo, `solve` não encontra uma solução para a equação abaixo:

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

Por outro lado, podemos usar `find_root` para encontrar uma solução para a equação acima no intervalo $0 < \phi < \pi/2$:

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

2.4.2 Diferenciação, Integração, etc.

O Sage é capaz de diferenciar e integrar diversas funções. Por exemplo, para diferenciar $\sin(u)$ com respeito a u, faça o seguinte:

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

Para calcular a quarta 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 as derivadas parciais de $x^2 + 17y^2$ com respeito 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
```

Passamos agora para integrais, tanto indefinidas como definidas. Para calcular $\int x \sin(x^2)\, dx$ e $\int_0^1 \frac{x}{x^2+1}\, dx$:

```
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 a decomposição em frações parciais 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 Resolvendo Equações Diferenciais

Você pode usar o Sage para investigar equações diferenciais ordinárias. Para resolver a equação $x' + x - 1 = 0$:

```
sage: t = var('t')     # define a variable t
sage: x = function('x')(t)    # define x to be a function of that variable
```

(continues on next page)

(continuação da página anterior)

```
sage: DE = diff(x, t) + x - 1
sage: desolve(DE, [x,t])
(_C + e^t)*e^(-t)
```

Esse método usa a interface do Sage para o Maxima *[Max]*. Logo, o formato dos resultados é um pouco diferente de outros cálculos realizados no Sage. Nesse caso, o resultado diz que a solução geral da equação diferencial é $x(t) = e^{-t}(e^t + c)$.

Você pode calcular a transformada de Laplace também; a transformada de Laplace de $t^2e^t - \sin(t)$ é calculada da seguinte forma:

```
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
```

A seguir, um exemplo mais complicado. O deslocamento, com respeito à posição de equilíbrio, de duas massas presas a uma parede através de molas, conforme a figura abaixo,

```
|------\/\/\/\/\---|massa1|----\/\/\/\/\/----|massa2|
         mola1                    mola2
```

é modelado pelo sistema de equações diferenciais de segunda ordem

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

onde, para $i = 1, 2$, m_i é a massa do objeto *i*, x_i é o deslocamento com respeito à posição de equilíbrio da massa *i*, e k_i é a constante de mola para a mola *i*.

Exemplo: Use o Sage para resolver o problema acima com $m_1 = 2$, $m_2 = 1$, $k_1 = 4$, $k_2 = 2$, $x_1(0) = 3$, $x_1'(0) = 0$, $x_2(0) = 3$, $x_2'(0) = 0$.

Solução: Primeiramente, calcule a transformada de Laplace da primeira equação (usando a notação $x = x_1$, $y = x_2$):

```
sage: de1 = maxima("2*diff(x(t),t, 2) + 6*x(t) - 2*y(t)")
sage: lde1 = de1.laplace("t","s"); lde1
2*((-%at('diff(x(t),t,1),t=0))+s^2*'laplace(x(t),t,s)-x(0)*s)-2*'laplace(y(t),t,s)+6*
↪'laplace(x(t),t,s)
```

O resultado é um pouco difícil de ler, mas diz que

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

(onde a transformada de Laplace de uma função em letra minúscula $x(t)$ é a função em letra maiúscula $X(s)$). Agora, calcule a transformada de Laplace da segunda equação:

```
sage: de2 = maxima("diff(y(t),t, 2) + 2*y(t) - 2*x(t)")
sage: lde2 = de2.laplace("t","s"); lde2
(-%at('diff(y(t),t,1),t=0))+s^2*'laplace(y(t),t,s)+2*'laplace(y(t),t,s)-2*
↪'laplace(x(t),t,s)-y(0)*s
```

O resultado significa que

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

Em seguida, substitua a condição inicial para $x(0)$, $x'(0)$, $y(0)$, e $y'(0)$, e resolva as equações 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)]]
```

Agora calcule a transformada de Laplace inversa para obter a resposta:

```
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)
```

Portanto, a solução é

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

Ela pode ser representada em um gráfico parametricamente usando os comandos

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

As componentes individuais podem ser representadas em gráfico usando

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

Leia mais sobre gráficos em *Gráficos*. Veja a seção 5.5 de *[NagleEtAl2004]* (em inglês) para mais informações sobre equações diferenciais.

2.4.4 Método de Euler para Sistemas de Equações Diferenciais

No próximo exemplo, vamos ilustrar o método de Euler para EDOs de primeira e segunda ordem. Primeiro, relembramos a ideia básica para equações de primeira ordem. Dado um problema de valor inicial da forma

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

queremos encontrar o valor aproximado da solução em $x = b$ com $b > a$.

Da definição de derivada segue que

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

onde $h > 0$ é um número pequeno. Isso, juntamente com a equação diferencial, implica que $f(x, y(x)) \approx \frac{y(x+h)-y(x)}{h}$. Agora resolvemos para $y(x + h)$:

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

Se chamarmos $hf(x, y(x))$ de "termo de correção", $y(x)$ de "valor antigo de y", e $y(x + h)$ de "novo valor de y", então essa aproximação pode ser reescrita como

$$y_{novo} \approx y_{antigo} + h * f(x, y_{antigo}).$$

Se dividirmos o intervalo de a até b em n partes, de modo que $h = \frac{b-a}{n}$, então podemos construir a seguinte tabela.

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

O objetivo é completar os espaços em branco na tabela, em uma linha por vez, até atingirmos ???, que é a aproximação para $y(b)$ usando o método de Euler.

A ideia para sistemas de EDOs é semelhante.

Exemplo: Aproxime numericamente $z(t)$ em $t = 1$ usando 4 passos do método de Euler, onde $z'' + tz' + z = 0$, $z(0) = 1$, $z'(0) = 0$.

Devemos reduzir a EDO de segunda ordem a um sistema de duas EDOs de primeira ordem (usando $x = z$, $y = z'$) e aplicar o 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
```

Portanto, $z(1) \approx 0.65$.

Podemos também representar em um gráfico os pontos (x, y) para obter uma figura da solução aproximada. A função `eulers_method_2x2_plot` fará isso; para usá-la, precisamos definir funções f e g que recebam um argumento com três 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 esta altura, `P` armazena dois gráficos: `P[0]`, o gráfico de x versus t, e `P[1]`, o gráfico de y versus t. Podemos visualizar os dois gráficos da seguinte forma:

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

(Para mais sobre gráficos, veja *Gráficos*.)

2.4.5 Funções Especiais

Diversos polinômios ortogonais e funções especiais estão implementadas, usando tanto o PARI *[GP]* como o Maxima *[Max]*. Isso está documentado nas seções apropriadas ("Orthogonal polynomials" and "Special functions", respectivamente) do manual de referência do Sage (em inglês).

```
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()
0.56515910399248...
sage: bessel_I(2,1.1).n()  # last few digits are random
0.16708949925104...
```

No momento, essas funções estão disponíveis na interface do Sage apenas para uso numérico. Para uso simbólico, use a interface do Maxima diretamente, como no seguinte exemplo:

```
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 Gráficos

O Sage pode produzir gráficos bidimensionais e tridimensionais.

2.5.1 Gráficos Bidimensionais

Em duas dimensões, o Sage pode desenhar círculos, linhas, e polígonos; gráficos de funções em coordenadas retangulares, e também coordenadas polares; gráficos de contorno e gráficos de campos vetoriais. Apresentamos alguns exemplos desses gráficos aqui. Para mais exemplos de gráficos com o Sage, veja *Resolvendo Equações Diferenciais* e *Maxima*, e também a documentação Sage Constructions.

Este comando produz um círculo amarelo de raio 1, centrado na origem.

```
sage: circle((0,0), 1, rgbcolor=(1,1,0))
Graphics object consisting of 1 graphics primitive
```

Você pode também produzir um círculo preenchido:

```
sage: circle((0,0), 1, rgbcolor=(1,1,0), fill=True)
Graphics object consisting of 1 graphics primitive
```

Outra possibilidade é criar um círculo atribuindo-o a uma variável; isso não cria um gráfico:

```
sage: c = circle((0,0), 1, rgbcolor=(1,1,0))
```

Para criar o gráfico, use `c.show()` ou `show(c)`, da seguinte forma:

```
sage: c.show()
```

Alternativamente, o comando `c.save('filename.png')` salva o gráfico no arquivo citado.

Agora, esses 'círculos' parecem mais elipses porque os eixos estão em escalas diferentes. Você pode alterar isso:

```
sage: c.show(aspect_ratio=1)
```

O comando `show(c, aspect_ratio=1)` produz o mesmo resultado, ou você pode salvar a figura usando `c.save('filename.png', aspect_ratio=1)`.

É fácil criar o gráfico de funções simples:

```
sage: plot(cos, (-5,5))
Graphics object consisting of 1 graphics primitive
```

Após especificar uma variável, você também pode criar gráficos paramétricos:

```
sage: x = var('x')
sage: parametric_plot((cos(x),sin(x)^3),(x,0,2*pi),rgbcolor=hue(0.6))
Graphics object consisting of 1 graphics primitive
```

É importante notar que os eixos dos gráficos vão se intersectar apenas se a origem estiver no escopo do gráfico, e que valores grandes podem ser representados usando notação científica.

```
sage: plot(x^2,(x,300,500))
Graphics object consisting of 1 graphics primitive
```

Você pode combinar vários gráficos somando-os:

```
sage: x = var('x')
sage: p1 = parametric_plot((cos(x),sin(x)),(x,0,2*pi),rgbcolor=hue(0.2))
sage: p2 = parametric_plot((cos(x),sin(x)^2),(x,0,2*pi),rgbcolor=hue(0.4))
sage: p3 = parametric_plot((cos(x),sin(x)^3),(x,0,2*pi),rgbcolor=hue(0.6))
sage: show(p1+p2+p3, axes=false)
```

Uma boa forma de produzir figuras preenchidas é criar uma lista de pontos (`L` no exemplo abaixo) e então usar o comando `polygon` para fazer o gráfico do polígono formado por esses pontos. Por exemplo, aqui está um "deltoid" verde:

```
sage: L = [[-1+cos(pi*i/100)*(1+cos(pi*i/100)),
....: 2*sin(pi*i/100)*(1-cos(pi*i/100))] for i in range(200)]
sage: p = polygon(L, rgbcolor=(1/8,3/4,1/2))
sage: p
Graphics object consisting of 1 graphics primitive
```

Digite `show(p, axes=false)` para visualizar isso sem os eixos.

Você pode adicionar texto ao gráfico:

```
sage: L = [[6*cos(pi*i/100)+5*cos((6/2)*pi*i/100),
....: 6*sin(pi*i/100)-5*sin((6/2)*pi*i/100)] for i in range(200)]
sage: p = polygon(L, rgbcolor=(1/8,1/4,1/2))
sage: t = text("hypotrochoid", (5,4), rgbcolor=(1,0,0))
sage: show(p+t)
```

Professores de cálculo frequentemente desenham o seguinte gráfico na lousa: não apenas um ramo do arco-seno, mas vários deles: isto é, o gráfico de $y = \sin(x)$ para x entre -2π e 2π, refletido com respeito a reta $x = y$. Os seguintes comandos fazem isso:

```
sage: v = [(sin(x),x) for x in srange(-2*float(pi),2*float(pi),0.1)]
sage: line(v)
Graphics object consisting of 1 graphics primitive
```

Como a função tangente possui imagem maior do que o seno, se você usar o mesmo método para fazer o gráfico da função inversa da função tangente, você deve alterar as coordenadas mínima e máxima para o eixo x:

```
sage: v = [(tan(x),x) for x in srange(-2*float(pi),2*float(pi),0.01)]
sage: show(line(v), xmin=-20, xmax=20)
```

O Sage também cria gráficos usando coordenadas polares, gráficos de contorno e gráficos de campos vetoriais (para tipos especiais de funções). Aqui está um exemplo de gráfico de contorno:

```
sage: f = lambda x,y: cos(x*y)
sage: contour_plot(f, (-4, 4), (-4, 4))
Graphics object consisting of 1 graphics primitive
```

2.5.2 Gráficos Tridimensionais

O Sage pode ser usado para criar gráficos tridimensionais. Tanto no Sage Notebook, como no console (linha de comando), esses gráficos serão exibidos usando o software de código aberto *[Jmol]*, que permite girar e ampliar a figura usando o mouse.

Use `plot3d` para criar o gráfico de uma função da forma $f(x, y) = z$:

```
sage: x, y = var('x,y')
sage: plot3d(x^2 + y^2, (x,-2,2), (y,-2,2))
Graphics3d Object
```

Alternativamente, você pode usar `parametric_plot3d` para criar o gráfico de uma superfície onde cada coordenada x, y, z é determinada por uma função de uma ou duas variáveis (os parâmetros, tipicamente u e v). O gráfico anterior pode ser representado parametricamente na forma:

```
sage: u, v = var('u, v')
sage: f_x(u, v) = u
sage: f_y(u, v) = v
sage: f_z(u, v) = u^2 + v^2
sage: parametric_plot3d([f_x, f_y, f_z], (u, -2, 2), (v, -2, 2))
Graphics3d Object
```

A terceira forma de fazer um gráfico de uma superfície no Sage é usando o comando `implicit_plot3d`, que cria um gráfico de uma superfície definida por uma equação $f(x, y, z) = 0$ (isso define um conjunto de pontos). Vamos fazer o gráfico de uma esfera usando a expressão usual:

```
sage: x, y, z = var('x, y, z')
sage: implicit_plot3d(x^2 + y^2 + z^2 - 4, (x,-2, 2), (y,-2, 2), (z,-2, 2))
Graphics3d Object
```

Aqui estão mais alguns exemplos:

Yellow Whitney's umbrella:

```
sage: u, v = var('u,v')
sage: fx = u*v
sage: fy = u
sage: fz = v^2
sage: parametric_plot3d([fx, fy, fz], (u, -1, 1), (v, -1, 1),
....: frame=False, color="yellow")
Graphics3d Object
```

Cross cap:

```
sage: u, v = var('u,v')
sage: fx = (1+cos(v))*cos(u)
sage: fy = (1+cos(v))*sin(u)
sage: fz = -tanh((2/3)*(u-pi))*sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi),
....: frame=False, color="red")
Graphics3d Object
```

Toro retorcido:

```
sage: u, v = var('u,v')
sage: fx = (3+sin(v)+cos(u))*cos(2*v)
sage: fy = (3+sin(v)+cos(u))*sin(2*v)
sage: fz = sin(u)+2*cos(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi),
....: frame=False, color="red")
Graphics3d Object
```

Lemniscata:

```
sage: x, y, z = var('x,y,z')
sage: f(x, y, z) = 4*x^2 * (x^2 + y^2 + z^2 + z) + y^2 * (y^2 + z^2 - 1)
sage: implicit_plot3d(f, (x, -0.5, 0.5), (y, -1, 1), (z, -1, 1))
Graphics3d Object
```

2.6 Algumas Questões Frequentes sobre Funções

Alguns aspectos sobre definição de funções (por exemplo, para diferenciação, ou para criar gráficos) podem se tornar confusos. Nesta seção, procuramos tratar algumas questões relevantes.

Aqui estão várias formas de definir objetos que merecem ser chamamos de "funções":

1. Defina uma função em Python, como descrito em *Funções, Tabulação, e Contagem*. Essas funções podem ser usadas para criar gráficos, mas não podem ser diferenciadas ou integradas.

```
sage: def f(z): return z^2
sage: type(f)
<... 'function'>
sage: f(3)
9
sage: plot(f, 0, 2)
Graphics object consisting of 1 graphics primitive
```

Na última linha, observe a sintaxe. Se fosse usado `plot(f(z), 0, 2)` ocorreria um erro, porque `z` é uma variável muda na definição de `f` e não está definida fora do contexto da função. De fato, somente `f(z)` já provoca um erro. Os seguintes comandos vão funcionar neste caso, embora em geral eles devam ser evitados pois podem ocasionar erros (veja o item 4 abaixo).

```
sage: var('z')   # define z to be a variable
z
sage: f(z)
z^2
sage: plot(f(z), 0, 2)
Graphics object consisting of 1 graphics primitive
```

Acima, `f(z)` é uma expressão simbólica, o próximo item na nossa lista.

2. Defina um "expressão simbólica que pode ser evocada". Essas podem ser usadas para criar gráficos, e podem ser diferenciadas ou integradas.

```
sage: g(x) = x^2
sage: g         # g sends x to x^2
x |--> x^2
sage: g(3)
9
sage: Dg = g.derivative(); Dg
x |--> 2*x
sage: Dg(3)
6
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
sage: plot(g, 0, 2)
Graphics object consisting of 1 graphics primitive
```

Note que enquanto `g` é uma expressão simbólica que pode ser evocada, `g(x)` é um objeto diferente, embora relacionado, que pode ser usado para criar gráficos, ou ser diferenciado, integrado, etc., embora com algumas ressalvas: veja o item 5 abaixo.

```
sage: g(x)
x^2
sage: type(g(x))
<type 'sage.symbolic.expression.Expression'>
sage: g(x).derivative()
2*x
sage: plot(g(x), 0, 2)
Graphics object consisting of 1 graphics primitive
```

3. Use uma função pré-definida. Essas podem ser representadas em gráfico, e com uma pequena ajuda, diferenciadas e integradas.

```
sage: type(sin)
<class 'sage.functions.trig.Function_sin'>
sage: plot(sin, 0, 2)
Graphics object consisting of 1 graphics primitive
sage: type(sin(x))
<type 'sage.symbolic.expression.Expression'>
sage: plot(sin(x), 0, 2)
Graphics object consisting of 1 graphics primitive
```

Por si só, `sin` não pode ser diferenciado, pelo menos não para produzir `cos`.

```
sage: f = sin
sage: f.derivative()
Traceback (most recent call last):
...
AttributeError: ...
```

Usando `f = sin(x)` no lugar de `sin` funciona, mas é ainda melhor usar `f(x) = sin(x)` para definir uma expressão simbólica que pode ser evocada.

```
sage: S(x) = sin(x)
sage: S.derivative()
x |--> cos(x)
```

Aqui estão alguns problemas comuns, com explicações:

4. Cálculo acidental.

```
sage: def h(x):
....:     if x<2:
....:         return 0
....:     else:
....:         return x-2
```

O problema: `plot(h(x), 0, 4)` cria o gráfico da reta $y = x - 2$, não da função definida por `h`. O motivo? No comando `plot(h(x), 0, 4)`, primeiro `h(x)` é calculada: isso significa substituir `x` na função `h`, o que significa que `x<2` é calculado.

```
sage: type(x<2)
<type 'sage.symbolic.expression.Expression'>
```

Quando uma equação simbólica é calculada, como na definição de `h`, se ela não é obviamente verdadeira, então ela retorna False. Logo `h(x)` é calculada como `x-2`, e essa é a função que será representada no gráfico.

A solução: não use `plot(h(x), 0, 4)`; em vez disso, use

```
sage: plot(h, 0, 4)
Graphics object consisting of 1 graphics primitive
```

5. Acidentalmente produzindo uma constante em vez de uma função.

```
sage: f = x
sage: g = f.derivative()
sage: g
1
```

O problema: `g(3)`, por exemplo, retorna o erro "ValueError: the number of arguments must be less than or equal to 0."

```
sage: type(f)
<type 'sage.symbolic.expression.Expression'>
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

`g` não é uma função, é uma constante, logo não possui variáveis associadas, e você não pode substituir nenhum valor em `g`.

Solução: existem vária opções.

- Defina `f` inicialmente como uma expressão simbólica.

```
sage: f(x) = x          # instead of 'f = x'
sage: g = f.derivative()
sage: g
x |--> 1
sage: g(3)
1
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

- Ou com `f` como definida originalmente, defina `g` como uma expressão simbólica.

```
sage: f = x
sage: g(x) = f.derivative()  # instead of 'g = f.derivative()'
sage: g
x |--> 1
sage: g(3)
1
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

- Ou com `f` e `g` como definidas originalmente, especifique a variável para a qual você está substituindo.

```
sage: f = x
sage: g = f.derivative()
sage: g
1
sage: g(x=3)    # instead of 'g(3)'
1
```

Finalmente, aqui vai mais uma forma de saber a diferença entre as derivadas de `f = x` e `f(x) = x`.

```
sage: f(x) = x
sage: g = f.derivative()
sage: g.variables()  # the variables present in g
()
sage: g.arguments()  # the arguments which can be plugged into g
(x,)
sage: f = x
sage: h = f.derivative()
sage: h.variables()
()
sage: h.arguments()
()
```

Como esse exemplo procura ilustrar, `h` não aceita argumentos, e é por isso que `h(3)` retorna um erro.

2.7 Anéis Básicos

Quando se define matrizes, vetores, ou polinômios, é as vezes útil, e as vezes necessário, especificar o "anel" sobre o qual o objeto será definido. Um *anel* é uma estrutura matemática na qual se tem noções de adição e multiplicação bem definidas; se você nunca ouviu falar sobre anéis, você provavelmente só precisa saber a respeito dos seguintes exemplos:

- os inteiros $\{..., -1, 0, 1, 2, ...\}$, que são chamados `ZZ` no Sage.
- os números racionais – i. e., frações, ou razões, de inteiros – que são chamados `QQ` no Sage.
- os números reais, chamados de `RR` no Sage.
- os números complexos, chamados de `CC` no Sage.

Você pode precisar saber sobre essas distinções porque o mesmo polinômio, por exemplo, pode ser tratado diferentemente dependendo do anel sobre o qual está definido. A propósito, o polinômio $x^2 - 2$ possui duas raízes, $\pm\sqrt{2}$. Essas raízes não são racionais, logo, se você esta lidando com polinômios com coeficientes racionais, os polinômios não serão fatorados. Com coeficientes reais, eles serão. Portanto você pode querer especificar o anel para garantir que você vai obter a informação que deseja. Os dois comandos a seguir definem os conjuntos de polinômios com coeficientes racionais e coeficientes reais, respectivamente. Os conjuntos são chamados "ratpoly" e "realpoly", mas

esses nomes não são importantes aqui; todavia, note que as strings ".<t>" e ".<z>" especificam o nome das variáveis usadas em cada caso.

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

Agora ilustramos a nossa discussão sobre fatorar $x^2 - 2$:

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

Comentários similares também se aplicam a matrizes: a forma reduzida de uma matriz pode depender do anel sobre o qual ela esta definida, como também pode os seus autovalores e autovetores. Para mais sobre polinômios, veja *Polinômios*, para mais sobre matrizes, veja *Álgebra Linear*.

O símbolo `I` representa a raiz quadrada de -1; `i` é um sinônimo de `I`. Obviamente, isso não é um número racional:

```
sage: i  # square root of -1
I
sage: i in QQ
False
```

Nota: O código acima pode não funcionar como esperado se a variável `i` estiver atribuída a um outro valor, por exemplo, se ela for usada como a variável de um laço (loop). Nesse caso, digite:

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

para restabelecer o valor original de `i`.

Há uma sutileza ao definir números complexos: como mencionado acima, o símbolo `i` representa a raiz quadrada de -1, mas é uma raiz quadrada de -1 *formal* ou *simbólica*. Evocando `CC(i)` ou `CC.0` obtém-se a raiz de -1 complexa. Aritmética envolvendo tipos diferentes de números é possível graças ao que se chama de coação, veja *Famílias, Conversão e Coação*.

```
sage: i = CC(i)        # floating point complex number
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()         # imaginary part
0.666666666666667
sage: z.real() == a    # automatic coercion before comparison
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        # automatic coercion before addition
0.766666666666667
```

(continues on next page)

(continuação da página anterior)

```
sage: 0.1 + 2/3       # coercion rules are symmetric in SAGE
0.766666666666667
```

Aqui estão mais exemplos de anéis básicos em Sage. Como observado acima, o anel dos números racionais pode ser referido usando `QQ`, ou também `RationalField()` (um *corpo*, ou *field* em inglês, é um anel no qual a operação de multiplicação é comutativa, e todo elemento não-nulo possui um elemento inverso com respeito à operação de multiplicação. Logo, os racionais formam um corpo, mas os inteiros não):

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

O número decimal `1.2` é considerado como um elemento de `QQ`: número decimais que são também racionais podem ser coagidos ao conjunto de números racionais (veja *Famílias, Conversão e Coação*). Os números π e $\sqrt{2}$ não são racionais, todavia:

```
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
```

Para uso em matemática mais avançada, o Sage também pode especificar outros anéis, como corpos finitos, inteiros p-ádicos, o anel dos números algébricos, anéis de polinômios, e anéis de matrizes. Aqui está a construção de alguns deles:

```
sage: GF(3)
Finite Field of size 3
sage: GF(27, 'a')  # need to name the generator if not a prime field
Finite Field in a of size 3^3
sage: Zp(5)
5-adic Ring with capped relative precision 20
sage: sqrt(3) in QQbar # algebraic closure of QQ
True
```

2.8 Álgebra Linear

O Sage fornece os objetos usuais em álgebra linear, por exemplo, o polinômio característico, matriz escalonada, traço, decomposição, etc., de uma matriz.

Criar e multiplicar matrizes é fácil e natural:

```
sage: A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: w = vector([1,1,-4])
sage: w*A
```

(continues on next page)

(continuação da página anterior)

```
(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]
```

Note que no Sage, o núcleo de uma matriz A é o núcleo à esquerda, i.e., o conjunto de vetores w tal que $wA = 0$.

Resolver equações matriciais é fácil usando o método `solve_right`. Calculando `A.solve_right(Y)` obtém-se uma matrix (ou vetor) X tal que $AX = Y$:

```
sage: Y = vector([0, -4, -1])
sage: X = A.solve_right(Y)
sage: X
(-2, 1, 0)
sage: A * X   # checking our answer...
(0, -4, -1)
```

Uma barra invertida `\` pode ser usada no lugar de `solve_right`; use `A \ Y` no lugar de `A.solve_right(Y)`.

```
sage: A \ Y
(-2, 1, 0)
```

Se não existir solução, o Sage retorna um erro:

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

Similarmente, use `A.solve_left(Y)` para resolver para X em $XA = Y$.

O Sage também pode calcular autovalores e autovetores:

```
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)]
```

(A sintaxe para a resposta de `eigenvectors_left` é uma lista com três componentes: (autovalor, autovetor, multiplicidade).) Autovalores e autovetores sobre `QQ` ou `RR` também podem ser calculados usando o Maxima (veja *Maxima*).

Como observado em *Anéis Básicos*, o anel sobre o qual a matriz esta definida afeta alguma de suas propriedades. A seguir, o primeiro argumento do comando `matrix` diz para o Sage considerar a matriz como uma matriz de inteiros (o caso `ZZ`), uma matriz de números racionais (`QQ`), ou uma matriz de números reais (`RR`):

```
sage: AZ = matrix(ZZ, [[2,0], [0,1]])
sage: AQ = matrix(QQ, [[2,0], [0,1]])
```

(continues on next page)

(continuação da página anterior)

```
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]
```

2.8.1 Espaços de Matrizes

Agora criamos o espaço $\mathrm{Mat}_{3\times 3}(\mathbf{Q})$ de matrizes 3×3 com entradas racionais:

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

(Para especificar o espaço de matrizes 3 por 4, você usaria `MatrixSpace(QQ,3,4)`. Se o número de colunas é omitido, ele é considerado como igual ao número de linhas, portanto, `MatrixSpace(QQ,3)` é sinônimo de `MatrixSpace(QQ,3,3)`.) O espaço de matrizes é equipado com sua base canônica:

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

Vamos criar uma matriz como um elemento de `M`.

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

A seguir calculamos a sua forma escalonada e o 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]
```

Agora ilustramos o cálculo com matrizes definidas sobre um corpo finito:

```
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
```

(continues on next page)

(continuação da página anterior)

```
[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)]
```

Criamos o subespaço sobre $\mathbf{F}_2$ gerado pelas linhas acima.

```
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]
```

A base de S usada pelo Sage é obtida a partir das linhas não-nulas da forma escalonada da matriz de geradores de S.

2.8.2 Álgebra Linear Esparsa

O Sage fornece suporte para álgebra linear esparsa.

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

O algoritmo multi-modular no Sage é bom para matrizes quadradas (mas não muito bom para matrizes que não são quadradas):

```
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()
```

Note que o Python é sensível a maiúsculas e minúsculas:

```
sage: M = MatrixSpace(QQ, 10,10, Sparse=True)
Traceback (most recent call last):
...
TypeError: __classcall__() got an unexpected keyword argument 'Sparse'
```

2.9 Polinômios

Nesta seção vamos ilustrar como criar e usar polinômios no Sage.

2.9.1 Polinômios em Uma Variável

Existem três formas de criar anéis de polinômios.

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

Esse comando cria um anel de polinômios e diz para o Sage usar a letra 't' para representar a variável indeterminada quando imprimir na tela. Todavia, isso não define o símbolo `t` para uso no Sage, logo você não pode usá-lo para definir um polinômio (como $t^2 + 1$) pertencente a `R`.

Uma forma alternativa é

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

As mesmas observações com respeito a `t` valem também nesse caso.

Uma terceira e conveniente forma de definir polinômios é

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

ou

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

ou ainda

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

Isso tem o efeito colateral de definir a variável `t` como a variável indeterminada do anel de polinômios, logo você pode facilmente construir elementos de `R` da seguinte forma. (Note que essa terceira alternativa é muito semelhante à notação usada em Magma, e da mesma forma que no Magma ela pode ser usada para diversos tipos de objetos.)

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

Qualquer que seja o método usado para definir um anel de polinômios, você pode recuperar a variável indeterminada como o 0-ésimo gerador:

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

Note que uma construção similar funciona com os números complexos: os números complexos podem ser vistos como sendo gerados pelo símbolo `i` sobre os números reais; logo temos o seguinte:

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

Para anel de polinômios, você pode obter tanto o anel como o seu gerador, ou somente o gerador, no momento em que o anel for criado, da seguinte forma:

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

Finalmente apresentamos um pouco de aritmética em $\mathbf{Q}[t]$.

```
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
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)]
```

Note que a fatorização corretamente leva em conta e armazena a parte unitária.

Se você fosse usar, por exemplo, a função `R.cyclotomic_polynomial` intensamente para algum projeto de pesquisa, além de citar o Sage, você deveria tentar descobrir qual componente do Sage é de fato usado para calcular esses polinômios, e citá-lo também. Nesse caso, se você digitar `R.cyclotomic_polynomial??` para ver o código fonte, você irá facilmente ver uma linha `f = pari.polcyclo(n)` o que significa que o PARI é usado para o cálculo dos polinômios ciclotrômicos. Cite o PARI também no seu trabalho.

Dividindo dois polinômios cria-se um elemento do corpo de frações (o qual o Sage cria automaticamente).

```
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-se a série de Laurent, pode-se calcular a expansão em série no corpo de frações 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)
```

Se nomearmos a variável de outra forma, obtemos um anel de polinômios em uma variável 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
```

O anel é determinado pela variável. Note que criar um outro anel com variável indeterminada `x` não retorna um anel diferente.

```
sage: R = PolynomialRing(QQ, "x")
sage: T = PolynomialRing(QQ, "x")
sage: R == T
True
sage: R is T
True
sage: R.0 == T.0
True
```

O Sage também possui suporte para séries de potências e séries de Laurent sobre um anel arbitrário. No seguinte exemplo, nós criamos um elemento de $\mathbf{F}_7[[T]]$ e dividimos para criar um 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
```

Você também pode criar anéis de polinômios usando a notação de colchetes duplos:

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

2.9.2 Polinômios em Mais De Uma Variável

Para trabalhar com polinômios em várias variáveis, nós primeiro declaramos o anel de polinômios e as variáveis.

```
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
```

Da mesma forma como ocorre com polinômios em uma variável, existem três maneiras de fazer isso:

```
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
```

Se você quiser usar os nomes das variáveis com apenas uma letra, então você pode usar os seguinte comando:

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

A seguir fazemos um pouco 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
```

Você também pode usar uma notação mais matemática para criar um anel de polinômios.

```
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,))
```

Polinômios em mais de uma variável são implementados no Sage usando dicionários em Python e a "representação distribuída" de um polinômio. O Sage usa o Singular *[Si]*, por exemplo, para o cálculo do maior divisor comum e bases de Gröbner para ideais algébricos.

```
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 seguir criamos o ideal (f, g) gerado por f e g, simplesmente multiplicando `(f,g)` por `R` (nós poderíamos também escrever `ideal([f,g])` ou `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
```

A base de Gröbner acima não é uma lista mas sim uma sequência imutável. Isso implica que ela possui universo (universe) e parente (parent), e não pode ser modificada (o que é bom pois ocasionaria erros em outras rotinas que usam bases de Gröbner).

```
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.
```

Um pouco (não tanto quanto gostaríamos) de álgebra comutativa está disponível no Sage, implementado via Singular. Por exemplo, podemos calcular a decomposição primaria e primos associados de 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]
```

2.10 Famílias, Conversão e Coação

Esta seção pode parecer mais técnica do que as anteriores, mas acreditamos que é importante entender o significado de famílias e coação de modo a usar anéis e outras estruturas algébricas no Sage de forma efetiva e eficiente.

Note que vamos explicar algumas noções, mas não vamos mostrar aqui como implementá-las. Um tutorial voltado à implementação está disponível (em inglês) como um tutorial temática.

2.10.1 Elementos

Caso se queira implementar um anel em Python, uma primeira aproximação seria criar uma classe para os elementos `X` do anel e adicionar os requeridos métodos (com underscores duplos) `__add__`, `__sub`, `__mul__`, obviamente garantindo que os axiomas de anel são verificados.

Como o Python é uma linguagem de tipagem forte (ainda que de tipagem dinâmica), poderia-se, pelo menos a princípio, esperar-se que fosse implementado em Python uma classe para cada anel. No final das contas, o Python contém um tipo `<int>` para os inteiros, um tipo `<float>` para os reais, e assim por diante. Mas essa estratégia logo encontra uma limitação: Existe um número infinito de anéis, e não se pode implementar um número infinito de classes.

Em vez disso, poderia-se criar uma hierarquia de classes projetada para implementar elementos de estruturas algébricas ubíquas, tais como grupos, anéis, anéis comutativos, corpos, álgebras, e assim por diante.

Mas isso significa que elementos de anéis bastante diferentes podem ter o mesmo tipo.

```
sage: P.<x,y> = GF(3)[]
sage: Q.<a,b> = GF(4,'z')[]
sage: type(x)==type(a)
True
```

Por outro lado, poderia-se ter também classes diferentes em Python fornecendo implementações diferentes da mesma estrutura matemática (por exemplo, matrizes densas versus matrizes esparsas).

```
sage: P.<a> = PolynomialRing(ZZ)
sage: Q.<b> = PolynomialRing(ZZ, sparse=True)
sage: R.<c> = PolynomialRing(ZZ, implementation='NTL')
sage: type(a) == type(b)
False
sage: type(a) == type(c)
False
sage: type(b) == type(c)
False
```

Isso apresenta dois problemas: Por um lado, se tivéssemos elementos que são duas instancias da mesma classe, então poderia-se esperar que o método `__add__` dessas classes permitisse somá-los; mas não se deseja isso, se os elementos pertencem a anéis bastante diferentes. Por outro lado, se possui-se elementos que pertencem a implementações diferentes do mesmo anel, então gostaria-se de somá-los, mas isso não pode ser feito diretamente se eles pertencem a classes diferentes em Python.

A solução para esses problemas é chamada coação e será explicada a seguir.

Todavia, é essencial que cada elemento saiba a qual pertence. Isso está disponível através método `parent()`:

```
sage: a.parent(); b.parent(); c.parent()
Univariate Polynomial Ring in a over Integer Ring
Sparse Univariate Polynomial Ring in b over Integer Ring
Univariate Polynomial Ring in c over Integer Ring (using NTL)
```

2.10.2 Famílias e Categorias

De forma similar à hierarquia de classes em Python voltada para elementos de estruturas algébricas, o Sage também fornece classes para as estruturas algébricas que contém esses elementos. Estruturas contendo elementos são chamadas "estruturas parente" no Sage, e existe uma classe básica para elas. Paralelamente à hierarquia de noções matemáticas, tem-se uma hierarquia de classes, a saber, para conjuntos, anéis, corpos e assim por diante:

```
sage: isinstance(QQ,Field)
True
sage: isinstance(QQ, Ring)
True
sage: isinstance(ZZ,Field)
False
sage: isinstance(ZZ, Ring)
True
```

Em álgebra, objetos que compartilham o mesmo tipo de estruturas algébricas são agrupados nas assim chamadas "categorias". Logo, existe uma analogia aproximada entre a hierarquia de classes em Sage e a hierarquia de categorias. Todavia, essa analogia de classes em Python e categorias não deve ser enfatizada demais. No final das contas, categorias matemáticas também são implementadas no Sage:

```
sage: Rings()
Category of rings
sage: ZZ.category()
Join of Category of euclidean domains
    and Category of infinite enumerated sets
    and Category of metric spaces

sage: ZZ.category().is_subcategory(Rings())
True
sage: ZZ in Rings()
True
sage: ZZ in Fields()
False
sage: QQ in Fields()
True
```

Enquanto a hierarquia de classes no Sage é centrada nos detalhes de implementação, a construção de categorias em Sage é mais centrada na estrutura matemática. É possível implementar métodos e testes gerais independentemente de uma implementação específica nas categorias.

Estruturas da mesma família em Sage são supostamente objetos únicos em Python. Por exemplo, uma vez que um anel de polinômios sobre um certo anel base e com uma certa lista de geradores é criada, o resultado é arquivado:

```
sage: RR['x','y'] is RR['x','y']
True
```

2.10.3 Tipos versus Parentes

O tipo `RingElement` não deve ser confundido com a noção matemática de elemento de anel; por razões práticas, as vezes um objeto é uma instancia de `RingElement` embora ele não pertence a um anel:

```
sage: cristovao = ZZ(1492)
sage: isinstance(cristovao, RingElement)
True
```

Enquanto *famílias* são únicas, elementos iguais de uma família em Sage não são necessariamente idênticos. Isso contrasta com o comportamento do Python para alguns (embora não todos) inteiros:

```
sage: int(1) is int(1) # Python int
True
sage: int(-15) is int(-15)
False
sage: 1 is 1           # Sage Integer
False
```

É importante observar que elementos de anéis diferentes em geral não podem ser distinguidos pelos seus tipos, mas sim por sua família:

```
sage: a = GF(2)(1)
sage: b = GF(5)(1)
sage: type(a) is type(b)
True
sage: parent(a)
Finite Field of size 2
sage: parent(b)
Finite Field of size 5
```

Logo, de um ponto de vista algébrico, **o parente de um elemento é mais importante do que seu tipo.**

2.10.4 Conversão versus Coação

Em alguns casos é possível converter um elemento de uma estrutura parente em um elemento de uma outra estrutura parente. Tal conversão pode ser tanto explícita como implícita (essa é chamada *coação*).

O leitor pode conhecer as noções de *conversão de tipo* e *coação de tipo* como na linguagem C, por exemplo. Existem noções de *conversão* e *coação* em Sage também. Mas as noções em Sage são centradas em *família*, não em tipos. Então, por favor não confunda conversão de tipo em C com conversão em Sage!

Aqui se encontra uma breve apresentação. Para uma descrição detalhada e informações sobre a implementação, referimos à seção sobre coação no manual de referência e para o tutorial.

Existem duas possibilidades extremas com respeito à possibilidade de fazer aritmética com elementos de *anéis diferentes*:

- Anéis diferentes são mundos diferentes, e não faz nenhum sentido somar ou multiplicar elementos de anéis diferentes; mesmo `1 + 1/2` não faz sentido, pois o primeiro somando é um inteiro e o segundo um racional.

Ou

- Se um elemento `r1` de uma aner `R1` pode de alguma forma ser interpretado em um outro anel `R2`, então todas as operações aritméticas envolvendo `r1` e qualquer elemento de `R2` são permitidas. O elemento neutro da multiplicação existe em todos os corpos e em vários anéis, e eles devem ser todos iguais.

O Sage faz uma concessão. Se `P1` e `P2` são estruturas da mesma família e `p1` é um elemento de `P1`, então o usuário pode explicitamente perguntar por uma interpretação de `p1` em `P2`. Isso pode não fazer sentido em todos os casos

ou não estar definido para todos os elementos de P1, e fica a cargo do usuário assegurar que isso faz sentido. Nos referimos a isso como **conversão**:

```
sage: a = GF(2)(1)
sage: b = GF(5)(1)
sage: GF(5)(a) == b
True
sage: GF(2)(b) == a
True
```

Todavia, uma conversão *implícita* (ou automática) ocorrerá apenas se puder ser feita *completamente* e *consistentemente*. Rigor matemático é essencial nesse ponto.

Uma tal conversão implícita é chamada **coação**. Se coação for definida, então deve coincidir com conversão. Duas condições devem ser satisfeitas para uma coação ser definida:

1. Uma coação de P1 para P2 deve ser dada por uma estrutura que preserva mapeamentos (por exemplo, um homomorfismo de anéis). Não é suficiente que *alguns* elementos de P1 possam ser mapeados em P2, e o mapa deve respeitar a estrutura algébrica de P1.
2. A escolha desses mapas de coação deve ser consistente: Se P3 é uma terceira estrutura parente, então a composição da coação adotada de P1 para P2 com a coação de P2 para P3 deve coincidir com a coação adotada de P1 para P3. Em particular, se existir uma coação de P1 para P2 e P2 para P1, a composição deve ser o mapa identidade em P1.

Logo, embora é possível converter cada elemento de GF(2) para GF(5), não há coação, pois não existe homomorfismo de anel entre GF(2) e GF(5).

O segundo aspecto - consistência - é um pouco mais difícil de explicar. Vamos ilustrá-lo usando anéis de polinômios em mais de uma variável. Em aplicações, certamente faz mais sentido ter coações que preservam nomes. Então temos:

```
sage: R1.<x,y> = ZZ[]
sage: R2 = ZZ['y','x']
sage: R2.has_coerce_map_from(R1)
True
sage: R2(x)
x
sage: R2(y)
y
```

Se não existir homomorfismo de anel que preserve nomes, coação não é definida. Todavia, conversão pode ainda ser possível, a saber, mapeando geradores de anel de acordo com sua posição da lista de geradores:

```
sage: R3 = ZZ['z','x']
sage: R3.has_coerce_map_from(R1)
False
sage: R3(x)
z
sage: R3(y)
x
```

Mas essas conversões que preservam a posição não se qualificam como coação: Compondo um mapa que preserva nomes de ZZ['x','y'] para ZZ['y','x'], com um mapa que preserva nomes de ZZ['y','x'] para ZZ['a','b'], resultaria em um mapa que não preserva nomes nem posição, violando a consistência.

Se houver coação, ela será usada para comparar elementos de anéis diferentes ou fazer aritmética. Isso é frequentemente conveniente, mas o usuário deve estar ciente que estender a relação == além das fronteiras de famílias diferentes pode facilmente resultar em problemas. Por exemplo, enquanto == é supostamente uma relação de equivalência sobre os elementos de *um* anel, isso não é necessariamente o caso se anéis *diferentes* estão envolvidos. Por exemplo, 1 em

ZZ e em um corpo finito são considerados iguais, pois existe uma coação canônica dos inteiros em qualquer corpo finito. Todavia, em geral não existe coação entre dois corpos finitos diferentes. Portanto temos

```
sage: GF(5)(1) == 1
True
sage: 1 == GF(2)(1)
True
sage: GF(5)(1) == GF(2)(1)
False
sage: GF(5)(1) != GF(2)(1)
True
```

Similarmente,

```
sage: R3(R1.1) == R3.1
True
sage: R1.1 == R3.1
False
sage: R1.1 != R3.1
True
```

Uma outra consequência da condição de consistência é que coação pode apenas ir de anéis exatos (por exemplo, os racionais QQ) para anéis não-exatos (por exemplo, os números reais com uma precisão fixa RR), mas não na outra direção. A razão é que a composição da coação de QQ em RR com a conversão de RR para QQ deveria ser a identidade em QQ. Mas isso é impossível, pois alguns números racionais distintos podem ser tratados como iguais em RR, como no seguinte exemplo:

```
sage: RR(1/10^200+1/10^100) == RR(1/10^100)
True
sage: 1/10^200+1/10^100 == 1/10^100
False
```

Quando se compara elementos de duas famílias P1 e P2, é possível que não haja coação entre os dois anéis, mas existe uma escolha canônica de um parente P3 de modo que tanto P1 como P2 são coagidos em P3. Nesse caso, coação vai ocorrer também. Um caso de uso típico é na soma de um número racional com um polinômio com coeficientes inteiros, resultando em um polinômio com coeficientes racionais:

```
sage: P1.<x> = ZZ[]
sage: p = 2*x+3
sage: q = 1/2
sage: parent(p)
Univariate Polynomial Ring in x over Integer Ring
sage: parent(p+q)
Univariate Polynomial Ring in x over Rational Field
```

Note que a princípio o resultado deveria também fazer sentido no corpo de frações de ZZ['x']. Todavia, o Sage tenta escolher um parente *canônico* comum que parece ser o mais natural (QQ['x'] no nosso exemplo). Se várias famílias potencialmente comuns parecem igualmente naturais, o Sage *não* vai escolher um deles aleatoriamente. Os mecanismos sobre os quais essa escolha se baseia é explicado em um tutorial

Nenhuma coação para um parente comum vai ocorrer no seguinte exemplo:

```
sage: R.<x> = QQ[]
sage: S.<y> = QQ[]
sage: x+y
Traceback (most recent call last):
```

(continues on next page)

(continuação da página anterior)

```
...
TypeError: unsupported operand parent(s) for +: 'Univariate Polynomial Ring in x over␣
↪Rational Field' and 'Univariate Polynomial Ring in y over Rational Field'
```

A razão é que o Sage não escolhe um dos potenciais candidatos `QQ['x']['y']`, `QQ['y']['x']`, `QQ['x','y']` ou `QQ['y','x']`, porque todas essas estruturas combinadas em pares diferentes parecem ser de famílias comuns naturais, e não existe escolha canônica aparente.

2.11 Grupos Finitos, Grupos Abelianos

O Sage possui suporte para fazer cálculos com grupos de permutação, grupos finitos clássicos (tais como $SU(n, q)$), grupos matriciais finitos (com os seus próprios geradores), e grupos abelianos (até mesmo infinitos). A maior parte disso é implementada usando a interface com o GAP.

Por exemplo, para criar um grupo de permutação, forneça uma lista de geradores, como no seguinte exemplo.

```
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-ish output
[Permutation Group with generators [(1,2,3)(4,5), (3,4)],
 Permutation Group with generators [(1,5)(3,4), (1,5)(2,4), (1,3,5)]]
sage: G.center()
Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by␣
↪[()]
sage: G.random_element()           # random output
(1,5,3)(2,4)
sage: print(latex(G))
\langle (3,4), (1,2,3)(4,5) \rangle
```

Você pode também obter a tabela de caracteres (em formato LaTeX) no Sage:

```
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)
```

O Sage também inclui grupos clássicos matriciais sobre corpos 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],
```

(continues on next page)

(continuação da página anterior)

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

Você também pode fazer cálculos usando grupos abelianos (finitos ou 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
```

2.12 Teoria de Números

O Sage possui extensa funcionalidade para teoria de números. Por exemplo, podemos fazer aritmética em $\mathbf{Z}/N\mathbf{Z}$ da seguinte forma:

```
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
```

O Sage contém funções comuns em teoria de números. Por exemplo,

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

Perfeito!

A função `sigma(n,k)` do Sage soma as k-ésimas potências dos divisores de `n`:

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

A seguir ilustramos o algoritmo de Euclides estendido, a função ϕ de Euler, e o teorema do resto Chinês:

```
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
```

Agora verificamos algo sobre o 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
```

Por fim, ilustramos o teorema do resto Chinês.

```
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]
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: list(Partitions(4))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

2.12.1 Números p-ádicos

O corpo dos números p-ádicos está implementado em Sage. Note que uma vez que um corpo p-ádico é criado, você não pode alterar a sua precisão.

```
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)
```

Muito trabalho foi feito implementando anéis de inteiros em corpos p-ádicos ou corpos numéricos além de Z. O leitor interessado é convidado a perguntar mais detalhes aos especialistas na lista `sage-support` no Google Groups.

Diversos métodos relacionados já estão implementados na classe 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(type="pari")
Galois group PARI group [6, -1, 2, "S3"] of degree 3 of the Number Field
in a with defining polynomial 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
```

(continues on next page)

(continuação da página anterior)

```
sage: K.class_number()
1
```

2.13 Um Pouco Mais de Matemática Avançada

2.13.1 Geometria Algébrica

Você pode definir variedades algébricas arbitrárias no Sage, mas as vezes alguma funcionalidade não-trivial é limitada a anéis sobre $\mathbf{Q}$ ou corpos finitos. Por exemplo, vamos calcular a união de duas curvas planas afim, e então recuperar as curvas como as componentes irredutíveis da união.

```
sage: x, y = AffineSpace(2, QQ, 'xy').gens()
sage: C2 = Curve(x^2 + y^2 - 1)
sage: C3 = Curve(x^3 + y^3 - 1)
sage: D = C2 + C3
sage: D
Affine Plane Curve over Rational Field defined by
   x^5 + x^3*y^2 + x^2*y^3 + y^5 - x^3 - y^3 - x^2 - y^2 + 1
sage: D.irreducible_components()
[
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 + y^2 - 1,
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^3 + y^3 - 1
]
```

Você também pode encontrar todos os pontos de interseção das duas curvas, intersectando-as, e então calculando as componentes irredutíveis.

```
sage: V = C2.intersection(C3)
sage: V.irreducible_components()
[
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y,
  x - 1,
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y - 1,
  x,
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x + y + 2,
  2*y^2 + 4*y + 3
]
```

Portanto, por exemplo, $(1, 0)$ e $(0, 1)$ estão em ambas as curvas (o que é claramente visível), como também estão certos pontos (quadráticos) cuja coordenada y satisfaz $2y^2 + 4y + 3 = 0$.

O Sage pode calcular o ideal toroidal da cúbica torcida no espaço-3 projetivo:

```
sage: R.<a,b,c,d> = PolynomialRing(QQ, 4)
sage: I = ideal(b^2-a*c, c^2-b*d, a*d-b*c)
sage: F = I.groebner_fan(); F
Groebner fan of the ideal:
Ideal (b^2 - a*c, c^2 - b*d, -b*c + a*d) of Multivariate Polynomial Ring
```

(continues on next page)

(continuação da página anterior)

```
in a, b, c, d over Rational Field
sage: F.reduced_groebner_bases ()
[[-c^2 + b*d, -b*c + a*d, -b^2 + a*c],
 [-b*c + a*d, -c^2 + b*d, b^2 - a*c],
 [-c^3 + a*d^2, -c^2 + b*d, b*c - a*d, b^2 - a*c],
 [-c^2 + b*d, b^2 - a*c, b*c - a*d, c^3 - a*d^2],
 [-b*c + a*d, -b^2 + a*c, c^2 - b*d],
 [-b^3 + a^2*d, -b^2 + a*c, c^2 - b*d, b*c - a*d],
 [-b^2 + a*c, c^2 - b*d, b*c - a*d, b^3 - a^2*d],
 [c^2 - b*d, b*c - a*d, b^2 - a*c]]
sage: F.polyhedralfan()
Polyhedral fan in 4 dimensions of dimension 4
```

2.13.2 Curvas Elípticas

A funcionalidade para curvas elípticas inclui a maior parte da funcionalidade para curvas elípticas do PARI, acesso aos dados da base de dados Cremona (isso requer um pacote adicional), os recursos do mwrank, isto é, "2-descends" com cálculos do grupo de Mordell-Weil completo, o algoritmo SEA (sigla em inglês), cálculo de todas as isogenias, bastante código novo para curvas sobre $\mathbf{Q}$, e parte do software "algebraic descent" de Denis Simons.

O comando `EllipticCurve` para criar curvas elípticas possui várias formas:

- EllipticCurve([a_1, a_2, a_3, a_4, a_6]): Fornece a curva elíptica

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

  onde os a_i's são coagidos para a família de a_1. Se todos os a_i possuem parente $\mathbf{Z}$, então eles são coagidos para $\mathbf{Q}$.

- EllipticCurve([a_4, a_6]): Conforme acima, mas $a_1 = a_2 = a_3 = 0$.
- EllipticCurve(label): Fornece a curva elíptica da base de dados Cremona com o "label" (novo) dado. O label é uma string, tal como `"11a"` ou `"37b2"`. As letras devem ser minúsculas (para distinguir dos labels antigos).
- EllipticCurve(j): Fornece uma curva elíptica com invariante j.
- EllipticCurve(R, [a_1, a_2, a_3, a_4, a_6]): Cria uma curva elíptica sobre um anel R com os a_i's.

Agora ilustramos cada uma dessas construções:

```
sage: EllipticCurve([0,0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field

sage: EllipticCurve([GF(5)(0),0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5

sage: EllipticCurve([1,2])
Elliptic Curve defined by y^2  = x^3 + x + 2 over Rational Field

sage: EllipticCurve('37a')
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field

sage: EllipticCurve_from_j(1)
Elliptic Curve defined by y^2 + x*y = x^3 + 36*x + 3455 over Rational Field

sage: EllipticCurve(GF(5), [0,0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5
```

O par $(0, 0)$ é um ponto na curva elíptica E definida por $y^2 + y = x^3 - x$. Para criar esse ponto digite `E([0,0])`. O Sage pode somar pontos em uma curva elíptica (lembre-se que é possível definir uma estrutura de grupo aditivo em curvas elípticas onde o ponto no infinito é o elemento nulo, e a some de três pontos colineares sobre a curva é zero):

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: P = E([0,0])
sage: P + P
(1 : 0 : 1)
sage: 10*P
(161/16 : -2065/64 : 1)
sage: 20*P
(683916417/264517696 : -18784454671297/4302115807744 : 1)
sage: E.conductor()
37
```

As curvas elípticas sobre os números complexos são parametrizadas pelo invariante j. O Sage calcula o invariante j da seguinte forma:

```
sage: E = EllipticCurve([0,0,0,-4,2]); E
Elliptic Curve defined by y^2 = x^3 - 4*x + 2 over Rational Field
sage: E.conductor()
2368
sage: E.j_invariant()
110592/37
```

Se criarmos uma curva com o mesmo invariante j que a curva E, ela não precisa ser isomórfica a E. No seguinte exemplo, as curvas não são isomórficas porque os seus condutores são diferentes.

```
sage: F = EllipticCurve_from_j(110592/37)
sage: F.conductor()
37
```

Todavia, uma torção de F por um fator 2 resulta em uma curva isomórfica.

```
sage: G = F.quadratic_twist(2); G
Elliptic Curve defined by y^2 = x^3 - 4*x + 2 over Rational Field
sage: G.conductor()
2368
sage: G.j_invariant()
110592/37
```

Nós podemos calcular os coeficientes a_n de uma série-L ou forma modular $\sum_{n=0}^{\infty} a_n q^n$ associada à curva elíptica. Esse cálculo usa a biblioteca C do PARI.

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.anlist(30)
[0, 1, -2, -3, 2, -2, 6, -1, 0, 6, 4, -5, -6, -2, 2, 6, -4, 0, -12, 0, -4,
 3, 10, 2, 0, -1, 4, -9, -2, 6, -12]
sage: v = E.anlist(10000)
```

Leva apenas um segundo para calcular todos os a_n para $n \leq 10^5$:

```
sage: %time v = E.anlist(100000)
CPU times: user 0.98 s, sys: 0.06 s, total: 1.04 s
Wall time: 1.06
```

Curvas elípticas podem ser construídas usando o "label" da base de dados Cremona. Isso importa a curva elíptica com informações prévias sobre o seu posto, números de Tomagawa, regulador, etc.

```
sage: E = EllipticCurve("37b2")
sage: E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational
Field
sage: E = EllipticCurve("389a")
sage: E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x  over Rational Field
sage: E.rank()
2
sage: E = EllipticCurve("5077a")
sage: E.rank()
3
```

Nós também podemos acessar a base de dados Cremona diretamente.

```
sage: db = sage.databases.cremona.CremonaDatabase()
sage: db.curves(37)
{'a1': [[0, 0, 1, -1, 0], 1, 1], 'b1': [[0, 1, 1, -23, -50], 0, 3]}
sage: db.allcurves(37)
{'a1': [[0, 0, 1, -1, 0], 1, 1],
 'b1': [[0, 1, 1, -23, -50], 0, 3],
 'b2': [[0, 1, 1, -1873, -31833], 0, 1],
 'b3': [[0, 1, 1, -3, 1], 0, 3]}
```

Os objetos obtidos pela base de dados não são do tipo `EllipticCurve`. Eles são elementos de uma base de dados e possuem alguns campos, e apenas isso. Existe uma versão básica da base de dados Cremona, que já é distribuída na versão padrão do Sage, e contém informações limitadas sobre curvas elípticas de condutor ≤ 10000. Existe também uma versão estendida opcional, que contém informações extensas sobre curvas elípticas de condutor ≤ 120000 (em outubro de 2005). Por fim, existe ainda uma versão (2GB) opcional de uma base de dados para o Sage que contém centenas de milhares de curvas elípticas na base de dados Stein-Watkins.

2.13.3 Caracteres de Dirichlet

Um *caractere de Dirichlet* é a extensão de um homomorfismo $(\mathbf{Z}/N\mathbf{Z})* \to R^*$, para algum anel R, para o mapa $\mathbf{Z} \to R$ obtido mapeando os inteiros x tais que $\gcd(N, x) > 1$ em 0.

```
sage: G = DirichletGroup(12)
sage: G.list()
[Dirichlet character modulo 12 of conductor 1 mapping 7 |--> 1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 4 mapping 7 |--> -1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 3 mapping 7 |--> 1, 5 |--> -1,
Dirichlet character modulo 12 of conductor 12 mapping 7 |--> -1, 5 |--> -1]
sage: G.gens()
(Dirichlet character modulo 12 of conductor 4 mapping 7 |--> -1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 3 mapping 7 |--> 1, 5 |--> -1)
sage: len(G)
4
```

Tendo criado o grupo, a seguir calculamos um elemento e fazemos cálculos com ele.

```
sage: G = DirichletGroup(21)
sage: chi = G.1; chi
Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> zeta6
```

(continues on next page)

(continuação da página anterior)

```
sage: chi.values()
[0, 1, zeta6 - 1, 0, -zeta6, -zeta6 + 1, 0, 0, 1, 0, zeta6, -zeta6, 0, -1,
 0, 0, zeta6 - 1, zeta6, 0, -zeta6 + 1, -1]
sage: chi.conductor()
7
sage: chi.modulus()
21
sage: chi.order()
6
sage: chi(19)
-zeta6 + 1
sage: chi(40)
-zeta6 + 1
```

É também possível calcular a ação do grupo de Galois $\mathrm{Gal}(\mathbf{Q}(\zeta_N)/\mathbf{Q})$ sobre esses caracteres, bem como a decomposição em produto direto correspondente à fatorização do módulo.

```
sage: chi.galois_orbit()
[Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> -zeta6 + 1,
 Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> zeta6]

sage: go = G.galois_orbits()
sage: [len(orbit) for orbit in go]
[1, 2, 2, 1, 1, 2, 2, 1]

sage: G.decomposition()
[
Group of Dirichlet characters modulo 3 with values in Cyclotomic Field of order 6 and␣
↪degree 2,
Group of Dirichlet characters modulo 7 with values in Cyclotomic Field of order 6 and␣
↪degree 2
]
```

A seguir, construímos o grupo de caracteres de Dirichlet mod 20, mas com valores em $\mathbf{Q}(i)$:

```
sage: K.<i> = NumberField(x^2+1)
sage: G = DirichletGroup(20,K)
sage: G
Group of Dirichlet characters modulo 20 with values in Number Field in i with␣
↪defining polynomial x^2 + 1
```

Agora calculamos diversos invariantes de `G`:

```
sage: G.gens()
(Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1,
Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> i)

sage: G.unit_gens()
(11, 17)
sage: G.zeta()
i
sage: G.zeta_order()
4
```

No próximo exemplo criamos um caractere de Dirichlet com valores em um corpo numérico. Nós especificamos explicitamente a escolha da raiz da unidade no terceiro argumento do comando `DirichletGroup` abaixo.

```
sage: x = polygen(QQ, 'x')
sage: K = NumberField(x^4 + 1, 'a'); a = K.0
sage: b = K.gen(); a == b
True
sage: K
Number Field in a with defining polynomial x^4 + 1
sage: G = DirichletGroup(5, K, a); G
Group of Dirichlet characters modulo 5 with values in the group of order 8 generated␣
↪by a in Number Field in a with defining polynomial x^4 + 1
sage: chi = G.0; chi
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> a^2
sage: [(chi^i)(2) for i in range(4)]
[1, a^2, -1, -a^2]
```

Aqui `NumberField(x^4 + 1, 'a')` diz para o Sage usar o símbolo "a" quando imprimir o que é `K` (um corpo numérico definido pelo polinômio $x^4 + 1$). O nome "a" não está declarado até então. Uma vez que `a = K.0` (ou equivalentemente `a = K.gen()`) é calculado, o símbolo "a" representa a raiz do polinômio gerador $x^4 + 1$.

2.13.4 Formas Modulares

O Sage pode fazer alguns cálculos relacionados a formas modulares, incluindo dimensões, calcular espaços de símbolos modulares, operadores de Hecke, e decomposições.

Existem várias funções disponíveis para calcular dimensões de espaços de formas modulares. Por exemplo,

```
sage: dimension_cusp_forms(Gamma0(11),2)
1
sage: dimension_cusp_forms(Gamma0(1),12)
1
sage: dimension_cusp_forms(Gamma1(389),2)
6112
```

A seguir ilustramos o cálculo dos operadores de Hecke em um espaço de símbolos modulares de nível 1 e peso 12.

```
sage: M = ModularSymbols(1,12)
sage: M.basis()
([X^8*Y^2,(0,0)], [X^9*Y,(0,0)], [X^10,(0,0)])
sage: t2 = M.T(2)
sage: t2
Hecke operator T_2 on Modular Symbols space of dimension 3 for Gamma_0(1)
of weight 12 with sign 0 over Rational Field
sage: t2.matrix()
[ -24    0    0]
[   0  -24    0]
[4860    0 2049]
sage: f = t2.charpoly('x'); f
x^3 - 2001*x^2 - 97776*x - 1180224
sage: factor(f)
(x - 2049) * (x + 24)^2
sage: M.T(11).charpoly('x').factor()
(x - 285311670612) * (x - 534612)^2
```

Podemos também criar espaços para $\Gamma_0(N)$ e $\Gamma_1(N)$.

```
sage: ModularSymbols(11,2)
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign
```

(continues on next page)

(continuação da página anterior)

```
 0 over Rational Field
sage: ModularSymbols(Gamma1(11),2)
Modular Symbols space of dimension 11 for Gamma_1(11) of weight 2 with
sign 0 and over Rational Field
```

Vamos calcular alguns polinômios característicos e expansões q.

```
sage: M = ModularSymbols(Gamma1(11),2)
sage: M.T(2).charpoly('x')
x^11 - 8*x^10 + 20*x^9 + 10*x^8 - 145*x^7 + 229*x^6 + 58*x^5 - 360*x^4
     + 70*x^3 - 515*x^2 + 1804*x - 1452
sage: M.T(2).charpoly('x').factor()
(x - 3) * (x + 2)^2 * (x^4 - 7*x^3 + 19*x^2 - 23*x + 11)
        * (x^4 - 2*x^3 + 4*x^2 + 2*x + 11)
sage: S = M.cuspidal_submodule()
sage: S.T(2).matrix()
[-2  0]
[ 0 -2]
sage: S.q_expansion_basis(10)
[
    q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 - 2*q^9 + O(q^10)
]
```

Podemos até mesmo calcular espaços de símbolos modulares com carácter.

```
sage: G = DirichletGroup(13)
sage: e = G.0^2
sage: M = ModularSymbols(e,2); M
Modular Symbols space of dimension 4 and level 13, weight 2, character
[zeta6], sign 0, over Cyclotomic Field of order 6 and degree 2
sage: M.T(2).charpoly('x').factor()
(x - zeta6 - 2) * (x - 2*zeta6 - 1) * (x + zeta6 + 1)^2
sage: S = M.cuspidal_submodule(); S
Modular Symbols subspace of dimension 2 of Modular Symbols space of
dimension 4 and level 13, weight 2, character [zeta6], sign 0, over
Cyclotomic Field of order 6 and degree 2
sage: S.T(2).charpoly('x').factor()
(x + zeta6 + 1)^2
sage: S.q_expansion_basis(10)
[
q + (-zeta6 - 1)*q^2 + (2*zeta6 - 2)*q^3 + zeta6*q^4 + (-2*zeta6 + 1)*q^5
  + (-2*zeta6 + 4)*q^6 + (2*zeta6 - 1)*q^8 - zeta6*q^9 + O(q^10)
]
```

Aqui está um outro exemplo de como o Sage pode calcular a ação de operadores de Hecke em um espaço de formas modulares.

```
sage: T = ModularForms(Gamma0(11),2)
sage: T
Modular Forms space of dimension 2 for Congruence Subgroup Gamma0(11) of
weight 2 over Rational Field
sage: T.degree()
2
sage: T.level()
11
sage: T.group()
```

(continues on next page)

(continuação da página anterior)

```
Congruence Subgroup Gamma0(11)
sage: T.dimension()
2
sage: T.cuspidal_subspace()
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 2 for
Congruence Subgroup Gamma0(11) of weight 2 over Rational Field
sage: T.eisenstein_subspace()
Eisenstein subspace of dimension 1 of Modular Forms space of dimension 2
for Congruence Subgroup Gamma0(11) of weight 2 over Rational Field
sage: M = ModularSymbols(11); M
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign
0 over Rational Field
sage: M.weight()
2
sage: M.basis()
((1,0), (1,8), (1,9))
sage: M.sign()
0
```

Denote por T_p os operadores de Hecke usuais (p primo). Como os operadores de Hecke T_2, T_3, e T_5 agem sobre o espaço de símbolos modulares?

```
sage: M.T(2).matrix()
[ 3  0 -1]
[ 0 -2  0]
[ 0  0 -2]
sage: M.T(3).matrix()
[ 4  0 -1]
[ 0 -1  0]
[ 0  0 -1]
sage: M.T(5).matrix()
[ 6  0 -1]
[ 0  1  0]
[ 0  0  1]
```

CAPÍTULO 3

A Linha de Comando Interativa

Na maior parte deste tutorial, assumimos que você iniciou o interpretador Sage usando o comando `sage`. Isso inicia uma versão personalizada da linha de comando IPython, e importa diversas funções e classes de modo que elas fiquem prontas para serem usadas a partir da linha de comando. Configuração adicional é possível editando o arquivo `$SAGE_ROOT/ipythonrc`. Assim que você inicia o Sage, você obtém o seguinte:

```
----------------------------------------------------------------------
| SAGE Version 3.1.1, Release Date: 2008-05-24                       |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------
sage:
```

Para sair do Sage pressione Ctrl-D ou digite `quit` ou `exit`.

```
sage: quit
Exiting SAGE (CPU time 0m0.00s, Wall time 0m0.89s)
```

O wall time é o tempo que passou no relógio "pendurado na sua parede". Isso é relevante, pois o tempo CPU não conta o tempo usado por subprocessos como GAP ou Singular.

(Evite terminar um processo do Sage usando `kill -9` a partir de um terminal, pois o Sage pode não terminal seus subprocessos, por exemplo, subprocessos do Maple, ou limpeza de arquivos temporários em `$HOME/.sage/tmp`.)

3.1 A Sua Sessão no Sage

A sessão é a sequência de entradas e saídas de dados desde o momento em que você inicia até o momento em que você termina o Sage. O Sage grava todas as entradas de dados, através do IPython. De fato, se você está usando a linha de comando (não o Notebook), então a qualquer momento você pode digitar `%history` (ou `%hist`) para obter uma lista de todas as linhas digitadas até então. Você pode digitar `?` no prompt do Sage para aprender mais sobre o IPython, por exemplo, "IPython offers numbered prompts ... with input and output caching. All input is saved and can be retrieved as variables (besides the usual arrow key recall). The following GLOBAL variables always exist (so don't overwrite them!)":

```
_:  previous input (interactive shell and notebook)
__: next previous input (interactive shell only)
_oh : list of all inputs (interactive shell only)
```

Aqui vai um exemplo:

```
sage: factor(100)
 _1 = 2^2 * 5^2
sage: kronecker_symbol(3,5)
 _2 = -1
sage: %hist   #This only works from the interactive shell, not the notebook.
1: factor(100)
2: kronecker_symbol(3,5)
3: %hist
sage: _oh
 _4 = {1: 2^2 * 5^2, 2: -1}
sage: _i1
 _5 = 'factor(ZZ(100))\n'
sage: eval(_i1)
 _6 = 2^2 * 5^2
sage: %hist
1: factor(100)
2: kronecker_symbol(3,5)
3: %hist
4: _oh
5: _i1
6: eval(_i1)
7: %hist
```

Vamos omitir a numeração das linhas no restante deste tutorial e em outras documentações do Sage.

Você também pode salvar uma lista de comandos em uma macro.

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: M = ModularSymbols(37)
sage: %hist
1: E = EllipticCurve([1,2,3,4,5])
2: M = ModularSymbols(37)
3: %hist
sage: %macro em 1-2
Macro `em` created. To execute, type its name (without quotes).
```

```
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over
Rational Field
sage: E = 5
sage: M = None
sage: em
Executing Macro...
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over
Rational Field
```

Quando se usa a linha de comando, qualquer comando UNIX pode ser executado a partir do Sage inserindo um ponto de exclamação `!` como prefixo. Por exemplo,

```
sage: !ls
auto  example.sage glossary.tex  t  tmp  tut.log  tut.tex
```

fornece a lista de arquivos do atual diretório.

O `PATH` possui o diretório bin do Sage em primeiro, portanto se você digitar `p`, `gap`, `singular`, `maxima`, etc., você executa a versão incluída no Sage.

```
sage: !gp
Reading GPRC: /etc/gprc ...Done.

                          GP/PARI CALCULATOR Version 2.2.11 (alpha)
                  i686 running linux (ix86/GMP-4.1.4 kernel) 32-bit version
...
sage: !singular
                     SINGULAR                                /  Development
 A Computer Algebra System for Polynomial Computations   /   version 3-0-1
                                                       0<
     by: G.-M. Greuel, G. Pfister, H. Schoenemann        \   October 2005
FB Mathematik der Universitaet, D-67653 Kaiserslautern    \
```

3.2 Gravando Entradas e Saídas de dados

Gravar a sua sessão no Sage não é o mesmo que salvá-la (veja *Salvando e Abrindo Sessões Completas*). Para gravar a entrada de dados (e opcionalmente a saída) use o comando `logstart`. Digite `logstart?` para mais detalhes. Você pode usar esse comando para gravar tudo o que você digita, toda a saída de dados, e até mesmo usar essa entrada de dados que você guardou em uma sessão futura (simplesmente importando o arquivo log).

```
was@form:~$ sage
------------------------------------------------------------------------
| SAGE Version 3.0.2, Release Date: 2008-05-24                         |
| Type notebook() for the GUI, and license() for information.          |
------------------------------------------------------------------------

sage: logstart setup
Activating auto-logging. Current session state plus future input saved.
Filename       : setup
Mode           : backup
Output logging : False
Timestamping   : False
State          : active
sage: E = EllipticCurve([1,2,3,4,5]).minimal_model()
sage: F = QQ^3
sage: x,y = QQ['x,y'].gens()
sage: G = E.gens()
sage:
Exiting SAGE (CPU time 0m0.61s, Wall time 0m50.39s).
was@form:~$ sage
------------------------------------------------------------------------
| SAGE Version 3.0.2, Release Date: 2008-05-24                         |
| Type notebook() for the GUI, and license() for information.          |
------------------------------------------------------------------------

sage: load "setup"
Loading log file <setup> one line at a time...
Finished replaying log file <setup>
sage: E
Elliptic Curve defined by y^2 + x*y  = x^3 - x^2 + 4*x + 3 over Rational
```

(continues on next page)

(continuação da página anterior)

```
Field
sage: x*y
x*y
sage: G
[(2 : 3 : 1)]
```

Se você usa o Sage no terminal `konsole` do Linux KDE, então você pode gravar a sessão da seguinte forma: após iniciar o Sage no `konsole`, selecione "settings", então "history...", então "set unlimited". Quando você estiver pronto para guardar a sua sessão, selecione "edit" e então "save history as..." e digite um nome para salvar o texto de sua sessão em seu computador. Após salvar esse arquivo, você poderia abri-lô em um editor, tal como o xemacs, e imprimi-lo.

3.3 Colar Texto Ignora Prompts

Suponha que você está lendo uma sequência de comandos em Sage ou Python e quer copiá-los no Sage. Mas eles têm os irritantes prompts >>> ou `sage:` para te aborrecer. De fato, você pode copiar e colar um exemplo, incluindo os prompts se você quiser, no Sage. Em outras palavras, automaticamente o Sage remove os caracteres >>> ou `sage:` antes de colar o conteúdo no Python. Por exemplo,

```
sage: 2^10
1024
sage: sage: sage: 2^10
1024
sage: >>> 2^10
1024
```

3.4 Comandos de Tempo

Se você colocar o comando `%time` no começo de uma linha de comando, o tempo que o comando leva para ser executado vai aparecer após a saída de dados. Por exemplo, nós podemos comparar o tempo de execução para certas operações de exponenciação de várias formas. Os tempos abaixo vão ser provavelmente muito diferentes para o seu computador, ou até mesmo para versões diferentes do Sage. Primeiro, usando o Python

```
sage: %time a = int(1938)^int(99484)
CPU times: user 0.66 s, sys: 0.00 s, total: 0.66 s
Wall time: 0.66
```

Isso significa que levou 0.66 segundos no total, e o wall time, isto é, a quantidade de tempo que passou no seu relógio de parede, é também 0.66 segundos. Se o seu computador está executado outros programas o wall time pode ser muito maior do que o tempo de CPU.

A seguir verificamos o tempo de exponenciação usando o tipo Integer do Sage, o qual é implementado (em Cython) usando a biblioteca GMP:

```
sage: %time a = 1938^99484
CPU times: user 0.04 s, sys: 0.00 s, total: 0.04 s
Wall time: 0.04
```

Usando a biblioteca C do PARI:

```
sage: %time a = pari(1938)^pari(99484)
CPU times: user 0.05 s, sys: 0.00 s, total: 0.05 s
Wall time: 0.05
```

A GMP é melhor, mas por pouco (como esperado, pois a versão do PARI contida no Sage usa a GMP para aritmética de inteiros).

Você pode também contar o tempo de um bloco de comandos usado o comando `cputime`, como ilustrado abaixo:

```
sage: t = cputime()
sage: a = int(1938)^int(99484)
sage: b = 1938^99484
sage: c = pari(1938)^pari(99484)
sage: cputime(t)                          # somewhat random output
0.64
```

```
sage: cputime?
...
    Return the time in CPU second since SAGE started, or with optional
    argument t, return the time since time t.
    INPUT:
        t -- (optional) float, time in CPU seconds
    OUTPUT:
        float -- time in CPU seconds
```

O comando `walltime` se comporta como o comando `cputime`, exceto que ele conta o tempo do relógio.

Nós podemos também calcular a potência acima em alguns softwares de álgebra incluídos no Sage. Em cada caso executamos um comando trivial no sistema de modo a inicializar o servidor para aquele programa. O tempo mais relevante é o tempo do relógio. Todavia, se houver uma diferença significativa entre o wall time e o CPU time então isso pode indicar algum problema de performance que vale a pena investigar.

```
sage: time 1938^99484;
CPU times: user 0.01 s, sys: 0.00 s, total: 0.01 s
Wall time: 0.01
sage: gp(0)
0
sage: time g = gp('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.04
sage: maxima(0)
0
sage: time g = maxima('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.30
sage: kash(0)
0
sage: time g = kash('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.04
sage: mathematica(0)
        0
sage: time g = mathematica('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.03
sage: maple(0)
0
```

(continues on next page)

(continuação da página anterior)

```
sage: time g = maple('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.11
sage: gap(0)
0
sage: time g = gap.eval('1938^99484;;')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 1.02
```

Note que o GAP e o Maxima são os mais lentos neste teste (isso foi executado no computador `sage.math.washington.edu`). Devido ao processamento extra (overhead) da interface pexpect, talvez não seja apropriado comparar esses resultados com o Sage, que é o mais rápido.

3.5 Outras Dicas para o IPython

Como observado acima, o Sage usa o IPython como interface, logo você pode usar quaisquer comandos e recursos do IPython. Você pode ler a Documentação completa do IPython (em inglês).

- Você pode usar `%bg` para executar um comando no background, e então usar `jobs` para acessar os resultados, da seguinte forma. (Os comentários `not tested` estão aqui porque a sintaxe `%bg` não funciona bem com o sistema de testes automáticos do Sage. Se você digitar esses comandos, eles devem funcionar. Isso é obviamente mais útil com comandos que demoram para serem completados.)

  ```
  sage: def quick(m): return 2*m
  sage: %bg quick(20)  # not tested
  Starting job # 0 in a separate thread.
  sage: jobs.status()  # not tested
  Completed jobs:
  0 : quick(20)
  sage: jobs[0].result  # the actual answer, not tested
  40
  ```

  Note que os comandos executados no background não usam o pre-processador (preparser) do Sage – veja *O Pré-Processador: Diferenças entre o Sage e o Python* para mais informações. Uma forma (estranha talvez) de contornar esse problema seria executar

  ```
  sage: %bg eval(preparse('quick(20)')) # not tested
  ```

  É mais seguro e simples, todavia, usar `%bg` apenas em comandos que não requerem o pre-processador (preparser).

- Você pode usar `%edit` (ou `%ed` ou `ed`) para abrir um editor, se você desejar digitar algum código mais complexo. Antes de iniciar o Sage, certifique-se de que a variável de ambiente `EDITOR` está definida com o seu editor favorito (colocando `export EDITOR=/usr/bin/emacs` ou `export EDITOR=/usr/bin/vim` or algo similar no lugar apropriado, como um arquivo `.profile`). A partir do prompt do Sage, o comando `%edit` irá abrir o editor escolhido. Então usando o editor você pode definir uma função:

  ```
  def some_function(n):
      return n**2 + 3*n + 2
  ```

  Salve e feche o editor. No restante da sua sessão do Sage, você pode usar então a função `some_function`. Se você quiser modificá-la, digite `%edit some_function` no prompt do Sage.

- Se você for calcular algo e quiser modificar o resultado para outro uso, execute o cálculo e então digite `%rep`: isso irá colocar o resultado do comando anterior no prompt do Sage, pronto para ser editado.:

```
sage: f(x) = cos(x)
sage: f(x).derivative(x)
-sin(x)
```

A esta altura, se você digitar `%rep` no prompt do Sage, você irá obter um novo prompt, seguido de `-sin(x)`, com o cursor no final da linha.

Para mais informações, digite `%quickref` para ver um guia rápido de referência do IPython. Quando este tutorial foi escrito (Fevereiro 2015), o Sage usa a versão 2.3.0 do IPython, e a documentation for its magic commands está disponível na internet.

3.6 Erros e Exceções

Quando algo errado ocorre, você usualmente verá uma "exceção" do Python. O Python até mesmo tenta sugerir o que ocasionou a exceção, por exemplo, `NameError` ou `ValueError` (veja o Manual de Referência do Python *[Py]* para uma lista completa de exceções). Por exemplo,

```
sage: 3_2
------------------------------------------------------------
   File "<console>", line 1
     ZZ(3)_2
           ^
SyntaxError: invalid syntax

sage: EllipticCurve([0,infinity])
------------------------------------------------------------
Traceback (most recent call last):
...
TypeError: Unable to coerce Infinity (<class 'sage...Infinity'>) to Rational
```

O debugador interativo é as vezes útil para entender o que houve de errado. Você pode ativá-lo e desativá-lo usando `%pdb` (o padrão é desativado). O prompt `ipdb>` aparece se uma exceção é levantada e o debugador está ativado. A partir do debugador, você pode imprimir o estado de qualquer variável local, e mover a pilha de execução para cima e para baixo. Por exemplo,

```
sage: %pdb
Automatic pdb calling has been turned ON
sage: EllipticCurve([1,infinity])
---------------------------------------------------------------------------
<type 'exceptions.TypeError'>             Traceback (most recent call last)
...

ipdb>
```

Para uma lista de comandos do debugador, digite `?` no prompt `ipbd>`:

```
ipdb> ?

Documented commands (type help <topic>):
========================================
EOF     break   commands   debug    h       l     pdef   quit    tbreak
a       bt      condition  disable  help    list  pdoc   r       u
alias   c       cont       down     ignore  n     pinfo  return  unalias
args    cl      continue   enable   j       next  pp     s       up
b       clear   d          exit     jump    p     q      step    w
```

(continues on next page)

(continuação da página anterior)

```
whatis where

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

Digite Ctrl-D ou `quit` para retornar ao Sage.

3.7 Busca Reversa e Completamento Tab

Busca reversa: Digite o começo de um comando, e então `Ctrl-p` (ou tecle a seta para cima) para voltar para cada linha que você digitou que começa daquela forma. Isso funciona mesmo se você encerrou o Sage e iniciou novamente mais tarde. Você também pode fazer uma busca reversa ao longo da história usando `Ctrl-r`. Todos esses recursos usam o pacote `readline`, que está disponível no Linux.

Para ilustrar a busca reversa, primeiro crie o e espaço vetorial tri-dimensional $V = \mathbf{Q}^3$ da seguinte forma:

```
sage: V = VectorSpace(QQ,3)
sage: V
Vector space of dimension 3 over Rational Field
```

Você pode usar a seguinte notação mais compacta:

```
sage: V = QQ^3
```

Então é fácil listar todas as funções para V usando completamento. Digite `V`, e então pressione a tecla `[tab]` no seu teclado:

```
sage: V.[tab key]
V._VectorSpace_generic__base_field
...
V.ambient_space
V.base_field
V.base_ring
V.basis
V.coordinates
...
V.zero_vector
```

Se você digitar as primeiras letras de uma função, e então a tecla `[tab]`, você obtém apenas funções que começam conforme indicado.

```
sage: V.i[tab key]
V.is_ambient  V.is_dense    V.is_full     V.is_sparse
```

Se você gostaria de saber o que uma função faz, por exemplo, a função coordinates, digite `V.coordinates?` para ajuda ou `V.coordinates??` para ver o código fonte, como explicado na próxima sessão.

3.8 Sistema de Ajuda Integrado

O Sage possui um sistema de ajuda integrado. Digite o nome da função seguido de ? para ver informações sobre a função.

```
sage: V = QQ^3
sage: V.coordinates?
Type:           instancemethod
Base Class:     <type 'instancemethod'>
String Form:    <bound method FreeModule_ambient_field.coordinates of Vector
space of dimension 3 over Rational Field>
Namespace:      Interactive
File:           /home/was/s/local/lib/python2.4/site-packages/sage/modules/f
ree_module.py
Definition:     V.coordinates(self, v)
Docstring:
    Write v in terms of the basis for self.

    Returns a list c such that if B is the basis for self, then

            sum c_i B_i = v.

    If v is not in self, raises an ArithmeticError exception.

    EXAMPLES:
        sage: M = FreeModule(IntegerRing(), 2); M0,M1=M.gens()
        sage: W = M.submodule([M0 + M1, M0 - 2*M1])
        sage: W.coordinates(2*M0-M1)
        [2, -1]
```

Como mostrado acima, o comando de ajuda mostra o tipo do objeto, o arquivo no qual ele é definido, e uma descrição útil da função com exemplos que você pode colar na sua sessão atual. Quase todos esses exemplos são regularmente testados automaticamente para certificar que eles se comportam exatamente como esperado.

Outro recurso que vai muito na direção do espírito de código aberto do Sage é que se `f` é uma função do Python, então o comando `f??` mostra o código fonte que define `f`. Por exemplo,

```
sage: V = QQ^3
sage: V.coordinates??
Type:           instancemethod
...
Source:
def coordinates(self, v):
        """
        Write $v$ in terms of the basis for self.
        ...
        """
        return self.coordinate_vector(v).list()
```

Isso nos diz que tudo que a função `coordinates` faz é chamar a função `coordinate_vector` e converter o resultado para uma lista. O que a função `coordinate_vector` faz?

```
sage: V = QQ^3
sage: V.coordinate_vector??
...
def coordinate_vector(self, v):
        ...
```

(continues on next page)

(continuação da página anterior)

```
        return self.ambient_vector_space()(v)
```

A função `coordinate_vector` coage a sua entrada em um espaço ambiente, o que tem o efeito de calcular o vetor de coeficientes de v em termos de V. O espaço V já é o espaço ambiente pois é simplesmente $\mathbf{Q}^3$. Existe também uma função `coordinate_vector` para subespaços, que é diferente. Vamos criar um subespaço e ver:

```
sage: V = QQ^3; W = V.span_of_basis([V.0, V.1])
sage: W.coordinate_vector??
...
def coordinate_vector(self, v):
        """
          ...
        """
        # First find the coordinates of v wrt echelon basis.
        w = self.echelon_coordinate_vector(v)
        # Next use transformation matrix from echelon basis to
        # user basis.
        T = self.echelon_to_user_matrix()
        return T.linear_combination_of_rows(w)
```

(Se você acha que a implementação é ineficiente, por favor junte-se a nós para ajudar a optimizar as funções de álgebra linear.)

Você também pode digitar `help(command_name)` ou `help(class)` para ler um arquivo de ajuda sobre determinada classe.

```
sage: help(VectorSpace)
Help on class VectorSpace ...

class VectorSpace(__builtin__.object)
 |  Create a Vector Space.
 |
 |  To create an ambient space over a field with given dimension
 |  using the calling syntax ...
 :
 :
```

Quando você digita `q` para sair do sistema de ajuda, a sua sessão aparece na tela da mesma forma que anteriormente. O texto de ajuda não fica permanentemente em sua tela, ao contrário da saída de `function_name?` que as vezes fica. É partircularmente útil digitar `help(module_name)`. Por exemplo, espaços vetoriais são definidos em `sage.modules.free_module`, então digite `help(sage.modules.free_module)` para obter documentação sobre esse módulo. Quando visualizando documentação usando a ajuda, você pode procurar no texto digitando `/` e na ordem reversa digitando `?`.

3.9 Salvando e Carregando Objetos Individuais

Suponha que você calcule uma matriz, ou pior ainda, um espaço complicado de símbolos, e gostaria de salvá-los para uso posterior. O que você pode fazer? Existem várias estratégias que os sistemas computacionais de álgebra adotam para salvar objetos individuais.

1. **Salve seus cálculos:** Suportam apenas salvar e carregar uma sessão completa (por exemplo, GAP, Magma).
2. **Entrada e Saída Unificadas:** Faz com que cada objeto seja impresso de uma forma que possa ser lido novamente (GP/PARI).

3. **Eval:** Torna fácil processar um código arbitrário no interpretador (por exemplo, Singular, PARI).

Como o Sage usa o Python, ele adota uma estratégia diferente, que se baseia no fato de que qualquer objeto pode ser "serializado", isto é, transformado em uma string a partir da qual o objeto pode ser recuperado. Isso segue o espírito da estratégia unificada de entrada e saída do PARI, exceto que não possue a desvantagem que os objetos são impressos na tela em uma forma muito complicada. Além disso, o suporte para salvar e recuperar é (na maior parte dos casos) completamente automática, não requerendo nenhuma programação extra; é simplesmente um recurso do Python que foi implementado na linguagem desde o início de seu desenvolvimento.

Quase todos os objetos `x` podem ser armazenados em disco de forma comprimida usando `save(x, filename)` (ou em muitos casos `x.save(filename)`). Para carregar o objeto de volta no Sage use `load(filename)`.

```
sage: A = MatrixSpace(QQ,3)(range(9))^2
sage: A
[ 15  18  21]
[ 42  54  66]
[ 69  90 111]
sage: save(A, 'A')
```

Você deve agora sair do Sage e reiniciá-lo. Então você pode obter `A` de volta:

```
sage: A = load('A')
sage: A
[ 15  18  21]
[ 42  54  66]
[ 69  90 111]
```

Você pode fazer o mesmo com objetos mais complicados, por exemplo, curvas elípticas. Todos os dados sobre o objeto são guardados e restaurados com o objeto. Por exemplo,

```
sage: E = EllipticCurve('11a')
sage: v = E.anlist(100000)                # takes a while
sage: save(E, 'E')
sage: quit
```

A versão em disco de `E` ocupa 153 kilobytes, pois ela guarda os primeiros 1000000 a_n com ela.

```
~/tmp$ ls -l E.sobj
-rw-r--r--  1 was was 153500 2006-01-28 19:23 E.sobj
~/tmp$ sage [...]
sage: E = load('E')
sage: v = E.anlist(100000)                # instant!
```

(Em Python, salvar e restaurar é feito usando o módulo `cPickle`. Em particular, um objeto `x` do Sage pode ser salvo usando `cPickle.dumps(x, 2)`. Note o `2`!)

O sage não pode salvar e carregar objetos criados em algum outro sistema computacional de álgebra, por exemplo, GAP, Singular, Maxima, etc. Eles são carregados em um estado "inválido". Em GAP, embora muitos objetos podem ser impressos de uma forma que eles podem ser reconstruídos, muitos não, logo reconstrução a partir de suas representações impressas não é permitido.

```
sage: a = gap(2)
sage: a.save('a')
sage: load('a')
Traceback (most recent call last):
...
ValueError: The session in which this object was defined is no longer
running.
```

Objetos do GP/PARI também podem ser salvos e carregados pois suas representações em forma impressa são suficientes para reconstruí-los.

```
sage: a = gp(2)
sage: a.save('a')
sage: load('a')
2
```

Objetos que foram salvos podem ser abertos posteriormente em computadores com arquiteturas e sistemas operacionais diferentes, por exemplo, você poderia salvar uma matriz muito grande em um OS X de 32-bits e abri-lo em um Linux de 64-bits, encontrar a forma reduzida, e movê-lo de volta. Além disso, em muitos casos você pode até mesmo abrir objetos em versões do Sage diferentes daquela no qual o objeto foi salvo, desde que o código para aquele objeto não seja muito diferente. Todos os atributos do objetos são armazendos, assim como a classe (mas não o código fonte) que define o objeto. Se aquela classe não existir mais em uma nova versão do Sage, então o objeto não pode ser reaberto nessa versão. Mas você poderia abri-lo em uma versão mais antiga, obter o dicionário do objeto (com `x.__dict__`), salvar o dicionário, e abri-lo em uma versão mais nova.

3.9.1 Salvando como Texto

Você também pode salvar a representação em texto (ASCII) de objetos em um arquivo, simplesmente abrindo um arquivo em modo de escrita, e escrevendo a string que representa o objeto no arquivo (você pode salvar mais de um objeto dessa forma). Quando você terminar de escrever os objetos, feche o arquivo.

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: f = (x+y)^7
sage: o = open('file.txt','w')
sage: o.write(str(f))
sage: o.close()
```

3.10 Salvando e Abrindo Sessões Completas

O Sage é flexível para salvar e abrir sessões completas.

O comando `save_session(sessionname)` salva todas as variáveis que você definiu na sessão atual como um dicionário com o nome `sessionname`. (No caso raro de uma variável não poder ser salva, ela simplesmente não aparece no dicionário.) O resultado é um arquivo `.sobj` que pode ser aberto como qualquer outro objeto que foi salvo. Quando você abre os objetos que foram salvos em uma sessão, você obtém um dicionário cujas chaves (keys) são os nomes das variáveis e os valores são os objetos.

Você pode usar o comando `load_session(sessionname)` para carregar na presente sessão as variáveis definidas em `sessioname`. Note que isso não remove as variáveis já definidas na presente sessão; em vez disso, as duas sessões são combinadas.

Primeiro iniciamos o Sage e definimos algumas variáveis.

```
sage: E = EllipticCurve('11a')
sage: M = ModularSymbols(37)
sage: a = 389
sage: t = M.T(2003).matrix(); t.charpoly().factor()
 _4 = (x - 2004) * (x - 12)^2 * (x + 54)^2
```

A seguir nós salvamos a nossa sessão, o que armazena cada uma das variáveis acima em um arquivo. Então visualizamos o arquivo, que tem por volta de 3K bytes.

```
sage: save_session('misc')
Saving a
Saving M
Saving t
Saving E
sage: quit
was@form:~/tmp$ ls -l misc.sobj
-rw-r--r--  1 was was 2979 2006-01-28 19:47 misc.sobj
```

Por fim reiniciamos o Sage, definimos algumas variáveis extra, e carregamos a sessão que foi salva anteriormente.

```
sage: b = 19
sage: load_session('misc')
Loading a
Loading M
Loading E
Loading t
```

Cada variável que foi salva está de novo disponível. Além disso, a variável `b` não foi redefinida.

```
sage: M
Full Modular Symbols space for Gamma_0(37) of weight 2 with sign 0
and dimension 5 over Rational Field
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational
Field
sage: b
19
sage: a
389
```

3.11 A Interface do Notebook

Esta seção refere-se ao notebook Sage legado, ou "sagenb".

SageMath está em transição para uso do Jupyter como padrão, que tem uma estrutura diferente. A diferença mais importante para os usuários é que as planilhas individuais no Jupyter são salvas no seu sistema local (como qualquer outro arquivo é salvo), enquanto que no tipo de notebook anterior Sage (ou sagenb) o principal ponto de acesso está nos arquivos descritos abaixo através do servidor.

3.11.1 Notebook Sage legado

O Sage Notebook é iniciado digitando

```
sage: notebook()
```

na linha de comando do Sage. Isso inicia o Notebook e abre o seu navegador padrão para visualizá-lo. Os arquivos de estado do servidor são armazenados em `$HOME/.sage/sage\_notebook.sagenb`.

Outras opções incluem:

```
sage: notebook("directory")
```

a qual inicia um novo servidor para o Notebook usando arquivos em um dado diretório `directory.sagenb`, em vez do diretório padrão `$HOME/.sage/sage_notebook.sagenb`. Isso pode ser útil se você quiser ter uma coleção de folhas de trabalho (worksheets) associadas com um projeto específico, ou executar vários Notebooks separadamente ao mesmo tempo.

Quando você inicia o Notebook, ele primeiro cria os seguintes arquivos em `$HOME/.sage/sage_notebook.sagenb`:

```
conf.pickle
openid.pickle
twistedconf.tac
sagenb.pid
users.pickle
home/admin/
home/guest/
home/pub/
```

Após criar os arquivos acima, o Notebook inicia o servidor web.

Um "Notebook" é uma coleção de contas de usuário, cada qual pode ter várias folhas de trabalho (worksheets). Quando você cria uma nova folha de trabalho, os dados dela são armazenados no diretórios `home/username/number`. Em cada diretório desse há um arquivo texto `worksheet.html` - se algum problema ocorrer com as suas folhas de trabalho, ou com o Sage, esse arquivo texto contém toda informação necessária para reconstruir a folha de trabalho.

A partir do Sage, digite `notebook?` para mais informações sobre como iniciar um servidor.

O seguinte diagrama ilustra a arquitetura do Notebook Sage:

```
----------------------
|                     |
|                     |
|   firefox/safari    |
|                     |
|     javascript      |
|      program        |
|                     |
|                     |
----------------------
      |      ^
      | AJAX |
      V      |
----------------------
|                     |
|       sage          |                SAGE process 1
|       web           | ------------>  SAGE process 2    (Python processes)
|      server         |   pexpect      SAGE process 3
|                     |                    .
|                     |                    .
----------------------                    .
```

Para ajuda sobre as teclas de atalho disponíveis no Notebook, clique no link `Help`.

CAPÍTULO 4

Interfaces

Uma característica central do Sage é que ele permite fazer cálculos com objetos em vários sistemas de álgebra computacional usando uma interface comum e uma linguagem de programação clara.

Os métodos console e interact de uma interface executam tarefas bem diferentes. Por exemplo, usando GAP:

1. `gap.console()`: Isso abre um console do GAP - o controle é transferido para o GAP. Aqui o Sage não é nada mais do que uma forma conveniente de executar um programa, similar à shell Bash do Linux.
2. `gap.interact()`: Essa é uma forma de interagir com uma instância do GAP que pode estar cheia de objetos do Sage. Você pode importar objetos nessa seção do GAP (até mesmo a partir da interface interativa), etc.

4.1 GP/PARI

O PARI é um programa em C muito compacto, maduro, e extremamente otimizado cujo foco primário é teoria de números. Existem duas interfaces distintas que podem ser usadas no Sage:

- `gp` - o "**G** do **P** ARI" interpretador, e
- `pari` - a biblioteca C do PARI.

Por exemplo, os seguintes comandos são duas formas de realizar a mesma coisa. Eles parecem idênticos, mas o resultado é na verdade diferente, e o que acontece por trás da cena é bastante diferente.

```
sage: gp('znprimroot(10007)')
Mod(5, 10007)
sage: pari('znprimroot(10007)')
Mod(5, 10007)
```

No primeiro caso, uma cópia separada do interpretador GP é iniciada como um servidor, e a string `znprimroot(10007)` é enviada, calculada pelo GP, e o resultado é armazenado em uma variável no GP (que ocupa espaço na memória dos processos do GP que não serão liberados). Então o valor dessa variável é exibido. No segundo caso, nenhum programa separado é iniciado, e a string `znprimroot(10007)` é calculada por uma certa função da biblioteca C do PARI. O resultado é armazenado na memória em uso pelo Python, que é liberada quando a variável não for mais referenciada. Os objetos possuem tipos diferentes:

```
sage: type(gp('znprimroot(10007)'))
<class 'sage.interfaces.gp.GpElement'>
sage: type(pari('znprimroot(10007)'))
<type 'cypari2.gen.Gen'>
```

Então qual eu devo usar? Depende do que você está fazendo. A interface GP pode fazer absolutamente tudo o que você poderia fazer na linha de comando do GP/PARI, pois está simplesmente executando esse programa. Em particular, você pode carregar programas complicados em PARI e executá-los. Por outro lado, a interface do PARI (via a biblioteca C) é muito mais restritiva. Primeiro, nem todas as funções foram implementadas. Segundo, bastante código, por exemplo, envolvendo integração numérica, não irá funcionar através da interface PARI. Todavia, a interface PARI pode ser significamente mais rápida e mais robusta do que a interface GP.

(Se a interface GP ficar sem memória para calcular algum comando, ela irá silenciosamente e automaticamente duplicar a memória alocada e repetir o comando solicitado. Então os seus cálculos não irão ser interrompidos se você não antecipou corretamente a quantidade de memória que seria necessária. Esse é um truque útil que a interface usual do GP não parece fornecer. Com respeito à interface da biblioteca C do PARI, ela imediatamente copia cada objeto criado para fora da pilha de memória, logo essa pilha nunca irá crescer. Contudo, cada objeto não deve exceder 100MB de tamanho, ou a pilha irá estourar quando o objeto for criado. Essa procedimento de cópia impõe uma leve pena sobre a performace.)

Em resumo, o Sage usa a biblioteca C do pari para fornecer funcionalidade similar à fornecida pelo interpretador GP/PARI, exceto que com um gerenciamento sofisticado de memória e a linguagem de programação Python.

Primeiro criamos uma lista do PARI a partir de uma lista do Python.

```
sage: v = pari([1,2,3,4,5])
sage: v
[1, 2, 3, 4, 5]
sage: type(v)
<type 'cypari2.gen.Gen'>
```

Cada objeto do PARI é do tipo `Gen`. O tipo PARI do objeto subjacente pode ser obtido usando a função `type`.

```
sage: v.type()
't_VEC'
```

Em PARI, para criar uma curva elíptica digitamos `ellinit([1,2,3,4,5])`. Em Sage é similar, exceto que `ellinit` é um método que pode ser chamado em qualquer objeto do PARI, por exemplo, `t_VEC` v.

```
sage: e = v.ellinit()
sage: e.type()
't_VEC'
sage: pari(e)[:13]
[1, 2, 3, 4, 5, 9, 11, 29, 35, -183, -3429, -10351, 6128487/10351]
```

Agora que temos um objeto de curva elíptica, podemos calcular algumas coisas a respeito dele.

```
sage: e.elltors()
[1, [], []]
sage: e.ellglobalred()
[10351, [1, -1, 0, -1], 1, [11, 1; 941, 1], [[1, 5, 0, 1], [1, 5, 0, 1]]]
sage: f = e.ellchangecurve([1,-1,0,-1])
sage: f[:5]
[1, -1, 0, 4, 3]
```

4.2 GAP

O Sage vem com o GAP para matemática discreta computacional, especialmente teoria de grupos.

Aqui está um exemplo com a função `IdGroup` do GAP, a qual usa a base de dados opcional sobre grupos que precisa ser instalada separadamente, como explicado abaixo.

```
sage: G = gap('Group((1,2,3)(4,5), (3,4))')
sage: G
Group( [ (1,2,3)(4,5), (3,4) ] )
sage: G.Center()
Group( () )
sage: G.IdGroup()
[ 120, 34 ]
sage: G.Order()
120
```

Podemos realizar os mesmos cálculos no Sage sem explicitamente evocar a interface do GAP da seguinte forma:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.center()
Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by␣
↪[()]
sage: G.group_id()
[120, 34]
sage: n = G.order(); n
120
```

Para algumas funcionalidades do GAP, deve-se instalar um pacote Sage opcional. Isso pode ser feito com o comando:

```
sage -i gap_packages
```

4.3 Singular

O Singular fornece uma biblioteca massiva e madura para bases de Gröbner, máximo divisor comum para polinômios em várias variáveis, bases de espaços de Riemann-Roch de uma curva plana, e fatorização, entre outras coisas. Vamos ilustrar a fatorização de polinômios em várias variáveis usando a interface do Sage para o Singular (não digite `...`):

```
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: R1
polynomial ring, over a field, global ordering
//   coefficients: QQ
//   number of vars : 2
//        block   1 : ordering dp
//                  : names    x y
//        block   2 : ordering C
sage: f = singular('9*y^8 - 9*x^2*y^7 - 18*x^3*y^6 - 18*x^5*y^6 +'
....:     '9*x^6*y^4 + 18*x^7*y^5 + 36*x^8*y^4 + 9*x^10*y^4 - 18*x^11*y^2 -'
....:     '9*x^12*y^3 - 18*x^13*y^2 + 9*x^16')
```

Agora que definimos f, vamos imprimi-lo e fatorá-lo.

```
sage: f
9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^7*y^5-18*x^5*y^
↪6+9*x^6*y^4-18*x^3*y^6-9*x^2*y^7+9*y^8
```

(continues on next page)

(continuação da página anterior)

```
sage: f.parent()
Singular
sage: F = f.factorize(); F
[1]:
   _[1]=9
   _[2]=x^6-2*x^3*y^2-x^2*y^3+y^4
   _[3]=-x^5+y^2
[2]:
   1,1,2
sage: F[1][2]
x^6-2*x^3*y^2-x^2*y^3+y^4
```

Como com o exemplo para o GAP em *GAP*, podemos calcular a fatorização acima sem explicitamente usar a inteface do Singular (todavia, implicitamente o Sage usa a interface do Singular para os cálculos). Não digite `...`:

```
sage: x, y = QQ['x, y'].gens()
sage: f = 9*y^8 - 9*x^2*y^7 - 18*x^3*y^6 - 18*x^5*y^6 + 9*x^6*y^4 \
....:  + 18*x^7*y^5 + 36*x^8*y^4 + 9*x^10*y^4 - 18*x^11*y^2 - 9*x^12*y^3 \
....:  - 18*x^13*y^2 + 9*x^16
sage: factor(f)
(9) * (-x^5 + y^2)^2 * (x^6 - 2*x^3*y^2 - x^2*y^3 + y^4)
```

4.4 Maxima

O Maxima está incluido no Sage, assim como uma implementação do Lisp. O pacote gnuplot (que o Maxima usa para criar gráficos) é distribuído como um pacote adicional do Sage. Entre outras coisas, o Maxima executa manipulações simbólicas. Ele pode integrar e diferenciar funções simbolicamente, resolver EDOs de primeira ordem, grande parte das EDOs lineares de segunda ordem, e tem implementado o método da transformada de Laplace para EDOs lineares de qualquer ordem. O Maxima também suporta uma série de funções especiais, é capaz de criar gráficos via gnuplot, e possui métodos para resolver equações polinômiais e manipular matrizes (por exemplo, escalonar e calcular autovalores e autovetores).

Nós ilustramos a interface Sage/Maxima construíndo uma matriz cuja entrada i, j é i/j, para $i, j = 1, \ldots, 4$.

```
sage: f = maxima.eval('ij_entry[i,j] := i/j')
sage: A = maxima('genmatrix(ij_entry,4,4)'); A
matrix([1,1/2,1/3,1/4],[2,1,2/3,1/2],[3,3/2,1,3/4],[4,2,4/3,1])
sage: A.determinant()
0
sage: A.echelon()
matrix([1,1/2,1/3,1/4],[0,0,0,0],[0,0,0,0],[0,0,0,0])
sage: A.eigenvalues()
[[0,4],[3,1]]
sage: A.eigenvectors()
[[[0,4],[3,1]],[[[1,0,0,-4],[0,1,0,-2],[0,0,1,-4/3]],[[1,2,3,4]]]]
```

Aqui vai outro exemplo:

```
sage: A = maxima("matrix ([1, 0, 0], [1, -1, 0], [1, 3, -2])")
sage: eigA = A.eigenvectors()
sage: V = VectorSpace(QQ,3)
sage: eigA
[[[-2,-1,1],[1,1,1]],[[[0,0,1]],[[0,1,3]],[[1,1/2,5/6]]]]
sage: v1 = V(sage_eval(repr(eigA[1][0][0]))); lambda1 = eigA[0][0][0]
```

(continues on next page)

(continuação da página anterior)

```
sage: v2 = V(sage_eval(repr(eigA[1][1][0]))); lambda2 = eigA[0][0][1]
sage: v3 = V(sage_eval(repr(eigA[1][2][0]))); lambda3 = eigA[0][0][2]

sage: M = MatrixSpace(QQ,3,3)
sage: AA = M([[1,0,0],[1, - 1,0],[1,3, - 2]])
sage: b1 = v1.base_ring()
sage: AA*v1 == b1(lambda1)*v1
True
sage: b2 = v2.base_ring()
sage: AA*v2 == b2(lambda2)*v2
True
sage: b3 = v3.base_ring()
sage: AA*v3 == b3(lambda3)*v3
True
```

Por fim, apresentamos um exemplo de como usar o Sage para criar gráficos usando `openmath`. Alguns desses exemplos são modificações de exemplos do manual de referência do Maxima.

Um gráfico em duas dimensões de diversas funções (não digite `...`):

```
sage: maxima.plot2d('[cos(7*x),cos(23*x)^4,sin(13*x)^3]','[x,0,1]',  # not tested
....: '[plot_format,openmath]') # not tested
```

Um gráfico em 3D que você pode mover com o seu mouse:

```
sage: maxima.plot3d("2^(-u^2 + v^2)", "[u, -3, 3]", "[v, -2, 2]",  # not tested
....: '[plot_format, openmath]') # not tested

sage: maxima.plot3d("atan(-x^2 + y^3/4)", "[x, -4, 4]", "[y, -4, 4]", # not tested
....: "[grid, 50, 50]",'[plot_format, openmath]') # not tested
```

O próximo gráfico é a famosa faixa de Möbious:

```
sage: maxima.plot3d("[cos(x)*(3 + y*cos(x/2)), sin(x)*(3 + y*cos(x/2))," \ # not␣
↪tested
....: "y*sin(x/2)]", "[x, -4, 4]", "[y, -4, 4]", # not tested
....: '[plot_format, openmath]') # not tested
```

E agora a famosa garrafa de Klein:

```
sage: maxima("expr_1: 5*cos(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)+ 3.0)"\
....: "- 10.0")
5*cos(x)*(sin(x/2)*sin(2*y)+cos(x/2)*cos(y)+3.0)-10.0
sage: maxima("expr_2: -5*sin(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)+ 3.0)")
-5*sin(x)*(sin(x/2)*sin(2*y)+cos(x/2)*cos(y)+3.0)
sage: maxima("expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y))")
5*(cos(x/2)*sin(2*y)-sin(x/2)*cos(y))
sage: maxima.plot3d("[expr_1, expr_2, expr_3]", "[x, -%pi, %pi]", # not tested
....: "[y, -%pi, %pi]", "['grid, 40, 40]", # not tested
....: '[plot_format, openmath]') # not tested
```

CAPÍTULO 5

Sage, LaTeX e Companheiros

AUTOR: Rob Beezer (2010-05-23)

O Sage e o dialeto LaTeX do TeX tem um relacionamento sinergético intenso. Esta seção tem como objetivo introduzir as diversas formas de interação entre eles, começado pelas mais básicas e indo até as menos usuais. (Logo você pode não querer ler esta seção inteira na sua primeira passagem por este tutorial.)

5.1 Panorama Geral

Pode ser mais fácil entender os vários usos do LaTeX com um panorama geral sobre os três principais métodos usados pelo Sage.

1. Todo objeto no Sage possui uma representação em LaTeX. Você pode acessar essa representação executando, no Notebook ou na linha de comando do Sage, `latex(foo)` where `foo` é algum objeto no Sage. O resultado é uma string que deve fornecer uma representação razoável de `foo` no modo matemático em LaTeX (por exemplo, quando cercado por um par de símbolos $). Alguns exemplos disso seguem abaixo.

   Dessa forma, o Sage pode ser usado efetivamente para construir partes de um documento LaTeX: crie ou calcule um objeto no Sage, imprima `latex()` do objeto e copie-e-cole o resultado no seu documento.

2. A interface Notebook é configurada para usar o MathJax para representar fórmulas matemáticas de forma clara em um web navegador. O MathJax é uma coleção de rotinas em JavaScript e fontes associadas. Tipicamente esses fontes ficam armazenadas em um servidor e são enviadas para o navegador juntamente com a página onde elas estão sendo usadas. No caso do Sage, o Notebook está sempre conectado a um servidor usado para executar os comando do Sage, e esse servidor também fornece as fontes do MathJax necessárias. Logo não é necessário configurar nada mais para ter formulas matemáticas representadas no seu navegador quando você usa o Notebook do Sage.

   O MathJax é implementado para representar um subconjunto grande, mas não completo, do TeX. Ele não suporta objetos como, por exemplo, tabelas complicadas e seções, e é focado para representar acuradamente pequenas fórmulas em TeX. A representação automática de fórmulas matemáticas no Notebook é obtida convertendo a representação `latex()` de um objeto (como descrito acima) em uma forma de HTML mais adequada ao MathJax.

Como o MathJax usa as suas próprias fontes de tamanho variável, ele é superior a outros métodos que convertem equações, ou outros pequenos trechos de TeX, em imagens estáticas.

3. Na linha de comando do Sage, ou no Notebook quando o código em LaTeX é complicado demais para o MathJax processar, uma instalação local do LaTeX pode ser usada. O Sage inclui quase tudo que você precisa para compilar e usar o Sage, mas uma exceção significativa é o TeX. Então nessas situações você precisa ter o TeX instalado, juntamente com algumas ferramentas de conversão, para usar os recursos completos.

Aqui nós demonstramos alguns usos básicos da função `latex()`.

```
sage: var('z')
z
sage: latex(z^12)
z^{12}
sage: latex(integrate(z^4, z))
\frac{1}{5} \, z^{5}
sage: latex('a string')
\text{\texttt{a{ }string}}
sage: latex(QQ)
\Bold{Q}
sage: latex(matrix(QQ, 2, 3, [[2,4,6],[-1,-1,-1]]))
\left(\begin{array}{rrr}
2 & 4 & 6 \\
-1 & -1 & -1
\end{array}\right)
```

A funcionalidade básica do MathJax é em sua maior parte automática no Notebook, mas nós podemos demonstrar esse suporte parcialmente com a classe `MathJax`. A função `eval` dessa classe converte um objeto do Sage em sua representação LaTeX e adiciona HTML que por sua vez evoca a classe "matemática" do CSS, a qual então emprega o MathJax.

```
sage: from sage.misc.latex import MathJax
sage: js = MathJax()
sage: var('z')
z
sage: js(z^12)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}z^{12}</
↪script></html>
sage: js(QQ)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Q}
↪</script></html>
sage: js(ZZ[x])
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Z}
↪[x]</script></html>
sage: js(integrate(z^4, z))
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\frac{1}
↪{5} \, z^{5}</script></html>
```

5.2 Uso Básico

Como indicado acima, a forma mais simples de explorar o suporte do Sage para o LaTeX é usando a função `latex()` para criar código LaTeX para representar objetos matemáticos. Essas strings podem então ser incorporadas em documentos LaTeX. Isso funciona igualmente no Notebook ou na linha de comando do Sage.

No outro extremo está o comando `view()`. Na linha de comando do Sage o comando `view(foo)` irá criar a representação em LaTeX de `foo`, incorporar isso em um documento simples em LaTeX, e então processar o documento

usando o LaTeX em seu sistema. Por fim, o visualizador apropriado será aberto para apresentar o documento gerado. Qual versão do TeX é usada, e portanto as opções para a saída e visualizador, podem ser personalizados (veja *Personalizando o Processamento em LaTeX*).

No Notebook, o comando `view(foo)` cria uma combinação apropriada de HTML e CSS para que o MathJax mostre a representação em LaTeX na folha de trabalho. Para o usuário, ele simplesmente cria uma versão cuidadosamente formatada do resultado, distinta da saída padrão em modo texto do Sage. Nem todo objeto no Sage possui uma representação em LaTeX adequada às capacidades limitadas do MathJax. Nesses casos, a interpretação pelo MathJax pode ser deixada de lado, e com isso o LaTeX do sistema é chamado, e o resultado dessa chamada é convertido em uma imagem que é inserida na folha de trabalho. Como alterar e controlar esse processo é discutido abaixo na seção *Personalizando a Criação de Código LaTeX*.

O comando interno `pretty_print()` ilustra a conversão de objetos do Sage para HTML que emprega o MathJax no Notebook.

```
sage: pretty_print(x^12)
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}x^{12}</script></html>
sage: pretty_print(integrate(sin(x), x))
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}-\cos\left(x\right)</
↪script></html>
```

O Notebook tem outros dois recursos para empregar o TeX. O primeiro é o botão "Typeset" bem acima da primeira célula da folha de trabalho, à direita dos quatro menus de opções. Quando selecionado, o resultado de qualquer cálculo vai ser interpretado pelo MathJax. Note que esse efeito não é retroativo – células calculadas anteriormente precisam ser recalculadas para ter o resultado representado pelo MathJax. Essencialmente, selecionar o botão "Typeset" é equivalente a aplicar o comando `view()` ao resultado de cada célula.

Um segundo recurso disponível no Notebook é possibilidade de inserir código TeX para fazer anotações na folha de trabalho. Quando o cursos esta posicionado entre células de modo que uma barra azul fica visível, então shift+clique irá abrir um mini processador de texto, TinyMCE. Isso permite digitar texto, usando um editor WSISYG para criar HTML e CSS. Logo é possível inserir texto formatado para complementar a folha de trabalho. Todavia, texto entre símbolos $, ou $$, é interpretado pelo MathJax como "inline" ou "display math" espectivamente.

5.3 Personalizando a Criação de Código LaTeX

Exitem várias formas de personalizar o código LaTeX gerado pelo comando `latex()`. No Notebook e na linha de comando existe um objeto pré-definido chamado `latex` que possui diversos métodos, os quais você pode listar digitando `latex.`, seguido da tecla tab (note a presença do ponto).

Um bom exemplo é o método `latex.matrix_delimiters`. Ele pode ser usado para alterar a notação de matrizes – parênteses grandes, colchetes, barras verticais. Nenhuma noção de estilo é enfatizada, você pode configurar como desejado. Observe como as barras invertidas usadas em LaTeX requerem uma barra adicional para que elas não sejam interpretadas pelo Python como um comando (ou seja, sejam implementadas simplesmente como parte de uma string.

```
sage: A = matrix(ZZ, 2, 2, range(4))
sage: latex(A)
\left(\begin{array}{rr}
0 & 1 \\
2 & 3
\end{array}\right)
sage: latex.matrix_delimiters(left='[', right=']')
sage: latex(A)
\left[\begin{array}{rr}
0 & 1 \\
2 & 3
```

(continues on next page)

(continuação da página anterior)

```
\end{array}\right]
sage: latex.matrix_delimiters(left='\\{', right='\\}')
sage: latex(A)
\left\{\begin{array}{rr}
0 & 1 \\
2 & 3
\end{array}\right\}
```

O método `latex.vector_delimiters` funciona de forma similar.

A forma como anéis e corpos comuns podem ser representados pode ser controlada pelo método `latex.blackboard_bold`. Esses conjuntos são representados por padrão em negrito, mas podem opcionalmente ser escritos em letras duplas como é comum em trabalhos escritos. Isso é obtido redefinindo a macro `\Bold{}` que faz parte do Sage.

```
sage: latex(QQ)
\Bold{Q}
sage: from sage.misc.latex import MathJax
sage: js = MathJax()
sage: js(QQ)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Q}
↪</script></html>

sage: latex.blackboard_bold(True)
sage: js(QQ)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbb{#1}}\Bold{Q}
↪</script></html>
sage: latex.blackboard_bold(False)
```

É possível aproveitar os recursos do TeX adicionando novas funções (macros em inglês) e novos pacotes. Primeiro, funções individuais podem ser adicionadas para serem usadas quando o MathJax interpreta pequenos trechos de códigos TeX no Notebook.

```
sage: latex.extra_macros()
''
sage: latex.add_macro("\\newcommand{\\foo}{bar}")
sage: latex.extra_macros()
'\\newcommand{\\foo}{bar}'
sage: var('x y')
(x, y)
sage: latex(x+y)
x + y
sage: from sage.misc.latex import MathJax
sage: js = MathJax()
sage: js(x+y)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}
↪\newcommand{\foo}{bar}x + y</script></html>
```

Macros adicionais usadas dessa forma serão também usadas eventualmente se a versão do TeX no seu sistema for usada para lidar com algo muito complicado para o MathJax. O comando `latex_extra_preamble` é usado para construir o preambulo de um documento completo em LaTeX. Ilustramos a seguir como fazer isso. Novamente note a necessidade de barras invertidas duplas nas strings do Python.

```
sage: latex.extra_macros('')
sage: latex.extra_preamble('')
sage: from sage.misc.latex import latex_extra_preamble
```

(continues on next page)

(continuação da página anterior)

```
sage: print(latex_extra_preamble())
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
sage: latex.add_macro("\\newcommand{\\foo}{bar}")
sage: print(latex_extra_preamble())
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
\newcommand{\foo}{bar}
```

Novamente, para expressões grandes ou mais complicadas do LaTeX, é possível adicionar pacotes (ou qualquer outra coisa) ao preambulo do arquivo LaTeX. Qualquer coisa pode ser incorporada no preambulo com o comando `latex.add_to_preamble`, e o comando mais especializado `latex.add_package_to_preamble_if_available` irá primeiro verificar se certo pacote está realmente disponível antes de adicioná-lo ao preambulo

Agora adicionamos o pacote geometry ao preambulo e usamos ele para definir o tamanho da região na página que o TeX vai usar (efetivamente definido as margens). Novamente, observe a necessidade de barras duplas nas strings do Python.

```
sage: from sage.misc.latex import latex_extra_preamble
sage: latex.extra_macros('')
sage: latex.extra_preamble('')
sage: latex.add_to_preamble('\\usepackage{geometry}')
sage: latex.add_to_preamble('\\geometry{letterpaper,total={8in,10in}}')
sage: latex.extra_preamble()
'\\usepackage{geometry}\\geometry{letterpaper,total={8in,10in}}'
sage: print(latex_extra_preamble())
\usepackage{geometry}\geometry{letterpaper,total={8in,10in}}
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
```

Um pacote pode ser adicionado juntamente com a verificação de sua existência, da seguinte forma. Como um exemplo, nós ilustramos uma tentativa de adicionar ao preambulo um pacote que supostamente não existe.

```
sage: latex.extra_preamble('')
sage: latex.extra_preamble()
''
sage: latex.add_to_preamble('\\usepackage{foo-bar-unchecked}')
sage: latex.extra_preamble()
'\\usepackage{foo-bar-unchecked}'
sage: latex.add_package_to_preamble_if_available('foo-bar-checked')
sage: latex.extra_preamble()
'\\usepackage{foo-bar-unchecked}'
```

5.4 Personalizando o Processamento em LaTeX

É também possível controlar qual variação do TeX é usada quando a versão do sistema for evocada, logo influenciando também o resultado. De forma similar, é também possível controlar quando o Notebook irá usar o MathJax (trechos simples em TeX) ou a versão do TeX do sistema (expressões mais complicadas).

O comando `latex.engine()` pode ser usado para controlar de os executáveis `latex`, `pdflatex` ou `xelatex`

do sistema são usados para processar expressões mais complicadas. Quando `view()` é chamado na linha de comando do Sage e o processador é definido como `latex`, um arquivo dvi é produzido e o Sage vai usar um visualizador de dvi (como o xdvi) para apresentar o resultado. Por outro lado, usando `view()` na linha de comando do Sage, quando o processador é definido como `pdflatex`, irá produzir um PDF e o Sage vai executar o programa disponível no seu sistema para visualizar arquivos PDF (acrobat, okular, evince, etc.).

No Notebook, é necessário interver na decisão de se o MathJax vai interpretar trechos em TeX, ou se o LaTeX do sistema deve fazer o trabalho se o código em LaTeX for complicado demais. O dispositivo é uma lista de strings, que se forem encontradas em um trecho de código LaTeX sinalizam para o Notebook usar o LaTeX (ou qualquer executável que for definido pelo comando `latex.engine()`). Essa lista é gerenciada pelos comandos `latex.add_to_mathjax_avoid_list` e `latex.mathjax_avoid_list`.

```
sage: latex.mathjax_avoid_list([])
sage: latex.mathjax_avoid_list()
[]
sage: latex.mathjax_avoid_list(['foo', 'bar'])
sage: latex.mathjax_avoid_list()
['foo', 'bar']
sage: latex.add_to_mathjax_avoid_list('tikzpicture')
sage: latex.mathjax_avoid_list()
['foo', 'bar', 'tikzpicture']
sage: latex.mathjax_avoid_list([])
sage: latex.mathjax_avoid_list()
[]
```

Suponha que uma expressão em LaTeX é produzida no Notebook com o comando `view()` ou enquanto o botão "Typeset" está selecionado, e então reconhecida, através da "lista de comandos a serem evitados no MathJax", como necessitando a versão do LaTeX no sistema. Então o executável selecionado (como especificado por `latex.engine()`) irá processar o código em LaTeX. Todavia, em vez de então abrir um visualizador externo (o que é o comportamento na linha de comando), o Sage irá tentar converter o resultado em uma imagem, que então é inserida na folha de trabalho como o resultado da célula.

Exatamente como essa conversão é feita depende de vários fatores – qual executável você especificou como processador e quais utilitários de conversão estão disponíveis no seu sistema. Quatro conversores usuais que irão cobrir todas as ocorrências são o `dvips`, `ps2pdf`, e `dvipng`, e do pacote `ImageMagick`, o `convert`. O objetivo é produzir um arquivo PNG para ser inserido de volta na folha de trabalho. Quando uma expressão em LaTeX pode ser convertida com sucesso em um arquivo dvi pelo processador LaTeX, então o dvipng deve dar conta da conversão. Se a expressão em LaTeX e o processador especificado criarem um arquivo dvi com conteúdo especial que o dvipng não pode converter, então o dvips vai criar um arquivo PostScript. Esse arquivo PostScript, ou um PDF criado por pelo processador `pdflatex`, é então convertido em um arquivo dvi pelo programa `convert`. A presença de dois desses conversores pode ser testado com as rotinas `have_dvipng()` e `have_convert()`.

Essas conversões são feitas automaticamente se você tiver os conversores necessários instalados; se não, então uma mensagem de erro é impressa dizendo o que está faltando e onde obter.

Para um exemplo concreto de como expressões complicadas em LaTeX podem ser processadas, veja o exemplo na próxima seção (*Exemplo: Grafos Combinatoriais com tkz-graph*) para usar o pacote `tkz-graph` para produzir ilustrações de grafos combinatoriais de alta qualidade. Para outros exemplos, existem alguns casos teste incluídos no Sage. Para usá-los, é necessário importar o objeto `sage.misc.latex.latex_examples`, que é uma instância da classe `sage.misc.latex.LatexExamples`, como mostrado abaixo. Essa classe possui exemplos de diagramas comutativos, grafos combinatoriais, teoria de nós e pstricks, os quais respectivamente testam os seguintes pacotes: xy, tkz-graph, xypic, pstricks. Após importar o objeto, use completamento tab em `latex_examples` para ver os exemplos disponíveis. Ao carregar um exemplo você irá obter explicações sobre o que é necessário para fazer o conteúdo do exemplo ser exibido corretamente. Para de fato ver os exemplos, é necessário usar `view()` (uma vez que o preambulo, processador, etc. estão configurados corretamente).

```
sage: from sage.misc.latex import latex_examples
sage: latex_examples.diagram()
LaTeX example for testing display of a commutative diagram produced
by xypic.

To use, try to view this object -- it won't work.  Now try
'latex.add_to_preamble("\\usepackage[matrix,arrow,curve,cmtip]{xy}")',
and try viewing again -- it should work in the command line but not
from the notebook.  In the notebook, run
'latex.add_to_mathjax_avoid_list("xymatrix")' and try again -- you
should get a picture (a part of the diagram arising from a filtered
chain complex).
```

5.5 Exemplo: Grafos Combinatoriais com tkz-graph

Ilustrações de alta qualidade de grafos combinatoriais (daqui por diante, simplesmente grafos) são possíveis com o pacote `tkz-graph`. Esse pacote baseia-se no `tikz` front-end da biblioteca `pgf`. Logo todos esses componentes precisam ser parte de uma instalação completa do LaTeX em seu sistema, e pode acontecer que alguns desses componentes não estejam em sua versão mais recente em algumas distribuições do TeX. Logo, para melhores resultados, seria necessário ou recomendável instalar esses pacotes como parte do seu diretório texmf pessoal. Criar, manter e personalizar uma instalação do TeX no sistema ou em um diretório pessoal vai além do escopo deste documento, mas deve ser fácil encontrar instruções para isso. Os arquivos necessários estão listados em *Uma Instalação Completa do TeX*.

Portanto, para começar precisamos nos certificar que os pacotes relevantes estão incluídos adicionando-os ao preambulo do eventual documento LaTeX. As imagens dos grafos não são formadas corretamente quando um arquivo dvi é usando como formato intermediário, logo é melhor definir o processador do LaTeX como `pdflatex`. A esta altura um comando como `view(graphs.CompleteGraph(4))` deve funcionar na linha de comando do Sage e produzir um PDF com a imagem completa do grafo K_4.

Para uma experiência semelhante no Notebook, é necessário desabilitar o processador MathJax para o código LaTeX do grafo usando a "lista de comandos a serem evitados pelo MathJax". Grafos são criados usando o ambiente `tikzpicture`, logo essa uma boa escolha para uma string a ser incluída na lista que acabamos de mencionar. Agora, `view(graphs.CompleteGraph(4))` em uma folha de trabalho deve executar o pdflatex para criar um PDF e então o programa `convert` para obter um gráfico PNG que vai ser inserido na folha de trabalho. Os seguintes comandos ilustram os passos para obter grafos processados pelo LaTeX no Notebook.

```
sage: from sage.graphs.graph_latex import setup_latex_preamble
sage: setup_latex_preamble()
sage: latex.extra_preamble() # random - depends on system's TeX installation
'\\usepackage{tikz}\n\\usepackage{tkz-graph}\n\\usepackage{tkz-berge}\n'
sage: latex.engine('pdflatex')
sage: latex.add_to_mathjax_avoid_list('tikzpicture')
sage: latex.mathjax_avoid_list()
['tikz', 'tikzpicture']
```

Agora, um comando como `view(graphs.CompleteGraph(4))` deve produzir um gráfico do grafo no Notebook, tendo usado `pdflatex` para processar os comandos do `tkz-graph` para construir o grafo. Note que há diversas opções que afetam o resultado do gráfico obtido usando o LaTeX via `tkz-graph`, o que mais uma vez está além do escopo desta seção (veja a seção do Manual de Referência com título "Opções do LaTeX para Grafos" para instruções e detalhes).

5.6 Uma Instalação Completa do TeX

Vários dos recursos avançados de integração do TeX com o Sage requerem uma instalação do TeX em seu sistema. Várias versões do Linux possuem pacotes do TeX baseados no TeX-live, para o OSX existe o TeXshop e para o windows existe o MikTex. O utilitário `convert` é parte do ImageMagick (que deve ser um pacote na sua versão do Linux ou ser fácil de instalar), e os três programas `dvipng`, `ps2pdf`, e `dvips` podem estar incluídos na sua distribuição do TeX. Os dois primeiros podem também ser obtidos em, respectivamente, http://sourceforge.net/projects/dvipng/ e como parte do Ghostscript.

A criação de grafos combinatoriais requer uma versão recente da biblioteca PGF, e os arquivos `tkz-graph.sty`, `tkz-arith.sty` e talvez `tkz-berge.sty`, que estão disponíveis em Altermundus site.

5.7 Programas Externos

Existem três programas disponíveis para integrar ainda mais o TeX e o Sage. O primeiro é o sagetex. Uma descrição concisa do sagetex é que ele é uma coleção de funções do TeX que permitem incluir em um documento LaTeX instruções para usar o Sage para calcular vários objetos, e/ou formatar objetos usando o comando `latex()` existente no Sage. Logo, como um passo intermediário para compilar um documento LaTeX, todos os recursos computacionais e de formatação do Sage podem ser executados automaticamente. Como um exemplo, um exame matemático pode manter uma correspondência entre questões e respostas usando o sagetex para fazer cálculos com o Sage. Veja *Usando o SageTeX* para mais informações.

O tex2sws começa com um documento LaTeX, mas define ambientes adicionais para inserir código em Sage. Quando processado com as ferramentas adequadas, o resultado é uma folha de trabalho do Sage, com conteúdo apropriadamente formatado para o MathJax e com código em Sage incorporado como células de entrada. Então um livro texto ou artigo pode ser criado em LaTeX, ter blocos de código em Sage incluídos, e o documento todo pode ser transformado em uma folha de trabalho do Sage onde o texto matemático é bem formatado e os blocos de código em Sage podem ser facilmente executados. Atualmente em desenvolvimento, veja tex2sws @ BitBucket para mais informações.

O sws2tex reverte o processo partindo de uma folha de trabalho do Sage e convertendo o conteúdo para LaTeX para ser posteriormente processado com as ferramentas disponíveis para documentos em LaTeX. Atualmente em desenvolvimento, veja sws2tex @ BitBucket para mais informações.

CAPÍTULO 6

Programação

6.1 Carregando e Anexando Arquivos do Sage

A seguir ilustramos como carregar no Sage programas escritos em um arquivo separado. Crie um arquivo chamado `example.sage` com o seguinte conteúdo:

```
print("Hello World")
print(2^3)
```

Você pode ler e executar o arquivo `example.sage` usando o comando `load`.

```
sage: load "example.sage"
Hello World
8
```

Você também pode anexar um arquivo em Sage à sessão em execução usando o comando `attach`.

```
sage: attach "example.sage"
Hello World
8
```

Agora se você alterar `example.sage` e adicionar uma linha em branco (por exemplo), então o conteúdo de `example.sage` será automaticamente recarregado no Sage.

Em particular, `attach` automaticamente recarrega um arquivo toda vez que ele for modificado, o que é útil para desenvolver e testar um programa, enquanto `load` carrega o arquivo apenas uma vez.

Quando o Sage carrega `example.sage` ele converte esse arquivo para o Python, o qual é então executado pelo interpretador do Python. Essa conversão é mínima; ela essencialmente consiste em encapsular inteiros em `Integer()`, números float em `RealNumber()`, substituir `^` por `**`, e substituir, por exemplo, `R.2` por `R.gen(2)`. A versão convertida de `example.sage` é armazenada no mesmo diretório de `example.sage` e é chamada `example.sage.py`. Esse arquivo contém o seguinte código:

```
print("Hello World")
print(Integer(2)**Integer(3))
```

Inteiros literais são encapsulados e ^ é substituído por **. (Em Python, ^ significa "ou exclusivo" e ** significa "exponenciação".)

Esse "" está implementado em sage/misc/interpreter.py.)

Você pode colar código tabulado com muitas linhas no Sage desde que existam linhas em branco separando blocos de código (isso não é necessário em arquivos). Todavia, a melhor forma de adicionar tal código a uma sessão do Sage é salvá-lo em um arquivo e usar attach, como descrito anteriormente.

6.2 Criando Código Compilado

Velocidade é crucial em cálculos matemáticos. Embora o Python seja uma linguagem conveniente de alto nível, certos cálculos podem ser várias vezes mais rápidos do que em Python se eles forem implementados usando tipos estáticos em uma linguagem compilada. Alguns aspectos do Sage seriam muito lentos se eles fossem escritos inteiramente em Python. Para lidar com isso, o Sage suporta uma "versão" compilada do Python chamada Cython (*[Cyt]* and *[Pyr]*). O Cython é simultaneamente similar ao Python e ao C. A maior parte das construções em Python, incluindo "list comprehensions", expressões condicionais, código como += são permitidos; você também pode importar código que você escreveu em outros módulos em Python. Além disso, você pode declarar variáveis em C arbitrárias, e qualquer chamada de bibliotecas em C pode ser feita diretamente. O código resultante é convertido para C e compilado usando um compilador para C.

Para criar o seu próprio código compilado em Sage, nomeie o arquivo com uma extensão .spyx (em vez de .sage). Se você está trabalhando na linha de comando, você pode anexar e carregar código compilado exatamente como se faz com código interpretado (no momento, anexar e carregar código em Cython não é suportado no Notebook). A compilação é realizada implicitamente sem que você tenha que fazer qualquer coisa explicitamente. Veja $SAGE_ROOT/examples/programming/sagex/factorial.spyx para um exemplo de uma implementação compilada para a função fatorial que usa diretamente a biblioteca GMP em C. Experimente o seguinte, usando cd, vá para o diretório $SAGE_ROOT/examples/programming/sagex/, e então faça o seguinte:

```
sage: load "factorial.spyx"
**************************************************
               Recompiling factorial.spyx
**************************************************
sage: factorial(50)
30414093201713378043612608166064768844377641568960512000000000000L
sage: time n = factorial(10000)
CPU times: user 0.03 s, sys: 0.00 s, total: 0.03 s
Wall time: 0.03
```

Aqui o sufixo L indica um "long integer" do Python (veja *O Pré-Processador: Diferenças entre o Sage e o Python*).

Note que o Sage vai recompilar factorial.spyx se você encerrar e reiniciar o Sage. A biblioteca compilada e compartilhada é armazenada em $HOME/.sage/temp/hostname/pid/spyx. Esses arquivos são excluídos quando você encerra o Sage.

Nenhum pré-processamento é aplicado em arquivos spyx, por exemplo, 1/3 vai resultar em 0 em um arquivo spyx em vez do número racional $1/3$. Se foo é uma função da biblioteca Sage, para usá-la em um arquivo spyx importe sage.all e use sage.all.foo.

```
import sage.all
def foo(n):
    return sage.all.factorial(n)
```

6.2.1 Acessando Funções em C em Arquivos Separados

É fácil também acessar funções em C definidas em arquivos *.c separados. Aqui vai um exemplo. Crie os arquivos `test.c` e `test.spyx` no mesmo diretório contendo:

Código C puro: `test.c`

```
int add_one(int n) {
  return n + 1;
}
```

Código Cython: `test.spyx`:

```
cdef extern from "test.c":
    int add_one(int n)

def test(n):
    return add_one(n)
```

Então o seguinte funciona:

```
sage: attach "test.spyx"
Compiling (...)/test.spyx...
sage: test(10)
11
```

Se uma biblioteca `foo` adicional é necessária para compilar código em C gerado a partir de um arquivo em Cython, adicione a linha `clib foo` no arquivo fonte em Cython. De forma similar, um arquivo em C adicional `bar` pode ser incluído na compilação declarando `cfile bar`.

6.3 Scripts Independentes em Python/Sage

O seguinte script em Sage fatora inteiros, polinômios, etc:

```
#!/usr/bin/env sage

import sys
from sage.all import *

if len(sys.argv) != 2:
    print("Usage: %s <n>" % sys.argv[0])
    print("Outputs the prime factorization of n.")
    sys.exit(1)

print(factor(sage_eval(sys.argv[1])))
```

Para usar esse script, sua `SAGE_ROOT` precisa estar na sua variável PATH. Se o script acima for chamado `factor`, aqui está um exemplo de como usá-lo:

```
bash $ ./factor 2006
2 * 17 * 59
```

6.4 Tipo de Dados

Cada objeto em Sage possui um tipo bem definido. O Python possui diversos tipos de dados, e a biblioteca do Sage adiciona ainda mais. Os tipos de dados de Python incluem strings, listas, tuplas, inteiros e floats, como ilustrado:

```
sage: s = "sage"; type(s)
<... 'str'>
sage: s = 'sage'; type(s)      # you can use either single or double quotes
<... 'str'>
sage: s = [1,2,3,4]; type(s)
<... 'list'>
sage: s = (1,2,3,4); type(s)
<... 'tuple'>
sage: s = int(2006); type(s)
<... 'int'>
sage: s = float(2006); type(s)
<... 'float'>
```

Além disso, o Sage acrescenta vários outros tipos. Por exemplo, espaços vetoriais:

```
sage: V = VectorSpace(QQ, 1000000); V
Vector space of dimension 1000000 over Rational Field
sage: type(V)
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category'>
```

Apenas certas funções podem ser aplicadas sobre `V`. Em outros softwares de matemática, essas seriam chamadas usando a notação "funcional" `foo(V,...)`. Em Sage, algumas funções estão anexadas ao tipo (ou classe) de `V`, e são chamadas usando uma sintaxe orientada a objetos como em Java ou C++, por exemplo, `V.foo()`. Isso ajuda a manter o espaço de variáveis global sem milhares de funções, e permite que várias funções diferentes com comportamento diferente possam ser chamadas foo, sem a necessidade de usar um mecanismo de identificação de tipos (ou casos) para decidir qual chamar. Além disso, se você reutilizar o nome de uma função, essa função continua ainda disponível (por exemplo, se você chamar algo `zeta`, e então quiser calcular o valor da função zeta de Riemann em 0.5, você continua podendo digitar `s=.5; s.zeta()`).

```
sage: zeta = -1
sage: s=.5; s.zeta()
-1.46035450880959
```

Em alguns casos muito comuns, a notação funcional usual é também suportada por conveniência e porque expressões matemáticas podem parecer confusas usando a notação orientada a objetos. Aqui vão alguns exemplos.

```
sage: n = 2; n.sqrt()
sqrt(2)
sage: sqrt(2)
sqrt(2)
sage: V = VectorSpace(QQ,2)
sage: V.basis()
    [
    (1, 0),
    (0, 1)
    ]
sage: basis(V)
    [
    (1, 0),
    (0, 1)
    ]
```

(continues on next page)

(continuação da página anterior)

```
sage: M = MatrixSpace(GF(7), 2); M
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 7
sage: A = M([1,2,3,4]); A
[1 2]
[3 4]
sage: A.charpoly('x')
x^2 + 2*x + 5
sage: charpoly(A, 'x')
x^2 + 2*x + 5
```

Para listar todas as funções para A, use completamento tab. Simplesmente digite `A.`, então tecle `[tab]` no seu teclado, como descrito em *Busca Reversa e Completamento Tab*.

6.5 Listas, Tuplas e Sequências

O tipo de dados lista armazena elementos de um tipo arbitrário. Como em C, C++, etc. (mas diferentemente da maioria dos sistemas de álgebra computacional), os elementos da lista são indexados a partir do 0:

```
sage: v = [2, 3, 5, 'x', SymmetricGroup(3)]; v
[2, 3, 5, 'x', Symmetric group of order 3! as a permutation group]
sage: type(v)
<... 'list'>
sage: v[0]
2
sage: v[2]
5
```

(Quando se indexa uma lista, é permitido que o índice não seja um int do Python!) Um Inteiro do Sage (ou Racional, ou qualquer objeto que possua um método `__index__`) também ira funcionar.

```
sage: v = [1,2,3]
sage: v[2]
3
sage: n = 2       # SAGE Integer
sage: v[n]        # Perfectly OK!
3
sage: v[int(n)]   # Also OK.
3
```

A função `range` cria uma lista de int's do Python (não Inteiros do Sage):

```
sage: range(1, 15)  # py2
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Isso é útil quando se usa "list comprehensions" para construir listas:

```
sage: L = [factor(n) for n in range(1, 15)]
sage: L
[1, 2, 3, 2^2, 5, 2 * 3, 7, 2^3, 3^2, 2 * 5, 11, 2^2 * 3, 13, 2 * 7]
sage: L[12]
13
sage: type(L[12])
<class 'sage.structure.factorization_integer.IntegerFactorization'>
sage: [factor(n) for n in range(1, 15) if is_odd(n)]
[1, 3, 5, 7, 3^2, 11, 13]
```

Para mais sobre como criar listas usando "list comprehensions", veja *[PyT]*.

Fatiamento de lista (list slicing) é um recurso fantástico. Se `L` é uma lista, então `L[m:n]` retorna uma sub-lista de `L` obtida começando do m-ésimo elemento e terminando no $(n-1)$-ésimo elemento, como ilustrado abaixo.

```
sage: L = [factor(n) for n in range(1, 20)]
sage: L[4:9]
[5, 2 * 3, 7, 2^3, 3^2]
sage: L[:4]
[1, 2, 3, 2^2]
sage: L[14:4]
[]
sage: L[14:]
[3 * 5, 2^4, 17, 2 * 3^2, 19]
```

Tuplas são semelhantes à listas, exceto que elas são imutáveis: uma vez criadas elas não podem ser alteradas.

```
sage: v = (1,2,3,4); v
(1, 2, 3, 4)
sage: type(v)
<... 'tuple'>
sage: v[1] = 5
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Sequências são um terceiro tipo de dados do Sage semelhante a listas. Diferentemente de listas e tuplas, Sequence não é um tipo de dados nativo do Python. Por definição, uma sequência é mutável, mas usando o método `set_immutable` da classe `Sequence` elas podem ser feitas imutáveis, como mostra o exemplo a seguir. Todos os elementos da sequência possuem um parente comum, chamado o universo da sequência.

```
sage: v = Sequence([1,2,3,4/5])
sage: v
[1, 2, 3, 4/5]
sage: type(v)
<class 'sage.structure.sequence.Sequence_generic'>
sage: type(v[1])
<type 'sage.rings.rational.Rational'>
sage: v.universe()
Rational Field
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v[0] = 3
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Sequências são derivadas de listas e podem ser usadas em qualquer lugar que listas são usadas.

```
sage: v = Sequence([1,2,3,4/5])
sage: isinstance(v, list)
True
sage: list(v)
[1, 2, 3, 4/5]
sage: type(list(v))
<... 'list'>
```

Como um outro exemplo, bases para espaços vetoriais são sequências imutáveis, pois é importante que elas não sejam modificadas.

```
sage: V = QQ^3; B = V.basis(); B
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: type(B)
<class 'sage.structure.sequence.Sequence_generic'>
sage: B[0] = B[1]
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: B.universe()
Vector space of dimension 3 over Rational Field
```

6.6 Dicionários

Um dicionário (também chamado as vezes de lista associativa ou "hash table") é um mapeamento de objetos em objetos arbitrários. (Exemplos de objetos que admitem uma lista associativa são strings e números; veja a documentação Python em https://docs.python.org/3/tutorial/index.html para detalhes).

```
sage: d = {1:5, 'sage':17, ZZ:GF(7)}
sage: type(d)
<... 'dict'>
sage: list(d.keys())
[1, 'sage', Integer Ring]
sage: d['sage']
17
sage: d[ZZ]
Finite Field of size 7
sage: d[1]
5
```

A terceira chave (key) ilustra como os índices de um dicionário podem ser complicados, por exemplo, um anel de inteiros.

Você pode transformar o dicionário acima em uma lista com os mesmos dados:

```
sage: list(d.items())
[(1, 5), ('sage', 17), (Integer Ring, Finite Field of size 7)]
```

É comum iterar sobre os pares em um dicionário:

```
sage: d = {2:4, 3:9, 4:16}
sage: [a*b for a, b in d.items()]
[8, 27, 64]
```

Um dicionário não possui ordem, como o exemplo acima mostra.

6.7 Conjuntos

O Python possui um tipo de conjuntos (set) nativo. O principal recurso que ele oferece é a rápida verificação se um objeto está ou não em um conjunto, juntamente com as operações comuns em conjuntos.

```
sage: X = set([1,19,'a']);   Y = set([1,1,1, 2/3])
sage: X   # random
{1, 19, 'a'}
sage: Y   # random
{2/3, 1}
sage: 'a' in X
True
sage: 'a' in Y
False
sage: X.intersection(Y)
{1}
```

O Sage também possui o seu próprio tipo de dados para conjuntos que é (em alguns casos) implementado usando o tipo nativo do Python, mas possuir algumas funcionalidades adicionais. Crie um conjunto em Sage usando `Set(...)`. Por exemplo,

```
sage: X = Set([1,19,'a']);   Y = Set([1,1,1, 2/3])
sage: X   # random
{'a', 1, 19}
sage: Y   # random
{1, 2/3}
sage: X.intersection(Y)
{1}
sage: print(latex(Y))
\left\{1, \frac{2}{3}\right\}
sage: Set(ZZ)
Set of elements of Integer Ring
```

6.8 Iteradores

Iteradores foram adicionados recentemente ao Python e são particularmente úteis em aplicações matemáticas. Aqui estão vários exemplos; veja *[PyT]* para mais detalhes. Vamos criar um iterador sobre o quadrados dos números inteiros até 10000000.

```
sage: v = (n^2 for n in xrange(10000000))  # py2
sage: v = (n^2 for n in range(10000000))  # py3
sage: next(v)
0
sage: next(v)
1
sage: next(v)
4
```

Criamos agora um iterador sobre os primos da forma $4p+1$ com p também primo, e observamos os primeiros valores.

```
sage: w = (4*p + 1 for p in Primes() if is_prime(4*p+1))
sage: w         # in the next line, 0xb0853d6c is a random 0x number
<generator object at 0xb0853d6c>
sage: next(w)
```

(continues on next page)

(continuação da página anterior)

```
13
sage: next(w)
29
sage: next(w)
53
```

Certos anéis, por exemplo, corpos finitos e os inteiros possuem iteradores associados a eles:

```
sage: [x for x in GF(7)]
[0, 1, 2, 3, 4, 5, 6]
sage: W = ((x,y) for x in ZZ for y in ZZ)
sage: next(W)
(0, 0)
sage: next(W)
(0, 1)
sage: next(W)
(0, -1)
```

6.9 Laços, Funções, Enunciados de Controle e Comparações

Nós já vimos alguns exemplos de alguns usos comuns de laços (loops) `for`. Em Python, um laço `for` possui uma estrutura tabulada, tal como

```
>>> for i in range(5):
...       print(i)
...
0
1
2
3
4
```

Note que os dois pontos no final do enunciado (não existe "do" ou "od" como no GAP ou Maple), e a identação antes dos comandos dentro do laço, isto é, `print(i)`. A tabulação é importante. No Sage, a tabulação é automaticamente adicionada quando você digita `enter` após ":", como ilustrado abaixo.

```
sage: for i in range(5):
....:     print(i)  # now hit enter twice
0
1
2
3
4
```

O símbolo = é usado para atribuição. O símbolo == é usado para verificar igualdade:

```
sage: for i in range(15):
....:     if gcd(i,15) == 1:
....:         print(i)
1
2
4
7
8
```

(continues on next page)

(continuação da página anterior)

```
11
13
14
```

Tenha em mente como a tabulação determina a estrutura de blocos para enunciados `if`, `for`, e `while`:

```
sage: def legendre(a,p):
....:     is_sqr_modp=-1
....:     for i in range(p):
....:         if a % p == i^2 % p:
....:             is_sqr_modp=1
....:     return is_sqr_modp

sage: legendre(2,7)
1
sage: legendre(3,7)
-1
```

Obviamente essa não é uma implementação eficiente do símbolo de Legendre! O objetivo é ilustrar vários aspectos da programação em Python/Sage. A função {kronecker}, que já vem com o Sage, calcula o símbolo de Legendre eficientemente usando uma biblioteca em C do PARI.

Finalmente, observamos que comparações, tais como ==, !=, <=, >=, >, <, entre números irão automaticamente converter ambos os números para o mesmo tipo, se possível:

```
sage: 2 < 3.1; 3.1 <= 1
True
False
sage: 2/3 < 3/2;   3/2 < 3/1
True
True
```

Use bool para desigualdades simbólicas:

```
sage: x < x + 1
x < x + 1
sage: bool(x < x + 1)
True
```

Quando se compara objetos de tipos diferentes no Sage, na maior parte dos casos o Sage tenta encontrar uma coação canônica para ambos os objetos em um parente comum (veja *Famílias, Conversão e Coação* para mais detalhes). Se isso for bem sucedido, a comparação é realizada entre os objetos que foram coagidos; se não for bem sucedido, os objetos são considerados diferentes. Para testar se duas variáveis fazem referência ao mesmo objeto use `is`. Como se vê no próximo exemplo, o int `1` do Python é único, mas o Inteiro `1` do Sage não é.

```
sage: 1 is 2/2
False
sage: int(1) is int(2)/int(2)   # py2
True
sage: 1 is 1
False
sage: 1 == 2/2
True
```

Nas duas linhas seguintes, a primeira igualdade é falsa (`False`) porque não existe um morfismo canônico $QQ \to \mathbf{F}_5$, logo não há uma forma de comparar o 1 em $\mathbf{F}_5$ com o $1 \in \mathbf{Q}$. Em contraste, existe um mapa canônico entre $\mathbf{Z} \to \mathbf{F}_5$, logo a segunda comparação é verdadeira (`True`)

```
sage: GF(5)(1) == QQ(1); QQ(1) == GF(5)(1)
False
False
sage: GF(5)(1) == ZZ(1); ZZ(1) == GF(5)(1)
True
True
sage: ZZ(1) == QQ(1)
True
```

ATENÇÃO: Comparação no Sage é mais restritiva do que no Magma, o qual declara $1 \in \mathbf{F}_5$ igual a $1 \in \mathbf{Q}$.

```
sage: magma('GF(5)!1 eq Rationals()!1')           # optional magma required
true
```

6.10 Otimização (Profiling)

Autor desta seção: Martin Albrecht (https://martinralbrecht.wordpress.com/)

> "Premature optimization is the root of all evil." - Donald Knuth

As vezes é útil procurar por gargalos em programas para entender quais partes gastam maior tempo computacional; isso pode dar uma boa ideia sobre quais partes otimizar. Python e portanto Sage fornecem várias opções de "profiling" (esse é o nome que se dá ao processo de otimização).

O mais simples de usar é o comando `prun` na linha de comando interativa. Ele retorna um sumário sobre o tempo computacional utilizado por cada função. Para analisar (a atualmente lenta! – na versão 1.0) multiplicação de matrizes sobre corpos finitos, por exemplo, faça o seguinte:

```
sage: k,a = GF(2**8, 'a').objgen()
sage: A = Matrix(k,10,10,[k.random_element() for _ in range(10*10)])
```

```
sage: %prun B = A*A
         32893 function calls in 1.100 CPU seconds

Ordered by: internal time

ncalls tottime percall cumtime percall filename:lineno(function)
 12127   0.160   0.000   0.160   0.000 :0(isinstance)
  2000   0.150   0.000   0.280   0.000 matrix.py:2235(__getitem__)
  1000   0.120   0.000   0.370   0.000 finite_field_element.py:392(__mul__)
  1903   0.120   0.000   0.200   0.000 finite_field_element.py:47(__init__)
  1900   0.090   0.000   0.220   0.000 finite_field_element.py:376(__compat)
   900   0.080   0.000   0.260   0.000 finite_field_element.py:380(__add__)
     1   0.070   0.070   1.100   1.100 matrix.py:864(__mul__)
  2105   0.070   0.000   0.070   0.000 matrix.py:282(ncols)
  ...
```

Aqui `ncalls` é o números de chamadas, `tottime` é o tempo total gasto por uma determinada função (excluíndo o tempo gasto em chamadas de subfunções), `percall` é o quociente de `tottime` dividido por `ncalls`. `cumtime` é o tempo total gasto nessa e em todas as subfunções (isto é, desde o início até o término da execução da função), `percall` é o quociente de `cumtime` dividido pelas chamadas primitivas, e `filename:lineno(function)` fornece os dados respectivos para cada função. A regra prática aqui é: Quanto mais no topo uma função aparece nessa lista, mais custo computacional ela acarreta. Logo é mais interessante para ser optimizada.

Como usual, `prun?` fornece detalhes sobre como usar o "profiler" e como entender a saída de dados.

A saída de dados pode ser escrita em um objeto para permitir uma análise mais detalhada:

```
sage: %prun -r A*A
sage: stats = _
sage: stats?
```

Note: digitando `stats = prun -r A\*A` obtém-se um erro de sintaxe porque prun é um comando do IPython, não uma função comum.

Para uma representação gráfica dos dados do "profiling", você pode usar o "hotspot profiler", um pequeno script chamado `hotshot2cachetree` e o programa `kcachegrind` (apenas no Unix). O mesmo exemplo agora com o "hotspot profiler":

```
sage: k,a = GF(2**8, 'a').objgen()
sage: A = Matrix(k,10,10,[k.random_element() for _ in range(10*10)])
sage: import hotshot
sage: filename = "pythongrind.prof"
sage: prof = hotshot.Profile(filename, lineevents=1)
```

```
sage: prof.run("A*A")
<hotshot.Profile instance at 0x414c11ec>
sage: prof.close()
```

Isso resulta em um arquivo `pythongrind.prof` no diretório de trabalho atual. Ele pode ser convertido para o formato cachegrind para visualização.

Em uma linha de comando do sistema, digite

```
hotshot2calltree -o cachegrind.out.42 pythongrind.prof
```

O arquivo de saída `cachegrind.out.42` pode ser examinado com `kcachegrind`. Note que a convenção de nomes `cachegrind.out.XX` precisa ser obedecida.

CAPÍTULO 7

Usando o SageTeX

O pacote SageTeX permite que você insira resultados de cálculos feitos com o Sage em um documento LaTeX. Esse pacote já vem com o Sage. Para usá-lo, você precisa "instalá-lo" em seu sistema LaTeX local; aqui instalar significa copiar um simples arquivo. Veja *Instalação* neste tutorial e a seção "Make SageTeX known to TeX" do Guia de instalação do Sage (em inglês).

Aqui vai um breve exemplo de como usar o SageTeX. A documentação completa pode ser encontrada em `SAGE_ROOT/local/share/texmf/tex/latex/sagetex`, onde `SAGE_ROOT` é o diretório onde se encontra a sua instalação. Esse diretório contém a documentação, um arquivo de exemplo, e alguns scripts em Python possivelmente úteis.

Para ver como o SageTeX funciona, siga as instruções para instalar o SageTeX (em *Instalação*) e copie o seguinte texto em um arquivo chamado `st_example.tex`, por exemplo.

Aviso: O texto abaixo vai apresentar diversos erros sobre "unknown control sequences" se você está visualizando isto na ajuda "live". Use a versão estática para ver o texto corretamente.

```
\documentclass{article}
\usepackage{sagetex}

\begin{document}

Using Sage\TeX, one can use Sage to compute things and put them into
your \LaTeX{} document. For example, there are
$\sage{number_of_partitions(1269)}$ integer partitions of $1269$.
You don't need to compute the number yourself, or even cut and paste
it from somewhere.

Here's some Sage code:

\begin{sageblock}
    f(x) = exp(x) * sin(2*x)
\end{sageblock}
```

(continues on next page)

(continuação da página anterior)

```
The second derivative of $f$ is

\[
  \frac{\mathrm{d}^{2}}{\mathrm{d}x^{2}} \sage{f(x)} =
  \sage{diff(f, x, 2)(x)}.
\]

Here's a plot of $f$ from $-1$ to $1$:

\sageplot{plot(f, -1, 1)}

\end{document}
```

Execute o LaTeX em `st_example.tex` da forma usual. Note que o LaTeX vai reclamar sobre algumas coisas, entre elas:

```
Package sagetex Warning: Graphics file
sage-plots-for-st_example.tex/plot-0.eps on page 1 does not exist. Plot
command is on input line 25.

Package sagetex Warning: There were undefined Sage formulas and/or
plots. Run Sage on st_example.sage, and then run LaTeX on
st_example.tex again.
```

Observe que, além dos arquivos usuais produzidos pelo LaTeX, existe um arquivo chamado `st_example.sage`. Esse é um script em Sage produzido quando você executa o LaTeX em `st_example.tex`. A mensagem de alerta pede para você executar o LaTeX em `st_example.sage`, então siga essa sugestão e faça isso. Você vai receber uma mensagem para executar o LaTeX em `st_example.tex` novamente, mas antes que você faça isso, observe que um novo arquivo foi criado: `st_example.sout`. Esse arquivo contém os resultados dos cálculos feitos pelo Sage, em um formato que o LaTeX pode usar para inserir em seu texto. Um novo diretório contendo um arquivo EPS do seu gráfico também foi criado. Execute o LaTeX novamente e você vai ver que tudo que foi calculado, incluindo os gráficos, foi incluído em seu documento.

As funções (macros em inglês) utilizadas acima devem ser fáceis de entender. Um ambiente `sageblock` insere código "verbatim" (exatamente como é digitado) e também executa o código quando você executa o Sage. Quando você insere `\sage{foo}`, é incluído em seu documento o resultado que você obteria executando `latex(foo)` no Sage. Comandos para fazer gráficos são um pouco mais complicados, mas em sua forma mais simples, `\sageplot{foo}` insere a imagem que você obtêm usando `foo.save('filename.eps')`.

Em geral, a rotina é a seguinte:

- execute o LaTeX no seu arquivo .tex;
- execute o Sage no arquivo .sage que foi gerado;
- execute o LaTeX novamente.

Você pode omitir a execução do Sage desde que você não tenha alterado os comandos em Sage em seu documento.

Há muito mais sobre o SageTeX, e como tanto o Sage como o LaTeX são ferramentas complexas e poderosas, é uma boa idéia ler a documentação para o SageTeX que se encontra em `SAGE_ROOT/local/share/texmf/tex/latex/sagetex`.

CAPÍTULO 8

Posfacio

8.1 Por quê o Python?

8.1.1 Vantagens do Python

A primeira linguagem de implementação do Sage é o Python (veja *[Py]*), embora rotinas que precisam ser muito rápidas são implementadas em uma linguagem compilada. O Python possui várias vantagens:

- **Salvar objetos** é bem suportado em Python. Existe suporte extenso em Python para salvar (na grande maioria dos casos) objetos arbitrários em arquivos em disco ou em uma base de dados.
- Suporte excelente para **documentação** de funções e pacotes no código fonte, incluindo extração automática de documentação e teste automático de todos os exemplos. Os exemplos são automaticamente testados regularmente para garantir que funcionam como indicado.
- **Gerenciamento de memória:** O Python agora possui um sistema de gerenciamento de memória e "garbage collector" muito bem pensados e robustos que lidam corretamente com referências circulares, e permitem variáveis locais em arquivos.
- O Python possui **diversos pacotes** disponíveis que podem ser de grande interesse para os usuários do Sage: análise numérica e álgebra linear, visualização 2D e 3D, comunicação via rede (para computação distribuída e servidores, por exemplo, via twisted), suporte a base de dados, etc.
- **Portabilidade:** O Python é fácil de compilar a partir do código fonte em poucos minutos na maioria das arquiteturas.
- **Manuseamento de exceções:** O Python possui um sofisticado e bem pensado sistema de manuseamento de exceções, através do qual programas podem facilmente se recuperar mesmo se ocorrerem erros no código que está sendo executado.
- **Debugador:** O Python inclui um debugador, de modo que quando alguma rotina falha por algum motivo, o usuário pode acessar extensiva informação sobre a pilha de cálculos e inspecionar o estado de todas as variáveis relevantes.
- **Profiler:** Existe um profiler para o Python, o qual executa programas e cria um relatório detalhando quantas vezes e por quando tempo cada função foi executada.

- **Uma Linguagem:** Em vez de escrever uma **nova linguagem** para matemática como foi feito para o Magma, Maple, Mathematica, Matlab, GP/PARI, GAP, Macaulay 2, Simath, etc., nós usamos a linguagem Python, que é uma linguagem de programação popular que está sendo desenvolvida e otimizada ativamente por centenas de engenheiros de software qualificados. O Python é uma grande história de sucesso em desenvolvimento com código aberto com um processo de desenvolvimento maduro (veja *[PyDev]*).

8.1.2 O Pré-Processador: Diferenças entre o Sage e o Python

Alguns aspectos matemáticos do Python podem ser confusos, logo o Sage se comporta diferentemente do Python em diversas situações.

- **Notação para exponenciação:** `**` versus `^`. Em Python, `^` significa "xor", não exponenciação, logo em Python temos

```
>>> 2^8
10
>>> 3^2
1
>>> 3**2
9
```

  Esse uso de `^` pode parecer estranho, e é ineficiente para pesquisa em matemática pura, pois a função "ou exclusivo" é raramente usada. Por conveniência, o Sage pre-processa todos as linhas de comandos antes de passá-las para o Python, substituindo ocorrências de `^` que não estão em strings por `**`:

```
sage: 2^8
256
sage: 3^2
9
sage: "3^2"
'3^2'
```

- **Divisão por inteiros:** A expressão em Python `2/3` não se comporta da forma que um matemático esperaria. Em Python 2, se `m` e `n` são inteiros (int), então `m/n` também é um inteiro (int), a saber, o quociente de `m` dividido por `n`. Portanto `2/3=0`. Tem havido discussões na comunidade do Python para modificar o Python de modo que `2/3` retorne um número de precisão flutuante (float) `0.6666...`, e `2//3` retorne `0`.

  Nós lidamos com isso no interpretador Sage, encapsulando inteiros literais em `Integer()` e fazendo a divisão um construtor para números racionais. Por exemplo:

```
sage: 2/3
2/3
sage: (2/3).parent()
Rational Field
sage: 2//3
0
sage: int(2)/int(3)    # py2
0
```

- **Inteiros longos:** O Python possui suporte nativo para inteiros com precisão arbitrária, além de int's do C. Esses são significantemente mais lentos do que os fornecidos pela biblioteca GMP, e têm a propriedade que eles são impressos com o sufixo `L` para distingui-los de int's (e isso não será modificado no futuro próximo). O Sage implementa inteiros com precisão arbitrária usando a biblioteca C do GMP, e esses são impressos sem o sufixo `L`.

Em vez de modificar o interpretador Python (como algumas pessoas fizeram para projetos internos), nós usamos a linguagem Python exatamente com ela é, e escrevemos um pré-processador para o IPython de modo que o compor-

tamento da linha de comando seja o que um matemático espera. Isso significa que qualquer programa existente em Python pode ser usado no Sage. Todavia, deve-se obedecer as regras padrão do Python para escrever programas que serão importados no Sage.

(Para instalar uma biblioteca do Python, por exemplo uma que você tenha encontrado na internet, siga as instruções, mas execute `sage -python` em vez de `python`. Frequentemente isso significa digitar `sage -python setup.py install`.)

8.2 Eu gostaria de contribuir de alguma forma. Como eu posso?

Se você quiser contribuir para o Sage, a sua ajuda será muito bem vinda! Ela pode variar desde substancial quantidade de código, até contribuições com respeito à documentação ou notificação de defeitos (bugs).

Explore a página na web do Sage para informações para desenvolvedores; entre outras coisas, você pode encontrar uma lista longa de projetos relacionados ao Sage ordenados por prioridade e categoria. O Guia para desenvolvedores do Sage (em inglês) também possui informações úteis, e você pode também visitar o grupo de discussões `sage-devel` no Google Groups.

8.3 Como eu faço referência ao Sage?

Se você escrever um artigo usando o Sage, por favor faça referência aos cálculos feitos com o Sage incluindo

```
[Sage] William A. Stein et al., Sage Mathematics Software (Version 4.3).
       The Sage Development Team, 2009, http://www.sagemath.org.
```

na sua bibliografia (substituindo 4.3 pela versão do Sage que você está usando). Além disso, procure observar quais componentes do Sage você está usando em seus cálculos, por exemplo, PARI, Singular, GAP, Maxima, e também site esses sistemas. Se você está em dúvida sobre qual software está sendo usado em seus cálculos, fique à vontade para perguntar no grupo `sage-devel` do Google Groups. Veja *Polinômios em Uma Variável* para mais discussões sobre esse aspecto.

Se por acaso você leu este tutorial do começo ao fim em uma só vez, e faz idéia de quanto tempo você levou, por favor nos informe no grupo `sage-devel` do Google Groups.

Divirta-se com o Sage!

CAPÍTULO 9

Apêndice

9.1 Precedência de operações aritméticas binárias

Quanto é `3^2*4 + 2%5`? A resposta (38) é determinada pela "tabela de precedência" abaixo. A tabela abaixo é baseada na tabela em § 5.14 do *Python Language Reference Manual* by G. Rossum and F. Drake. As operações estão listadas aqui em ordem crescente de precedência.

Operadores	Descrição
or	"ou" booleano
and	"e" booleano
not	"não" booleano
in, not in	pertence
is, is not	teste de identidade
>, <=, >, >=, ==, !=, <>	comparação
+, -	adição, subtração
*, /, %	multiplicação, divisão, resto
**, ^	exponenciação

Portanto, para calcular `3^2*4 + 2%5`, O Sage inclui parenteses de precedência da seguinte forma: `((3^2)*4) + (2%5)`. Logo, primeiro calcula `3^2`, que é `9`, então calcula `(3^2)*4` e `2%5`, e finalmente soma os dois.

CAPÍTULO 10

Bibliografia

CAPÍTULO 11

Índices e tabelas

- genindex
- modindex
- search

Referências Bibliográficas

[Cyt] Cython, http://www.cython.org/

[Dive] Dive into Python, disponível gratuitamente na internet em http://diveintopython.net/

[GAP] The GAP Group, GAP - Groups, Algorithms, and Programming, Version 4.4; 2005, http://www.gap-system.org/

[GAPkg] GAP Packages, http://www.gap-system.org/Packages/packages.html

[GP] PARI/GP http://pari.math.u-bordeaux.fr/

[Ip] The IPython shell http://ipython.scipy.org/

[Jmol] Jmol: an open-source Java viewer for chemical structures in 3D http://www.jmol.org/

[Mag] Magma http://magma.maths.usyd.edu.au/magma/

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

[NagleEtAl2004] Nagle, Saff, and Snider. *Fundamentals of Differential Equations*. 6th edition, Addison-Wesley, 2004.

[Py] The Python language http://www.python.org/ Reference Manual http://docs.python.org/

[PyDev] Python Developer's Guide, https://docs.python.org/devguide/

[Pyr] Pyrex, http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/

[PyT] The Python Tutorial http://www.python.org/

[SA] Sage web site http://www.sagemath.org/

[Si] G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 3.0. A Computer Algebra System for Polynomial Computations. Center for Computer Algebra, University of Kaiserslautern (2005). http://www.singular.uni-kl.de/

[SJ] William Stein, David Joyner, Sage: System for Algebra and Geometry Experimentation, Comm. Computer Algebra {39}(2005)61-64.

Índice