Sage 教程

Release 4.3

Sage 开发组

January 08, 2010

CONTENTS

Sage 是免费、开源的数学软件, 可用于代数、几何、数论、密码学、数值计算以及其他相关领域的教学和科研。Sage 的开发模型和所用到的技术强调开放、社区、合作以及协同工作: 我们是造汽车, 而不是重新发明轮子。Sage 的总体目标是成为一个实用的、免费的、开源的数学软件以代替 Maple, Mathematica, Magma 和 MATLAB。

本教程是在几个小时内熟悉 Sage 最好的办法。你可以阅读 HTML 或者 PDF 版本, 或者在 Sage notebook 中 (点击 `Help` , 再点击 `Tutorial`) 边阅读边使用。

本文档基于 Creative Commons Attribution-Share Alike 3.0 License 协议发布。

CHAPTER

ONE

简介

阅读本教程的全部内容最多只需要 3、4 个小时。你可以阅读 HTML 或 PDF 版本，或者在 Sage notebook 中点击 `Help` ，再点击 `Tutorial` ，边阅读边使用 Sage。

虽然 Sage 主要是用 Python 实现的，但是不懂 Python 也可以阅读本教程。如果你想要学习一下 Python (一种非常有趣的语言) ，网上有很多关于 Python 的优秀资源，比如 [PyT] 和 [Dive] 。如果你只是想快速的尝试一下 Sage，阅读本教程就对了。比如：

```
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]
sage: E.rank()
1

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

1.1 安装

如果你没有安装 Sage，只是想试几个命令，可以使用在线的 Sage notebook：http://www.sagenb.org 。

要在自己的电脑上安装 Sage，请参考 Sage 主页 [Sage] 上的 Sage 安装指南。这里我们只强调两点：

1. Sage 的安装包是“内置电池”的。也就是说，虽然 Sage 用到了 Python，IPython，PARI，GAP，Singular，Maxima，NTL，GMP 等等一些软件，但是你不需要单独安装这些软件，因为它们已经包含在 Sage 的发行版里了。然而，要使用 Sage 的一些特定的功能，比如 Macaulay 或者 KASH，你必须安装相关的 Sage 可选包或者已经单独安装了这些软件。Macaulay 和 KASH 都是 Sage 的扩展包（输入 `sage -optional` 可以得到可选扩展包列表，或者在 Sage 网站上浏览“下载”页）。
2. 安装编译好的二进制版本（可在 Sage 网站找到）可能比安装源码版更容易、更快。只需要解压缩之后运行 `sage` 即可。

1.2 使用 Sage 的方法

使用 Sage 的方法有好几种。

- **Notebook 图形界面：**参见参考手册中关于 Notebook 的章节，以及下面的 *Notebook 界面* ；
- **交互命令行：**参见 *交互命令行* ；
- **程序：**在 Sage 中编写解释型或者编译型的程序（参见 *读取和附加 Sage 文件* 和 *创建编译代码* ）；
- **脚本：**在独立的 Python 脚本中调用 Sage 库文件（参见 *独立的 Python/Sage 脚本* ）。

1.3 Sage 的长期目标

- **实用：** Sage 的预期用户是学数学的学生 (从高中生到研究生)、教师以及研究人员。我们的目标是在代数、几何、数论、微积分、数值计算等领域提供可用于探索和尝试的软件。Sage 使得进行与数学对象有关的交互实验变得容易。
- **高效：** 越快越好。Sage 使用高度优化的成熟软件, 如 GMP, PARI, GAP 和 NTL。这样, Sage 的某些运算非常快。
- **免费、开源：** 源代码必须可以自由的获取, 并且有较好的可读性, 这样用户才能真正了解系统是如何运行的, 并且更容易进行扩展。就像数学家要深入理解一个定理的话, 就要仔细地阅读定理的证明, 最起码要浏览一下。搞计算的人应该可以通过阅读源码来了解计算是如何进行的。如果你在论文中使用 Sage 进行计算, 你可以确保读者能够免费得到 Sage 及其源码。并且你可以打包或者重新发布你所使用的 Sage 版本。
- **易于编译：** Linux, OS X 和 Windows 的用户应该很容易使用源代码编译 Sage。这为用户修改系统提供了便利。
- **协作：** 为其他计算机代数系统提供健壮的接口, 包括 PARI, GAP, Singular, Maxima, KASH, Magma, Maple 和 Mathematica。Sage 希望统一并扩展现有的数学软件。
- **文档完善：** 教程, 编程指南, 参考手册和基本指南要包含大量的例子, 以及对数学背景的讨论。
- **可扩展：** 可以定义新的数据类型或者从内置的类型中继承, 可以使用其他语言编写的代码。
- **用户友好：** 给定对象所提供的功能应该是清晰易懂的, 文档和源码应该易于查看。用户支持要达到比较高的水平。

CHAPTER

TWO

导览

这一节是关于 Sage 中所含内容的一个导览。更多的实例请参考“Sage 的构成”，那份文档想回答一个常见问题：“如何构造……”。还可以参考“Sage 参考手册”，那份文档中有更多的例子。再次提醒，你可以用交互的方式学习这份文档，在 Sage notebook 中点击 `help` 链接。

如果你正在 Sage notebook 中阅读这份教程，可以按 `shift+enter` 来执行任一输入单元 (input cell) 中的代码。你还可以在按 shift+enter 之前修改代码。在苹果机上，你可能要按 `shift+return` 而不是 `shift+enter` 。

2.1 赋值，等式和算术运算

除了少数例外，Sage 使用了 Python 语言，因此大多数关于 Python 的入门书籍将帮助你学习 Sage。

Sage 使用 = 进行赋值，使用 ==, <=, >=, < 和 > 进行比较：

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

Sage 提供了基本的数学运算：

```
sage: 2**3    #  ** 的意思是指数
8
sage: 2^3     #  ^ 和 ** 一个意思（这点不象 Python）
```

```
8
sage: 10 % 3  #  对于整数参数, % 的意思是取模, 即求余数
1
sage: 10/4
5/2
sage: 10//4   #  对于整数参数, // 返回整数商
2
sage: 4 * (10 // 4) + 10 % 4 == 10
True
sage: 3^2*4 + 2%5
38
```

象 `3^2*4 + 2%5` 这样的表达式的计算结果取决于运算的顺序。计算顺序由“运算符优先级表”指定, 参见 二元数学运算符优先级 。

Sage 还提供了许多常见的数学函数, 这里是几个例子:

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

象最后一个例子那样, 有些数学表达式返回“精确”的值, 而不是近似的数值结果。要得到一个近似的数值解, 使用函数 `n` 或者方法 `n` （两者的全名都是 `numerical_approx`, 并且函数 `N` 和 `n` 是一样的）。它们都有可选参数 `prec` 和 `digits` , 前者指定结果的二进制位数, 即 bit 数, 后者指定结果的十进制位数。默认精度是 53 bit。

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

Python 是动态类型的, 因此每一个赋给变量的值都有一个类型, 但是在给定作用域内, 一个给定的变量可以接受任何 Python 类型的值。

```
sage: a = 5   # a 是整数
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a = 5/3  # 现在 a 是有理数
sage: type(a)
<type 'sage.rings.rational.Rational'>
sage: a = 'hello'  # 现在 a 是字符串
sage: type(a)
<type 'str'>
```

C 语言就很不一样了，它是静态类型的。一个被声明为整数的变量，在它的作用域内只能接受整数值。

Python 中一个潜在的容易混淆的地方是，以 0 开头的整数是八进制数，也就是以 8 为基的数。

```
sage: 011
9
sage: 8 + 1
9
sage: n = 011
sage: n.str(8)   # 将 n 以 8 进制字符串形式输出
'11'
```

这与 C 语言的规定是一致的。

2.2 获取帮助

Sage 拥有强大的内置文档，只需要输入函数或者常数的名字，再加个问号即可：

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

    The tangent function

    EXAMPLES:
        sage:  tan(pi)
        0
        sage:  tan(3.1415)
        -0.0000926535900581913
        sage:  tan(3.1415/4)
        0.999953674278156
        sage:  tan(pi/4)
        1
```

```
        sage:  tan(1/2)
        tan(1/2)
        sage:  RR(tan(1/2))
        0.546302489843790
sage:  log2?
Type:        <class 'sage.functions.constants.Log2'>
Definition:  log2( [noargspec] )
Docstring:

    The natural logarithm of the real number 2.

    EXAMPLES:
        sage:  log2
        log2
        sage:  float(log2)
        0.69314718055994529
        sage:  RR(log2)
        0.693147180559945
        sage:  R = RealField(200); R
        Real Field with 200 bits of precision
        sage:  R(log2)
        0.69314718055994530941723212145817656807550013436025525412068
        sage:  l = (1-log2)/(1+log2); l
        (1 - log(2))/(log(2) + 1)
        sage:  R(l)
        0.18123221829928249948761381864650311423330609774776013488056
        sage:  maxima(log2)
        log(2)
        sage:  maxima(log2).float()
        .6931471805599453
        sage:  gp(log2)
        0.69314718055994530941723212145817656807   # 64-bit
        0.69314718055994530941723212150             # 32-bit
sage:  sudoku?
File:        sage/local/lib/python2.5/site-packages/sage/games/sudoku.py
Type:        <type 'function'>
Definition:  sudoku(A)
Docstring:

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

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

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

Sage 还提供了“Tab 补全”功能: 输入函数名的前面几个字母, 然后按 tab 键。比如, 如果你输入 `ta` 再按 `TAB`, Sage 就会列出 `tachyon, tan, tanh, taylor`. 这是一个查找 Sage 函数或者其他结构的好方法。

2.3 函数, 缩进和计数

在 Sage 中定义一个新的函数, 需要使用 `def` 命令, 并且在变量列表后跟一个冒号。比如:

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

注: 根据你所阅读的本教程的版本的不同, 在这个例子中, 你可能会看到第二行有三个点”...” 。不要输入它们, 它们只是强调一下代码是缩进的。不管是什么情况, 在程序块的最后, 按 [Return/Enter] 插入一个空行以结束函数的定义。

你没有指定输入参数的类型。你可以指定多个输入, 每个参数都可以带一个可选的默认值。比如下面的函数中, 如果不指定 `divisor` 的话, 默认取 `divisor=2`.

```
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
```

在调用函数时，你还可以明确的指定一个或多个参数的值。如果你明确指定参数的值，参数可以以任何顺序出现。

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

与其他很多语言不同，Python 中的程序块不用花括号或者 begin，end 来标记，而是用精确的缩进来标记。比如下面的代码有一个语法错误，`return` 语句与它上面的语句缩进的不完全一致。

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

修正缩进格数之后，函数就对了：

```
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]
```

多数情况下，一行结束后会开始一个新行，这时行尾不需要分号。但是如果要将多个语句放在同一行，就要用分号隔开：

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

如果你要将一行代码分开放在多行, 要在行尾使用反斜杠:

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

在 Sage 中, 通过遍历一个范围内的整数进行计数。比如下面代码中的第一行相当于 C++ 或者 Java 中的 `for(i=0; i<3; i++)`:

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

下面的第一行相当于 `for(i=2;i<5;i++)`.

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

第三个参数控制步长, 下面的第一行相当于 `for(i=1;i<6;i+=2)`.

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

可能你经常需要将 Sage 中的计算结果以漂亮的表格形式输出, 一个简单的方法是使用格式化字符串。下面, 我们计算数的平方和立方, 并建立一个有三列的表格, 每一列都是 6 个字符宽。

```
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
```

Sage 中最最基本的数据结构是 list, 跟字面意思一样, list 就是任意对象的列表。比如我们刚才用到的 `range` 命令就产生一个 list:

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

下面是更复杂的 list:

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

象其他很多语言一样, list 的下标以 0 开始计数。

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

使用 `len(v)` 得到 `v` 的长度, 使用 `v.append(obj)` 向 `v` 的末尾添加新的对象, 使用 `del v[i]` 删除 `v` 的第 i 个元素:

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

另一个重要的数据结构是 dictionary (或 associative array) 。用法和 list 类似, 但它几乎可以使用所有的对象进行索引 (指标必须是固定的) :

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

你可以使用“类”定义新的数据结构。将数学对象用类进行封装是一个强大的技术, 可以帮你简化和组织 Sage 程序。下面我们定义一个类来表示不超过 n 的正偶数列表, 它由内置类型 `list` 继承而来。

```
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."
```

在建立对象时，调用 __init__ 方法进行初始化； __repr__ 方法打印对象。我们在 __init__ 方法的第二行调用 list 的 constructor 方法。我们可以象下面一样建立 Evens 类的一个对象： （译注：原文中使用的变量名为 e，考虑到 e 是内置的常数，因此换成了 ee ）

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

注意 ee 使用我们定义的 __repr__ 方法进行输出。要使用 list 的函数才能查看隐含的数据列表：

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

我们还可以访问 n 属性或者将 ee 当做 list。

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

2.4 基本的代数和微积分

Sage 可以执行许多基本的代数和微积分运算：如方程求解，微分，积分和 Laplace 变换。更多例子参见“Sage 的构成”。

2.4.1 解方程

精确求解方程

solve 函数用于解方程。要使用它，先要指定变量，然后将方程（或方程组）以及要求解的变量作为参数传给 solve.

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

可以求解单变量的方程：

```
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)]
```

也可以求解多变量的方程 (组) :

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

下面是一个由 Jason Grout 提供的 Sage 求解非线性方程组的例子。我们先求方程组的符号解。

```
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(2)*sqrt(5) - 2/3],
 [p == 1, q == 8, x == 4/3*sqrt(10) - 2/3, y == -1/6*sqrt(2)*sqrt(5) - 2/3]]
```

要求解的近似值, 可以这样:

```
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]]
```

(函数 `n` 输出数值近似值, 参数是以 bit 为单位的结果精度)

求方程的数值解

很多时候, `solve` 找不到给定方程或方程组的精确解。如果找不到, 你可以用 `find_root` 去找一个数值解。比如对于下面的方程, solve 不会返回任何有用的信息:

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

但是我们可以用 `find_root` 在区间 $0 < \phi < \pi/2$ 上寻找上述方程的解。

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

2.4.2 微分，积分等

Sage 知道如何求很多函数的微分和积分。比如求 $\sin(u)$ 对 u 的微分这样做：

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

计算 $\sin(x^2)$ 的 4 阶微分：

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

分别计算 $x^2 + 17y^2$ 对 x 和 y 的偏微分：

```
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
```

再来看积分，定积分、不定积分都可以计算。计算 $\int x\sin(x^2)\,dx$ 和 $\int_0^1 \frac{x}{x^2+1}\,dx$

```
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)
```

计算 $\frac{1}{x^2-1}$ 的部分分式分解：

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

2.4.3 解微分方程组

可以用 Sage 求解常微分方程组。求解方程 $x' + x - 1 = 0$:

```
sage: t = var('t')    # 定义变量 t
sage: x = function('x',t)   # 定义 x 是变量 t 的函数
sage: DE = diff(x, t) + x - 1
sage: desolve(DE, [x,t])
(c + e^t)*e^(-t)
```

这里用到了 Sage 调用 Maxima [Max] 的接口，所以它的输出看上去与其他 Sage 的输出略有不同。这里，上述微分方程的通解是：$x(t) = e^{-t}(e^t + c)$.

你也可以计算 Laplace 变换。下面计算 $t^2e^t - \sin(t)$ 的 Laplace 变换：

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

这儿有一个更复杂的例子。两个弹簧连在左边的墙上，

```
|------\/\/\/\/\---|mass1|----\/\/\/\/\/----|mass2|
         spring1                 spring2
```

物体偏离平衡态的位移可以描述为一个 2 阶常微分方程：

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

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

这里 m_i 是物体 i 的质量，x_i 是物体 i 偏离平衡态的位移，k_i 是弹簧 i 的弹性系数。

例：在下面的条件下，使用 Sage 求解上面的问题 $m_1 = 2$, $m_2 = 1$, $k_1 = 4$, $k_2 = 2$, $x_1(0) = 3$, $x_1'(0) = 0$, $x_2(0) = 3$, $x_2'(0) = 0$.

解：对第一个方程做 Laplace 变换（记 $x = x_1$, $y = x_2$ ）：

```
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)
```

结果很难读，意思其实是：

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

(这里对函数 $x(t)$ 的 Laplace 变换记为 $X(t)$)。对第二个方程做 Laplace 变换：

```
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
```

即

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

代入 $x(0)$, $x'(0)$, $y(0)$, 和 $y'(0)$ 的初始条件，并求解求出的两个方程：

```
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)]]
```

现在做逆 Laplace 变换得到结果:

```
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)
```

所以, 原方程组的解是:

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

可以把结果画出来:

```
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)
```

每一个分支都可以画出来:

```
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)
```

(更多关于做图的内容, 参见 绘图.)

参考文献: Nagle, Saff, Snider, Fundamentals of Differential Equations, 6th ed, Addison-Wesley, 2004. (见 § 5.5).

2.4.4 解微分方程组的 Euler 方法

下面的例子中，我们展示求解 1 阶，2 阶常微分方程组的 Euler 方法。我们先来回顾一下 1 阶方程的基本知识。给定如下形式的初值问题：

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

我们要找方程在 $x = b$ 处的近似解，且 $b > a$.

根据微分的定义

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

这里 $h > 0$ 是给定的，且较小的量。与微分方程一起得到 $f(x, y(x)) \approx \frac{y(x+h)-y(x)}{h}$. 现在求 $y(x+h)$:

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

如果将 $hf(x, y(x))$ 称为“校正项”（没有更好的名字），称 $y(x)$ 为 y 的旧值，$y(x+h)$ 为 y 的新值，那么该近似公式可以改写为：

$$y_{new} \approx y_{old} + h * f(x, y_{old}).$$

如果将由 a 到 b 的区间 n 等分，则 $h = \frac{b-a}{n}$，我们可以用一个表记录该方法得到的信息。

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

我们的目标是把表中的空格都填上，每次一行，直到到达 ??? 这一项，也就是 Euler 方法求得的 $y(b)$ 的近似值。

类似的，可以求解常微分方程组。

例：用 4 步 Euler 方法求 $z(t)$ 在 $t = 1$ 处的近似值，这里 $z'' + tz' + z = 0$, $z(0) = 1$, $z'(0) = 0$.

我们必须将 2 阶常微分方程化为两个 1 阶微分方法（令 $x = z$, $y = z'$ ）并再应用 Euler 方法：

```
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
```

即, $z(1) \approx 0.65$.

我们可以把点 (x, y) 画出来, 得到曲线的近似图像。函数 `eulers_method_2x2_plot` 可以做到这一点。为了应用该函数, 要先定义函数 f 和 g 来接受含三个坐标的参数: (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)
```

这里, `P` 保存了两个图像, `P[0]` 是 x 关于 t 的图像, `P[1]` 是 y 关于 t 的图像。我们把它们都画出来:

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

(更多关于做图的内容, 参见 绘图.)

2.4.5 特殊函数

一些正交多项式和特殊函数是使用 PARI [GAP] 和 Maxima [Max] 实现的。在 Sage 参考手册的相关章节 (“正交多项式”和“特殊函数”) 中有详细信息。

```
sage: x = polygen(QQ, 'x')
sage: chebyshev_U(2,x)
4*x^2 - 1
sage: bessel_I(1,1,"pari",250)
0.56515910399248502720769602760986330732889962162109200948029448947925564096
sage: bessel_I(1,1)
0.565159103992485
sage: bessel_I(2,1.1,"maxima")  # last few digits are random
0.16708949925104899
```

这里 Sage 直接求得数值解, 如果想求符号解, 请象下面这样直接使用 Maxima 接口:

```
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 绘图

Sage 可以绘制二维或三维图像。

2.5.1 绘制二维图像

二维图像中, Sage 可以画圆、直线和多边形, 在直角坐标系或极坐标系中做函数图像, 等高线图, 向量场图。下面是一些例子。更多关于绘图的例子请参考: *解微分方程组* 和 *Maxima*, 以及“Sage 的构成”。

下面的命令画一个以原点为中心, 半径为 1 的黄色的圆:

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

实心圆:

```
sage: circle((0,0), 1, rgbcolor=(1,1,0), fill=True)
```

还可以把圆赋给一个变量, 这样并不会将其画出来:

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

要画出来的话, 需要使用 `c.show()` 或 `show(c)`:

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

或者, 可以执行 `c.save('filename.png')` 把图像保存到文件中。

因为坐标轴的尺度不同, 现在这个“圆”看上去像椭圆。可以这样解决:

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

命令 `show(c, aspect_ratio=1)` 的效果是一样的, 或者你可以在保存图像的时候使用 `c.save('filename.png', aspect_ratio=1)`.

绘制基本函数的图像是非常容易的:

```
sage: plot(cos, (-5,5))
```

一旦指定了变量名, 就可以绘制参数方程的图像:

```
sage: x = var('x')
sage: parametric_plot((cos(x),sin(x)^3),(x,0,2*pi),rgbcolor=hue(0.6))
```

特别注意, 只有当原点在图片的可视范围内时, 坐标轴才会相交, 并且对于很大的数值, 会使用科学记数法标记。

```
sage: plot(x^2,(x,300,500))
```

你可以把多个图像组合在一起:

```
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)
```

画实心图形的一个好方法是先生成点的列表 (下面例子中的 L) 再用 polygon 命令画出由这些点形成的边界所组成的图形。比如下面是一个绿色的曲边三角形:

```
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
```

输入 show(p, axes=false) 可以看到没有坐标轴的图形。

可以在图像上添加文字:

```
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)
```

微积分教师经常在黑板上画下面的图像: 不是 arcsin 的一段, 而是几段, 即函数 $y = \sin(x)$ (其中 x 介于 -2π 与 2π) 关于 45 度线的翻转。下面的 Sage 命令可以构造出该图像:

```
sage: v = [(sin(x),x) for x in srange(-2*float(pi),2*float(pi),0.1)]
sage: line(v)
```

因为正切函数比正弦函数的范围大的多, 如果你想用同样的方法做出反正切函数, 需要调整一下 x 轴的最大、最小值:

```
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)
```

Sage 还可以绘制极坐标图形, 等高线图和向量场图 (对于某些特殊类型的函数) 。下面是等高线图的例子:

```
sage: f = lambda x,y: cos(x*y)
sage: contour_plot(f, (-4, 4), (-4, 4))
```

2.5.2 绘制三维图像

Sage 也能创建三维图像。无论是在 notebook 中，还是在 REPL 中，显示三维图像默认都是调用开源软件包 [Jmol]，它支持使用鼠标旋转和缩放图像。

使用 `plot3d` 绘制形如 $f(x,y)=z$ 的函数图像：

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

或者，你可以用 `parametric_plot3d` 绘制参数曲面，其中 x,y,z 由一个或两个变量（参数，通常为 u 和 v ）确定。上面的图像可以表达为参数方程形式：

```
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))
```

Sage 中第三种绘制三维图像的方法是 `implicit_plot3d`，它绘制形如 `f(x, y, z) = 0`（定义了一个点集）的函数的图像。我们用经典公式绘制一个球面：

```
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))
```

下面是一些例子：

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")
```

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")
```

Twisted torus:

```
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")
```

Lemniscate:

```
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))
```

2.6 函数的一些常见问题

自定义函数的某些问题可能令人迷惑 (如微分或绘图) 。这一节我们讨论一下相关的话题。

有多种方法定义可以被称为“函数”的东西:

1. 定义 Python 函数, 正如 *函数, 缩进和计数* 中所提到的那样。这些函数可用于绘图, 但是不能用于微分和积分。

```
sage: def f(z): return z^2
sage: type(f)
<type 'function'>
sage: f(3)
9
sage: plot(f, 0, 2)
```

注意最后一行的语法。如果使用 `plot(f(z),0,2)` 的话, 会报错。因为在 `f` 的定义中 `z` 是一个形式变量, 在 `f` 之外, `z` 没有定义。实际上只是 `f(z)` 有问题。下面的代码就可以正确工作, 但是一般来说这也是有问题的, 要尽量避免 (参见下面第 4 条) 。

```
sage: var('z')   # 定义 z 为一个变量
z
sage: f(z)
z^2
sage: plot(f(z), 0, 2)
```

这里, `f(x)` 是一个符号表达式, 这是我们下一条要讨论的。

1. 定义“可调用的符号表达式”。可用于绘图、微分和积分。

```
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)
```

注意 g 是可调用的符号表达式, g(x) 是一个与之相关, 但是不同类型的对象, 虽然 g(x) 也可用于绘图、微分等。参见下面第 5 条的展示。

```
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)
```

1. 预定义的 Sage "微积分函数", 可用于绘图。要进行微积分的话, 还要加工一下。

```
sage: type(sin)
<class 'sage.functions.trig.Function_sin'>
sage: plot(sin, 0, 2)
sage: type(sin(x))
<type 'sage.symbolic.expression.Expression'>
sage: plot(sin(x), 0, 2)
```

sin 本身不能进行微分, 至少不会得到 cos.

```
sage: f = sin
sage: f.derivative()
...
AttributeError: ...
```

使用 f = sin(x) 而不是 sin, 就可以进行微分了, 而且比用 f(x) = sin(x) 建立一个可调用的符号表达式要好。

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

下面解释一些常见的问题：

4. 意外的运算结果（Accidental evaluation）

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

问题: `plot(h(x), 0, 4)` 画出来的是直线 $y = x - 2$, 而不是 `h` 定义的折线。为什么？在命令 `plot(h(x), 0, 4)` 中, `h(x)` 首先被计算, 也就是将 `x` 代入 `h`, 从而 `x<2` 被计算。

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

当一个符号表达式被计算时, 象在 `h` 的定义中的那样, 如果不是明确的真, 就会返回假。于是 `h(x)` 计算的结果是 `x-2`, 这也是刚才所绘制的函数。

解决方案：不要用 `plot(h(x), 0, 4)`, 而是用

```
sage: plot(h, 0, 4)
```

5. 意外得到一个常数而不是一个函数

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

问题: `g(3)` 返回错误：“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` 不是一个函数, 而是一个常量, 所以它没有相关的自变量, 你也就不能再做其他运算。

解决方案：有好几种选择。

- 定义 `f` 为符号表达式。

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

- 或者 f 还是原来那样定义, 而把 g 定义为符号表达式。

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

- 或者 f 和 g 都还是原来那样定义, 但是指定你所替代的变量。

```
sage: f = x
sage: g = f.derivative()
sage: g
1
sage: g(x=3)    # 而不是 'g(3)'
1
```

最后, 还有一种方法来说明 f = x 与 f(x) = x 导数间的区别:

```
sage: f(x) = x
sage: g = f.derivative()
sage: g.variables()  # g 的变量
()
sage: g.arguments()  # g 的参数
(x,)
sage: f = x
sage: h = f.derivative()
sage: h.variables()
()
sage: h.arguments()
()
```

正象这个例子所展示的那样, h 不接受任何参数, 这正是 h(3) 报错的原因。

2.7 基本的环

当定义矩阵、向量或者多项式时，有时候需要，有时候是必须要指定其所在的“环”。 环 是一个数学结构，在其上定义的加法和乘法有很好的性质。如果你从来没有听说过这些概念，至少要了解以下这四个常用的环：

- 整数环 $\{..., -1, 0, 1, 2, ...\}$, Sage 中叫 ZZ;
- 有理数环 – 即，整数构成的分数，Sage 中叫 QQ;
- 实数环，Sage 中叫 RR;
- 复数环，Sage 中叫 CC.

你需要了解它们之间的区别，因为对于同一个多项式，处理方式完全取决于它定义在哪个环上。比如说，多项式 $x^2 - 2$ 有两个根，$\pm\sqrt{2}$. 这些根不是有理数，所以如果你是讨论有理系数多项式，那么该多项式不能进行因式分解，如果是实系数，就可以。所以你可能要指定环以保证得到的结果是你所期望的。下面两个命令定义有理系数多项式和实系数多项式集合。集合的名字分别是“ratpoly”和“realpoly”，这两个名字并不重要，但是要注意变量的名字“.<t>”和“.<z>”的使用。

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

现在我们展示 $x^2 - 2$ 的因式分解。

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

对于矩阵存在同样的问题：矩阵的行消去形式取决于其所定义的环，还有它的特征值和特征向量的计算。更多关于多项式构造的内容请参见：多项式，更多关于矩阵的内容请参见 线性代数.

符号 I 代表 -1 的平方根; i 等同于 I. 当然这不是一个有理数

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

注意：如果 i 已经被赋了其他的值，比如循环变量，那么上面的代码不会给出预期的结果。出现这种情况，请输入

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

来得到 i 的原始复数值。

定义复数的时候还有一个细节: 如上所述 `i` 代表 -1 的平方根, 但是它是 -1 的 *形式上的* 或 *符号的* 平方根。调用 `CC(i)` 或 `CC.0` 返回 -1 的 *复的* 平方根。

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

下面是关于 Sage 中基本环的几个例子。上面已经提到有理数环可以用 `QQ`, 也可以用 `RationalField()` (域 是环的一种, 乘法是可交换的, 且所有非零元素均有乘法逆元。所有有理数可以构成一个域, 但是整数不行)

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

十进制数 `1.2` 属于 `QQ`: 正好是有理数的十进制数可以被强制转换为有理数。π 和 $\sqrt{2}$ 都不是有理数:

```
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
```

为用于高等数学, Sage 还知道其他的环, 比如有限域, p-adic 整数, 代数数环, 多项式环和矩阵环。下面是其中一些环的构造:

```
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 线性代数

Sage 提供线性代数的标准构造, 如矩阵的特征多项式, 梯形格式, 迹, 分解等。

构造矩阵和矩阵的乘法都是很容易的, 也是很自然的:

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

注意, 在 Sage 中, 矩阵 A 的核 (kernel) 是“左核” (left kernel) , 即在向量空间中, w 满足 $wA = 0$.

解矩阵方程也很容易, 使用方法 `solve_right`. 执行 `A.solve_right(Y)` 返回一个矩阵 (或向量) X 满足 $AX = Y$:

```
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)
```

反斜杠 \ 可以代替 `solve_right`; 用 `A \ Y` 代替 `A.solve_right(Y)`.

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

如果无解, Sage 返回一个错误:

```
sage: A.solve_right(w)
...
ValueError: matrix equation has no solutions
```

类似的, 使用 `A.solve_left(Y)` 求解满足 $XA = Y$ 的 X.

Sage 还可以计算特征值和特征向量:

```
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)]
```

(`eigenvectors_left` 的输出是三元组的列表: (特征值, 特征向量, 重数)。) 在 `QQ` 或 `RR` 上的特征值和特征向量也可以用 Maxima 计算 (参见: *Maxima*)。

基本的环 中提到, 矩阵所在的环影响它的性质。下面 `matrix` 命令中的第一个参数告诉 Sage 这个矩阵是整数环 (`ZZ`) 上的, 有理数环 (`QQ`) 上的, 还是实数环 (`RR`) 上的:

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

```
[ 1.00000000000000 0.000000000000000]
[0.000000000000000  1.00000000000000]
```

2.8.1 矩阵空间

我们建立由 3×3 的有理数矩阵构成的空间 $\mathrm{Mat}_{3\times 3}(\mathbf{Q})$:

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

(如果要指定 3×4 矩阵组成的空间, 使用 `MatrixSpace(QQ,3,4)`. 如果省略列数, 则默认的等于行数, `MatrixSpace(QQ,3)` 等价于 `MatrixSpace(QQ,3,3)`.) Sage 将矩阵空间的基保存为一个列表。

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

新建一个矩阵作为 `M` 的元素。

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

然后我们计算它约简后的阶梯矩阵形式以及核。

```
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]
```

下面我们展示定义在有限域上的矩阵的运算:

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

我们构造一个由上面的行张成的 $\mathbf{F}_2$ 的子空间。

```
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]
```

S 的基是由 S 的行梯形矩阵形式中的非零元的行得到的。

2.8.2 稀疏线性代数

Sage 支持在 PID 上的稀疏线性代数。

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

Sage 中的多模算法对于方阵效果比较好 (但是对于非方阵效果不怎么好)：

```
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()
```

注意, Python 是区分大小写的:

```
sage: M = MatrixSpace(QQ, 10,10, Sparse=True)
...
TypeError: MatrixSpace() got an unexpected keyword argument 'Sparse'
```

2.9 多项式

这一节我们展示在 Sage 中如何创建和使用多项式。

2.9.1 一元多项式

有三种方法创建多项式环。

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

建立一个多项式环并告诉 Sage 在输出到屏幕时, 使用字符 't' 作为不确定的量。但是这种方法没有定义符号 `t` 在 Sage 中如何使用, 你不能用它输入一个属于 `R` 的多项式 (如 t^2+1)。

另一种方法是

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

这里的 `t` 有同样的问题。

第三种非常方便的方法是

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

或

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

或者, 甚至是

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

这样变量 t 定义为多项式环的不定量, 你可以很容易构造 R 中的元素, 象下面一样。(注意, 第三种方式与 Magma 中的构造方法类似, 但是在 Magma 中, 可以用这种方法定义的对象有很多。)

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

不管你用哪种方法定义一个多项式环, 你都可以将不定量恢复为 0 阶生成器 (0^{th} generator)。

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

复系数多项式的构造是类似的。复数可以视为由实数通过符号 i 生成的。所以可以如下构造:

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

在创建多项式环时, 可以得到环及其生成元, 或者只是生成元:

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

最后, 可以在 $\mathbf{Q}[t]$ 上进行一些运算。

```
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)]
```

注意到因式分解时正确的考虑并记录了单位 (unit)。

如果你要用某些函数, 比如 `R.cyclotomic_polynomial` 做更多的研究, 除了引用 Sage, 还应该尝试找出具体是什么组件计算分圆多项式, 并引用它们。这里, 如果你输入 `R.cyclotomic_polynomial??` 来查看源代码的话, 你会很快看到一行 `f = pari.polcyclo(n)`, 这说明 PARI 被用于求分圆多项式。应该在你的工作中引用 PARI。

两个多项式相除将产生一个分式域中的元素 (由 Sage 自动创建) 。

```
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
```

使用 Laurent 级数, 可以在分式域 `QQ[x]` 上计算级数的展开:

```
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)
```

如果我们命名的变量不同, 得到不同的一元多项式环。

```
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
```

环由变量确定。注意, 用 `x` 再建一个环, 并不能得到一个新的不同的环。

```
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
```

Sage 还支持任何基本环上的幂级数和 Laurent 级数环。下面的例子中，我们新建 $\mathbf{F}_7[[T]]$‘ 的一个元素，并使用除法新建 $\mathbf{F}_7((T))$ 的一个元素。

```
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
```

可以用双中括号的简单形式新建幂级数环：

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

2.9.2 多元多项式

要使用多元多项式，先要声明多项式环和变量。

```
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
```

跟定义一元多项式一样，有多种方法：

```
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
```

如果你希望变量的名字是单个字母，可以用下面的简短形式：

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

下面我们做一些运算。

```
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
```

你也可以用更多的数学记号来构造多项式环。

```
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,))
```

Sage 中, 多元多项式是用 Python 的字典 (dictionaries) 以及多项式的“分配形式” (distributive representation) 实现的。Sage 用了很多 Singular [Si], 如计算最大公因式和理想的 Gröbner 基。

```
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
```

下面我们新建一个由 f 和 g 生成的理想 (f,g), 简单把 `(f,g)` 和 `R` 乘在一起就行 (也可以用 `ideal([f,g])` 或 `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
```

顺便说一下, 上面的 Gröbner 基不是一个列表, 而是一个固定序列。这意味着它有范围 (universe), 有根源 (parent), 并且不能修改 (不能修改是好事, 因为如果改动基的话, 与 Gröbner 基相关的程序都可能出问题) 。

```
sage: B.parent()
Category of sequences in Multivariate Polynomial Ring in x, y over Rational
Field
```

```
sage: B.universe()
Multivariate Polynomial Ring in x, y over Rational Field
sage: B[1] = x
...
ValueError: object is immutable; please change a copy instead.
```

一些 (并不多) 交换代数的函数在 Sage 中也是可用的, 是由 Singular 实现的。比如, 我们可以计算 I 的基本分解和相伴素理想 (associated primes):

```
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 有限群, 阿贝尔群

Sage 支持排列群, 有限典型群 (如 $SU(n,q)$), 有限矩阵群 (使用你的生成元) 和阿贝尔群 (甚至是无限的) 。很多是通过调用 GAP 实现的。

例如, 要建立一个排列群, 只需给定生成元的列表, 象下面这样。

```
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()
Permutation Group with generators [()]
sage: G.random_element()           # random output
(1,5,3)(2,4)
sage: print latex(G)
\langle (3,4), (1,2,3)(4,5) \rangle
```

你还可以得到特征表 (LaTeX 格式的):

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

Sage 还包括在有限域上的典型群和矩阵群：

```
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_class_representatives()
    [
    [1 0]
    [0 1],
    [0 1]
    [6 1],
    ...
    [6 0]
    [0 6]
    ]
sage: G = Sp(4,GF(7))
sage: G._gap_init_()
'Sp(4, 7)'
sage: G
Symplectic Group of rank 2 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
```

你还可以使用阿贝尔群进行计算 (无限的和有限的)：

```
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
```

2.11 数论

Sage 对数论有广泛的支持。例如我们可以在 $\mathbf{Z}/N\mathbf{Z}$ 上进行运算：

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

Sage 包含标准的数论函数。如：

```
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
```

完美!

Sage 的 sigma(n,k) 函数求 n 的所有因子的 k 次幂的和:

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

下面展示的是推广的 Euclidean 算法, Euler 的 ϕ 函数和中国剩余定理:

```
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
```

下面验证关于 $3n+1$ 问题的一些内容。

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

最后我们展示中国剩余定理。

```
sage: x = crt(2, 1, 3, 5); x
-4
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))
[(1, 1, 1, 1), (1, 1, 2), (2, 2), (1, 3), (4,)]
```

2.11.1 p-adic 数

Sage 支持 p-adic 数域。注意, 一旦建立了一个 p-adic 数域, 就不能再修改它的精度了。

```
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)
```

在 p-adic 域或数域上的整数环的实现工作已经完成了很多。感兴趣的读者可以到 Sage 支持论坛 (sage-support Google group) 上向专家们询问相关细节。

许多相关方法已经在 NumberField 类中实现了。

```
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 with structure  of Number Field in a with
defining polynomial x^3 + x^2 - 2*x + 8
sage: K.class_number()
1
```

2.12 Some more advanced mathematics

2.12.1 Algebraic Geometry

You can define arbitrary algebraic varieties in Sage, but sometimes nontrivial functionality is limited to rings over $\mathbf{Q}$ or finite fields. For example, we compute the union of two affine plane curves, then recover the curves as the irreducible components of the union.

```
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 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
]
```

We can also find all points of intersection of the two curves by intersecting them and computing the irreducible components.

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

```
  x + y + 2
  2*y^2 + 4*y + 3
]
```

Thus, e.g., $(1, 0)$ and $(0, 1)$ are on both curves (visibly clear), as are certain (quadratic) points whose y coordinates satisfy $2y^2 + 4y + 3 = 0$.

Sage can compute the toric ideal of the twisted cubic in projective 3 space:

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

2.12.2 Elliptic Curves

Elliptic curve functionality includes most of the elliptic curve functionality of PARI, access to the data in Cremona's online tables (this requires an optional database package), the functionality of mwrank, i.e., 2-descents with computation of the full Mordell-Weil group, the SEA algorithm, computation of all isogenies, much new code for curves over $\mathbf{Q}$, and some of Denis Simon's algebraic descent software.

The command `EllipticCurve` for creating an elliptic curve has many forms:

- EllipticCurve([a_1, a_2, a_3, a_4, a_6]): Returns the elliptic curve

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

  where the a_i's are coerced into the parent of a_1. If all the a_i have parent $\mathbf{Z}$, they are coerced into $\mathbf{Q}$.

- EllipticCurve([a_4, a_6]): Same as above, but $a_1 = a_2 = a_3 = 0$.

- EllipticCurve(label): Returns the elliptic curve over from the Cremona database with the given (new!) Cremona label. The label is a string, such as `"11a"` or `"37b2"`. The letter must be lower case (to distinguish it from the old labeling).
- EllipticCurve(j): Returns an elliptic curve with j-invariant j.
- EllipticCurve(R, $[a_1, a_2, a_3, a_4, a_6]$): Create the elliptic curve over a ring R with given a_i's as above.

We illustrate each of these constructors:

```
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
```

The pair $(0, 0)$ is a point on the elliptic curve E defined by $y^2 + y = x^3 - x$. To create this point in Sage type `E([0,0])`. Sage can add points on such an elliptic curve (recall elliptic curves support an additive group structure where the point at infinity is the zero element and three co-linear points on the curve add to 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
```

The elliptic curves over the complex numbers are parameterized by the j-invariant. Sage computes j-invariant as follows:

```
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.j_invariant()
110592/37
```

If we make a curve with the same j-invariant as that of E, it need not be isomorphic to E. In the following example, the curves are not isomorphic because their conductors are different.

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

However, the twist of F by 2 gives an isomorphic curve.

```
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
```

We can compute the coefficients a_n of the L-series or modular form $\sum_{n=0}^{\infty} a_n q^n$ attached to the elliptic curve. This computation uses the PARI C-library:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: print 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)
```

It only takes a second to compute all a_n for $n \leq 10^5$:

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

Elliptic curves can be constructed using their Cremona labels. This pre-loads the elliptic curve with information about its rank, Tamagawa numbers, regulator, 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
```

We can also access the Cremona database directly.

```
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]}
```

The objects returned from the database are not of type `EllipticCurve`. They are elements of a database and have a couple of fields, and that's it. There is a small version of Cremona's database, which is distributed by default with Sage, and contains limited information about elliptic curves of conductor ≤ 10000. There is also a large optional version, which contains extensive data about all curves of conductor up to 120000 (as of October 2005). There is also a huge (2GB) optional database package for Sage that contains the hundreds of millions of elliptic curves in the Stein-Watkins database.

2.12.3 Dirichlet Characters

A *Dirichlet character* is the extension of a homomorphism $(\mathbf{Z}/N\mathbf{Z})^* \to R^*$, for some ring R, to the map $\mathbf{Z} \to R$ obtained by sending those integers x with $\gcd(N, x) > 1$ to 0.

```
sage: G = DirichletGroup(21)
sage: list(G)
[[1, 1], [-1, 1], [1, zeta6], [-1, zeta6], [1, zeta6 - 1], [-1, zeta6 - 1],
 [1, -1], [-1, -1], [1, -zeta6], [-1, -zeta6], [1, -zeta6 + 1],
 [-1, -zeta6 + 1]]
sage: G.gens()
([-1, 1], [1, zeta6])
sage: len(G)
12
```

Having created the group, we next create an element and compute with it.

```
sage: chi = G.1; chi
[1, zeta6]
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
```

It is also possible to compute the action of the Galois group $\mathrm{Gal}(\mathbf{Q}(\zeta_N)/\mathbf{Q})$ on these characters, as well as the direct product decomposition corresponding to the factorization of the modulus.

```
sage: G.galois_orbits()
[
[[1, 1]],
[[1, zeta6], [1, -zeta6 + 1]],
[[1, zeta6 - 1], [1, -zeta6]],
[[1, -1]],
[[-1, 1]],
[[-1, zeta6], [-1, -zeta6 + 1]],
[[-1, zeta6 - 1], [-1, -zeta6]],
[[-1, -1]]
]

sage: G.decomposition()
[
Group of Dirichlet characters of modulus 3 over Cyclotomic Field of order
6 and degree 2,
Group of Dirichlet characters of modulus 7 over Cyclotomic Field of order
6 and degree 2
]
```

Next, we construct the group of Dirichlet characters mod 20, but with values in $\mathbf{Q}(i)$:

```
sage: G = DirichletGroup(20)
sage: G.list()
[[1, 1], [-1, 1], [1, zeta4], [-1, zeta4], [1, -1], [-1, -1], [1, -zeta4],
 [-1, -zeta4]]
```

We next compute several invariants of `G`:

```
sage: G.gens()
([-1, 1], [1, zeta4])
sage: G.unit_gens()
[11, 17]
sage: G.zeta()
zeta4
sage: G.zeta_order()
4
```

In this example we create a Dirichlet character with values in a number field. We explicitly specify the choice of root of unity by the third argument to `DirichletGroup` below.

```
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 of modulus 5 over Number Field in a with
defining polynomial x^4 + 1
sage: G.list()
[[1], [a^2], [-1], [-a^2]]
```

Here `NumberField(x^4 + 1, 'a')` tells Sage to use the symbol "a" in printing what `K` is (a Number Field in a with defining polynomial $x^4 + 1$). The name "a" is undeclared at this point. Once `a = K.0` (or equivalently `a = K.gen()`) is evaluated, the symbol "a" represents a root of the generating polynomial x^4+1.

2.12.4 Modular Forms

Sage can do some computations related to modular forms, including dimensions, computing spaces of modular symbols, Hecke operators, and decompositions.

There are several functions available for computing dimensions of spaces of modular forms. For example,

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

Next we illustrate computation of Hecke operators on a space of modular symbols of level 1 and weight 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
```

We can also create spaces for $\Gamma_0(N)$ and $\Gamma_1(N)$.

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

Let's compute some characteristic polynomials and q-expansions.

```
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)
]
```

We can even compute spaces of modular symbols with character.

```
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 - 2*zeta6 - 1) * (x - zeta6 - 2) * (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)
]
```

Here is another example of how Sage can compute the action of Hecke operators on a space of modular forms.

```
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()
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
```

Let T_p denote the usual Hecke operators (p prime). How do the Hecke operators T_2, T_3, T_5 act on the space of modular symbols?

```
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]
```

CHAPTER

THREE

交互命令行

本教程中，假定你是用 `sage` 命令打开 Sage 解释器的。这将启动一个定制后的 IPython shell，并导入很多函数和类，可以从命令行随时调用它们。通过修改 `$SAGE_ROOT/ipythonrc` 文件可以定制更多的内容。启动 Sage 后，你看到的输出和下面的类似：

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

```
sage:
```

要退出 Sage，按 Ctrl+D，或者输入 `quit` 或 `exit`.

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

这里 wall time 是把表挂到墙上开始计算的时间。这是相关的，因为 CPU 时间没有跟踪子进程如 GAP 或 Singular 所用的时间。

(可以从终端用 `kill -9` 结束一个 Sage 进程，但是 Sage 不会结束子进程，如 Maple 过程，或清理 `$HOME/.sage/tmp` 中的临时文件。)

3.1 你的 Sage 会话 (session)

会话 (session) 是你从启动 Sage 到退出的输入、输出序列。Sage 通过 IPython 记录所有的 Sage 输入。事实上，如果你用的是交互命令行 (而不是 notebook 界面)，那么任何时候你都可以输出 `%hist` 来得到所有输入行的列表。你可以输入 ？查看更多关于 IPython 的信息。如，“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)
```

这是一个例子：

```
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
```

在本教程和其他 Sage 文档中，我们省略了输出结果的编号。

你还可以将会话中的输入列表保存到一个宏中。

```
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
```

当使用交互命令行时, 任何 UNIX 的 shell 命令都可以从 Sage 中执行, 但是命令前面要加上感叹号 !. 例如,

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

返回当前目录的文件列表。

PATH 变量中有 Sage 可执行文件的目录信息, 所以执行 gp, gap, singular, maxima, 等, 可以开始运行这些软件包含在 Sage 中的那个版本。

```
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 记录输入和输出

记录 (logging) 你的 Sage 会话与保存 (saving) 它 (参见 保存和读取完整的会话) 不是一个意思。要记录输入 (记录输出是可选的) , 使用 logstart 命令。输入 logstart? 查看更多信息。可以用这个命令记录所有的输入, 输出, 甚至在将来的会话中重放这些输入 (简单的重新载入记录文件即可) 。

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

如果你是在 Linux KDE 中使用 `konsole` 终端, 那么你可以这样保存会话: 在 `konsole` 中启动 Sage 后, 选择 “settings”, 再选 “history...”, 再选 “set unlimited”。当你准备要保存会话时, 选择 “edit” 再选 “save histor as...” 并输入文件名把会话保存到你的电脑上。保存后, 你可以在编辑器中重新载入并打印。

3.3 粘贴忽略提示符

假设你在读一个 Sage 或 Python 的会话, 你想把它们复制到 Sage 中。但是提示符 >>>` 或 `sage:` 很讨厌。实际上你可以复制并粘贴一个例子到 Sage 中, 包含提示符也没关系。或者说, Sage 的分词器默认的跳过 >>>` 或 `sage:` 提示符。例如,

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

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

3.4 查看命令执行的时间

如果将 %time 放在输入行的开始，那么命令执行的时间会显示在命令的输出之后。例如，我们可以比较几种不同求幂运算的时间。下面的时间在不同的电脑和不同的 Sage 版本中可能很不一样。首先看原始 Python 的计算时间：

```
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
```

这表示总共用了 0.66 秒，“Wall time”即你墙上的时间也是 0.66 秒。如果你的电脑还在运行很多其他的程序，那么 wall time 可能比 CPU 时间多很多。

下面我们查看用 Sage 的整数类型计算指数的时间，它是用 Cython 调用 GMP 库实现的。

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

使用 PARI 的 C 语言接口：

```
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
```

GMP 稍微好一点（跟预期的一样，因为 Sage 内置的 PARI 调用了 GMP 的整数运算）。

还可以象下面这样使用 cputime 命令来查看程序块的运行时间。

```
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
```

walltime 命令和 cputime 命令类似, 只是它计算的是 wall time.

我们还能计算 Sage 中包含的其他计算机代数系统的运算能力。下面的例子中, 每个系统我们都先执行一个无关紧要的命令来启动相应程序。最相关的时间是 wall time. 然而, 如果 wall time 和 CPU 时间差的较多时, 说明有可能存在性能方面的差异。

```
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
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
```

这个测试中, GAP 和 Maxima 最慢 (使用 sage.math.washington.edu 这台机子) 。由于 pexpect 接口的开销, 将这些与 Sage 进行比较可能是不公平的。

3.5 错误和异常

当出错时, 你通常会看到 Python 的“异常” (exception) 。Python 甚至试图指出什么原因导致异常。你经常见到异常的名字, 如 NameError 或 ValueError (参见 Python 参考手册 [Py] 中的完整异常列表) 。例如,

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

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

有时候交互的 debugger 对于除错非常有用。你可以使用 %pdb 打开或关闭它 (默认是关闭的) 。如果引发了一个异常, 并且 debugger 是打开的, 会出现提示符 ipdb>. 在 debugger 中, 你可以输出任何局部变量的状态, 上下移动执行的栈。例如,

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

ipdb>
```

在提示符 ipdb> 中输入 ? 可以得到 debugger 的命令列表:

```
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
whatis where

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

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

按 Ctrl+D 或输入 `quit` 返回 Sage.

3.6 反向查找和 Tab 补全

首先新建一个三维向量空间 $V = \mathbf{Q}^3$:

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

也可以用下面的简洁形式:

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

输入一个命令的开头, 再按 `Ctrl-p` （或者按上箭头）, 就可以回溯输入过的所有以这几个字母开头的命令行。就算是你退出了 Sage 再重新启动, 也能这样用。这一功能用到了 `readline` 包, 基本上所有流行的 Linux 发行版中都能用。

要列出来 V 的所有成员函数是很容易的, 只要用 tab 键。输入 `V.` 再按键盘上的 `[tab key]` 键。

```
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
```

如果你输入一个命令的开头几个字母, 再按 `[tab key]`, 可以得到所有以这些字母开头的命令的列表。

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

如果要查一个具体的函数, 如 coordinates 函数, 输入 `V.coordinates?` 查看帮助, 或者输入 `V.coordinates??` 查看源码。下一节还会讲。

3.7 集成帮助系统

Sage 拥有集成帮助系统。输入函数名加一个问号“?”, 可以查看函数相关的文档。

```
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]
```

如上所示, 输出结果告诉你对象的类型, 在哪个文件中被定义, 以及一些有用的描述和例子, 例子都可以复制到当前的会话中执行。几乎所有这些例子都经过正规的自动测试, 以保证其能够正常运行。

Sage 的另外一个非常符合开源精神的功能是, 如果 `f` 是一个 Python 函数, 那么输入 `f??` 就会显示定义 `f` 的源码。如:

```
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()
```

这就告诉我们 coordinates 所做的工作就是调用 coordinate_vector 函数并将结果输出到一个列表中。coordinate_vector 做了什么?

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

coordinate_vector 函数将它的输入代到环绕空间, 这会影响 V 中 v 的系数向量的计算。空间 V 已经是环绕的, 因为它是 $\mathbf{Q}^3$. 子空间也有 coordinate_vector 函数, 但是是不同的。我们新建一个子空间看看:

```
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)
```

(如果你觉得这个实现不够有效, 请帮助优化线性代数。)

你还可以输入 help(command_name) 或 help(class) 来得到给定类的帮助。

```
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 ...
:
:
```

当你输入 q 以退出帮助系统时, 你的会话跟你进来的时候一样。帮助系统的输出不会象 function_name? 那样搞乱你的会话。输入 help(module_name) 特别有效。例如, 向量空间定义在 sage.modules.free_module 中, 输入 help(sage.modules.free_module) 可以查看整个模块的文档。当使用帮助系统查看文档时, 可以输入 / 进行查找, 或输入 ? 进行反向查找。

3.8 保存和读取个人的对象

假设你正在计算一个矩阵, 或是模块化的符号空间, 你想保存下来以后再用。应该怎么做? 计算机代数系统有以下几种方法保存个人的对象。

1. **保存游戏：** 仅仅支持保存和读取完整的会话 (如, GAP, Magma)
2. **统一的输入 / 输出：** 每一个对象都以可以读回来的方式打印 (GP/PARI) 。
3. **Eval:** 在解释器中将它变成容易执行的代码 (如, Singular, PARI) 。

因为 Sage 使用的是 Python, 它用一种不同的方式实现, 每一个对象都可以序列化, 即, 转换为字符串以便将来恢复。这跟 PARI 中统一的 I/O 的想法是类似的, 只是在屏幕输出时不那么复杂。保存和读取 (在多数情况下) 是自动进行的, 不需要额外编程。这是 Python 从开始就支持的一个简单特性。

几乎所有的 Sage 对象 x, 都能用 save(x, filename) 以压缩格式保存到磁盘上 (或 x.save(filename)) 。读取时使用 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')
```

你现在可以退出 Sage 并重启。然后你可以把 A 找回来:

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

对于其他更复杂的对象, 如椭圆曲线, 也可以这么做。所有与对象有关的数据都会保存下来。如,

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

保存下来的 E 有 153KB, 因为它保存了前 100000 个 a_n.

```
~/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!
```

(在 Python 中, 保存和读取是使用 cPickle 模块实现的。特别的, 一个 Sage 对象 x 可以通过 cPickle.dumps(x, 2) 保存。注意 2!)

Sage 不能保存和读取其他计算机代数系统创建的个人对象, 如, GAP, Singular, Maxima 等。它们重载时标记为“无效”状态。在 GAP 中, 虽然很多对象的输出是一种可重构的形式, 但是很多不行, 所以不能用它们的输出结果重构。

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

GP/PARI 的对象可以保存和读取, 因为他们的输出形式足以重构。

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

已保存的对象可以被不同的架构或操作系统的计算机读取, 如, 你可以在 32 位的 OS X 机上保存一个大的矩阵, 并在 64 位的 Linux 机上重新读入, 转为阶梯矩阵再移回去。多数情况下, 你甚至可以使用不同版本的 Sage 读取已保存的对象, 只要那个对象的代码差别不大。所有的属性与定义对象的类一起被保存下来 (但不是源码) 。如果那个类在新版的 Sage 中不存在, 那么这个对象就不能在新版中被重新读取。但是你可以在老版本中读取, 连同对象的字典 (x.__dict__) 一起保存, 再在新版中打开。

3.8.1 保存为纯文本

你可以把对象保存为 ASCII 文本格式, 只需简单的打开一个文件并将字符串写入 (很多对象都可以这么做) 。写完之后关闭文件。

```
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()
```

3.9 保存和读取完整的会话

Sage 可以非常灵活的保存和读取会话。

命令 `save_session(sessionname)` 将所有当前会话中定义的变量保存为字典，并以 `sessionname` 命名。(很少有变量不能保存的情况，这时该变量就不保存到字典中去。) 会话保存到 `.sobj` 文件中，可以象之前保存的那些对象一样重新读取。当你读取一个会话中的对象时，你会得到一个字典，索引是变量的名字，值是对象。

你可以用 `load_session(sessionname)` 命令把定义在 `sessionname` 中的变量读取到当前会话。注意，这不会清除你在当前会话中已经定义的那些变量；或者说，合并了两个会话。

首先我们打开 Sage 并定义一些变量。

```
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
```

下面保存我们的会话，这将上面每一个变量都保存到一个文件中。查看这个文件，大小是 3K 左右。

```
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
```

最后我们重新启动 Sage，定义另外一个变量，并把我们保存的会话读进来。

```
sage: b = 19
sage: load_session('misc')
Loading a
Loading M
```

```
Loading E
Loading t
```

每一个保存的变量都再次可用，而且变量 b 也没有被覆盖。

```
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
```

3.10 Notebook 界面

要启动 Sage 的 notebook 界面，在 Sage 命令行中输入

```
sage: notebook()
```

这将启动 Sage 的 notebook 界面，并打开默认浏览器查看它。服务器的状态保存在 $HOME/.sage/sage_notebook.

其他的选项包括：

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

这将使用指定目录中的文件开启一个新的 notebook 服务器。当你想对特定项目指定一组工作表，或者同时运行几个独立的 notebook 服务器时，这个操作非常有用。

当你启动 notebook 时，它先在 $HOME/.sage/sage_notebook 中新建如下文件：

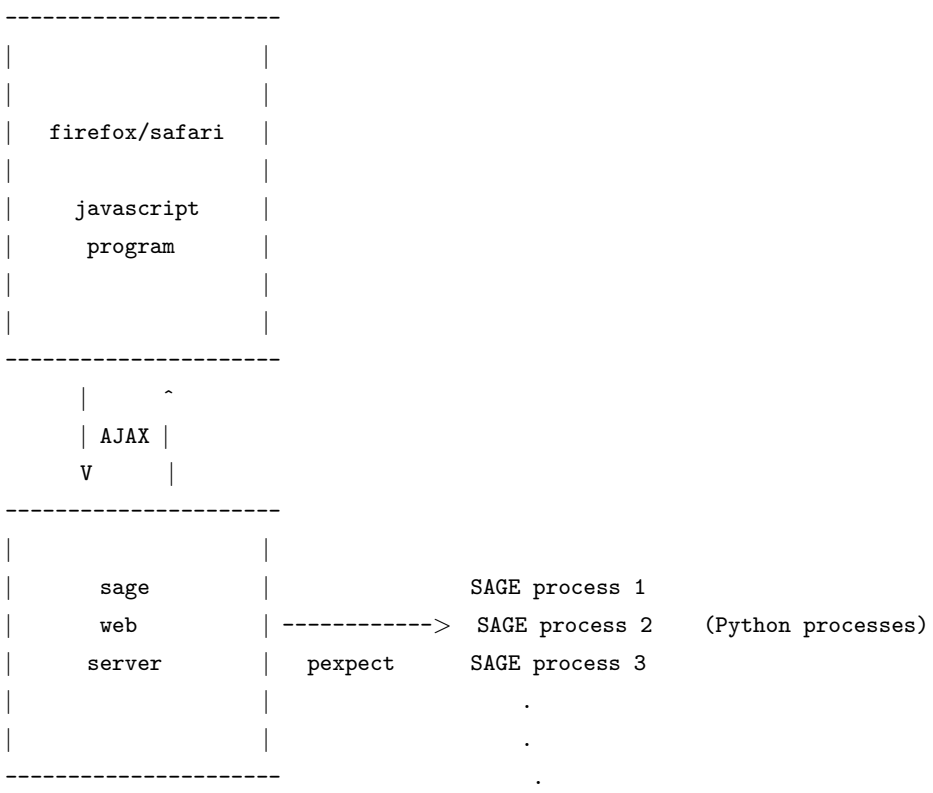
```
nb.sobj       (the notebook SAGE object file)
objects/      (a directory containing SAGE objects)
worksheets/   (a directory containing SAGE worksheets).
```

新建以上文件后，notebook 启动一个 web 服务器。

一个“notebook”是一组用户帐号的集合，每个帐号都可以拥有任意数量的工作表。当你新建一个工作表时，数据保存在 worksheets/username/number 目录。每一个这样的目录里面都有一个纯文本文件 worksheet.txt. 如果你的工作表，或者 Sage，或者别的什么，发生了什么事情，都会记录在这个人工可读的文件中，以便于重构你的工作表。

在 Sage 中, 输入 `notebook?` 查看更多关于如何启动 notebook 服务器的信息。

下面的示意图用来说明 Sage 的 notebook 的架构:



For help on a Sage command, `cmd`, in the notebook browser box, type `cmd?` and now hit <esc> (not <shift-enter>). 要得到 Sage 的 `cmd` 命令的帮助, 在 notebook 的输入框中输入 `cmd?` 再按 <esc> (而不是 <shift-enter>)。

关于 notebook 快捷键的帮助, 请点击 `Help` 链接。

CHAPTER

FOUR

接口

Sage 使用通用接口和简洁的程序语言实现“在一个屋檐下”调用许多不同的计算机代数系统进行计算。

接口的控制台和交互模式做不同的事情。如，使用 GAP：

1. `gap.console()`：打开 GAP 控制台 - 将控制交给 GAP。这时，Sage 的作用仅仅是方便的程序管理器，跟 Linux 的 bash shell 一样。
2. `gap.interact()`：这是一个方便的与运行着的 GAP 进行交互的方法。你可以把 Sage 对象导入 GAP 会话（甚至是从相互接口），等。

4.1 GP/PARI

PARI 是一个紧凑的，非常成熟的，高度优化主要关注数论的 C 语言程序，在 Sage 中有两种不同的接口可以调用：

- `gp` - “**G** o **P** ARI” 解释器
- `pari` - PARI 的 C 语言库。

例如，下面两种方法做同一件事情。它们看上去相同，但是输出完全不同，而且背后所发生的事情也完全不同。

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

前者，一个独立的 GP 解释器作为服务器运行，字符串 `'znprimroot(10007)'` 传给它，由 GP 执行，并将结果赋给一个 GP 变量 (占用 GP 子进程的内存空间并且不会被释放)。然后变量的值显示出来。后者，没有运行单独的程序，字符串 `'znprimroot(10007)'` 由特定的 PARI 的 C 语言库函数执行。结果存储在 Python 的堆中，当不再使用时会被释放。两个对象的类型不同：

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

那么应该用哪一个？取决于你要做什么。在通常 GP/PARI 命令行中可以做的任何事情都可以在 GP 接口中做，因为运行的就是那个程序。特别的，你可以载入复杂的 PARI 程序并执行。相反，PARI 接口（通过 C 语言库）方式限制更多。首先，并没有实现所有的成员函数。第二，很多代码，比如涉及数值积分，不能通过 PARI 接口执行。也可以说，PARI 接口方式比 GP 接口方式更快，稳定性更好。

(如果 GP 接口方式执行一个给定的输入行时内存溢出，它会自动将堆栈大小翻番，并重试。这样，如果你没有正确预计所需的内存量，计算也不会崩溃。这是一个不错的技术，但是通常的 GP 解释器并不提供。对于 PARI 的 C 语言库接口，它会立即将已建立的对象复制出堆栈，这样堆栈就不会再增长了。但是每一个对象不能超过 100MB，否则建立这个对象的时候，堆栈就会溢出。额外的复制操作会稍微影响性能。)

总之，除了使用不同的高级内存管理策略和 Python 语言之外，Sage 使用 PARI 的 C 语言库所提供的功能与 GP/PARI 解释器所提供的功能接近。

首先我们从 Python 列表新建一个 PARI 列表。

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

每一个 PARI 对象都是 `py_pari.gen` 类型的。基本对象的 PARI 类型可由成员函数 `type` 得到。

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

在 PARI 中，使用 `ellinit([1,2,3,4,5])` 新建一个椭圆曲线。Sage 中是类似的，只是 `ellinit` 是一个方法，可由任何 PARI 对象调用，如，我们的 `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]
```

现在我们有一个椭圆曲线对象，可以对它进行一些运算。

```
sage: e.elltors()
[1, [], []]
sage: e.ellglobalred()
[10351, [1, -1, 0, -1], 1]
```

```
sage: f = e.ellchangecurve([1,-1,0,-1])
sage: f[:5]
[1, -1, 0, 4, 3]
```

4.2 GAP

Sage 使用 GAP 4.4.10 计算离散数学, 特别是群论。

这有一个关于 GAP 的 IdGroup 函数的例子, 它依赖一个小的群数据库, 需要单独安装, 后面有说明。

```
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()    # requires optional database_gap package
[ 120, 34 ]
sage: G.Order()
120
```

可以在 Sage 中进行同样的计算, 而且不需要明确的调用 GAP 的接口。

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.center()
Permutation Group with generators [()]
sage: G.group_id()     # requires optional database_gap package
[120, 34]
sage: n = G.order(); n
120
```

(对于某些 GAP 功能, 需要安装两个 Sage 的可选包。输入 sage -optional 显示列表并选择形如 gap_packages-x.y.z 的包, 然后输入 sage -i gap_packages-x.y.z. 同样安装 database_gap-x.y.z. 那些不采用 GPL 协议的 GAP 包, 需要从 GAP 网站 [GAPkg] 上下载, 再解压到 $SAGE_ROOT/local/lib/gap-4.4.10/pkg.)

4.3 Singular

Singular 提供了大量的, 成熟的关于 Gröbner 基, 多元多项式最大公因式, 平面曲线的 Riemann-Roch 空间的基, 因式分解等内容的库函数。我们使用 Sage 的 Singular 接口展示多元多项式的因式分解 (... 不需要输入):

```
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: R1
//   characteristic : 0
//   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')
```

现在已经定义了 f, 我们输出它和它的因式。

```
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^
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
```

跟 *GAP* 中 GAP 的例子一样, 我们可以不用明确的调用 Singular 来进行上面的因式分解 (但是 Sage 在后台是调用 Singular 接口进行实际的计算)。

```
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

Maxima 包含在 Sage 中, 是用 clisp (Lisp 语言的一种) 实现的。Maxima 所使用的开源的基于 Tk/Tcl 的绘图程序 `openmath` 随 Sage 一同发布。然而, gnuplot 包 (Maxima 默认的绘图程序) 作为 Sage 的可选包发布。除了别的功能, Maxima 可以做符号计算。Maxima 可以做符号积分和微分, 解 1 阶常微分方程组, 大

多数 2 阶线性常微分方程组, 并且实现了对任意阶的线性方程组进行 Laplace 变换。Maxima 还了解很多特殊函数, 能够通过 gnuplot 绘图, 进行矩阵运算 (如行消去, 特征值和特征向量), 以及求解多项式方程组。

我们通过构造矩阵来展示 Sage 的 Maxima 接口, 其中 i, j 项是 i/j, $i, j = 1, \ldots, 4$.

```
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]]]]
```

另外一个例子:

```
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]
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
```

最后, 我们给一个通过 Sage 调用 `openmath` 绘图的例子。这里很多内容是根据 Maxima 参考手册修改而来。

绘制多个函数的二维图像:

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

“实时”的三维图像, 你可以用鼠标拖动:

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

下面是著名的 Möbius 带:

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

下面是著名的 Klein 瓶:

```
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]",\
...   "[y, -%pi, %pi]", "['grid, 40, 40]",\
...   '[plot_format, openmath]') # not tested
```

CHAPTER

FIVE

编程

5.1 读取和附加 Sage 文件

下面我们演示如何将单独编写的文件读取到 Sage 中。新建文件 `example.sage`, 包含以下内容:

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

你可以使用 `load` 命令读取和执行 `example.sage` 文件:

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

你也可以用 `attach` 命令, 在当前运行的会话后面附加一个 Sage 文件:

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

现在如果你修改 `example.sage` 并输入一个空白行 (即, 按 `return`), 那么 `example.sage` 的内容将会自动重新读入 Sage。

特别的, 当文件被修改后, "attach" 会自动重载, 这在调试代码的时候很方便。反之, `load` 只能读入一个文件一次。

当 Sage 读入 `example.sage` 时, 将其转为 Python, 之后由 Python 解释器执行。改变是极少的, 主要是将字面上的整数转为 `Integer()`, 字面上的实数转为 `RealNumber()`, 将 `^` 替换为 `**`, 以及类似 `R.2` 转为 `R.gen(2)` 的这种替换。转换后的 `example.sage` 叫 `example.sage.py` 并保存在相同的目录下。这个文件包含以下代码:

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

字面整数被转换, ^ 替换为 **. (Python 中, ^ 表示“异或”, ** 表示“乘方”。)

预分词工作是由 sage/misc/interpreter.py 实现的。

你可以把多行已缩进的代码粘贴到 Sage 中, 只要有一个新行标识代码块的结束 (这在文件中是不需要的)。然而, 将这样的代码输入 Sage 的最好方法是保存为文件, 并使用 attach 命令。

5.2 创建编译代码

在数学计算中, 速度是非常关键的。虽然 Python 是一种非常方便的高级语言, 但是某些微积分运算如果用静态的编译语言实现肯定会比 Python 快几个数量级。如果完全用 Python 编写的话, Sage 的某些运算就会太慢了。为了解决这个问题, Sage 支持一种编译型的 Python, 叫 Cython ([Cython] 和 [Pyrex])。Cython 既像 Python 又像 C。多数 Python 的结构, 包括列表, 条件表达式, 象 += 这样的代码都是可用的。也可以导入你已经写好的其他 Python 模块中的代码。并且还可以声明任意的 C 变量, 可以直接调用任意的 C 语言库。最终的代码被转化为 C 语言, 并使用 C 编译器编译。

为了编译你自己的编译型 Sage 代码, 文件要用扩展名 .spyx (而不是 .sage). 如果是用命令行, 可以跟解释型代码一样, attach 和 load 编译型代码 (notebook 界面下不能 attach 或 load Cython 的代码)。实际的编译过程是后台完成的, 不需要你明确的做什么。参见 $SAGE_ROOT/examples/programming/sagex/factorial.spyx, 这是一个直接调用 GMP 的 C 语言库的编译型的阶乘函数。你自己试一下, 转到 $SAGE_ROOT/examples/programming/sagex/ 目录, 然后这样做:

```
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
```

这里末尾的 L 表示这是一个 Python 长整数 (参见 预处理器：*Sage 与 Python 的差别*).

注意, 当你退出后重新进入 Sage 时, Sage 会重新编译 factorial.spyx. 编译后的共享对象库保存在 $HOME/.sage/temp/hostname/pid/spyx. 这些文件在你退出 Sage 时会被删除。

对于 spyx 文件不会进行预分词, 如, 在 spyx 文件中, 1/3 得到 0 而不是有理数 1/3. 如果 foo 是 Sage 库中的一个函数, 要在 spyx 文件中使用, 需要导入 sage.all 并且使用 sage.all.foo.

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

5.2.1 在单独文件中访问 C 语言函数

访问定义在单独的 *.c 文件中的 C 语言函数也是很容易的。这有一个例子。在同一个目录下新建文件 `test.c` 和 `test.spyx`, 包含以下内容:

纯 C 代码: `test.c`

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

Cython 代码: `test.spyx`:

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

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

下面这样做:

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

如果在 Cython 文件中编译 C 代码还要用到另外一个 `foo` 库, 在 Cython 源码中加一行 `clib foo`. 类似的, 可以用声明 `cfile bar` 来包含一个 C 文件 `bar`.

5.3 独立的 Python/Sage 脚本

下面的 Sage 脚本对整数或多项式进行因式分解:

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

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]))
```

要使用这个脚本, 你的 SAGE_ROOT 必须在在 PATH 中。如果上面的脚本叫 factor, 下面是用法示例:

```
bash $ ./factor 2006
2 * 17 * 59
bash $ ./factor "32*x^5-1"
(2*x - 1) * (16*x^4 + 8*x^3 + 4*x^2 + 2*x + 1)
```

5.4 数据类型

Sage 中的每一个对象都有良好定义的类型。Python 有广泛的基本内置类型, Sage 库又添加了很多。Python 内置的类型包括字符串, 列表, 元组, 整数和实数等, 如下:

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

Sage 增加了很多其他类型, 如, 向量空间:

```
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'>
```

只有特定的函数才能在作用在 V 上。其他数学软件中, 可能会用“函数”形式 foo(V,...). 在 Sage 中, 特定的函数附加于 V 的类型 (或类) 上, 并使用类似 Java 或 C++ 的面向对象的语法, 即, V.foo(...). 这可以使全局的命名空间保持整洁, 而不被成千上万的函数搞乱。而且不同作用的函数都可以叫 foo, 还不用做参数的类型检查 (或 case 语句) 来决定要调用哪一个。如果你重用了一个函数的名字, 那个函数还是可用的 (如, 你把什么东西命名为 zeta, 然后又想计算 0.5 的 Riemann-Zeta 函数值, 你还是可以输入 s=.5; s.zeta()).

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

通常情况下，常用的函数调用方式也是支持的，这样方便些，而且数学表达式用面向对象的方式调用看着不习惯。这有几个例子。

```
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)
    ]
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
```

要列出 A 的所有成员函数，使用 tab 补全功能。先输入 `A.`，再按 `[tab]`，正如 *反向查找和 Tab 补全* 中所述。

5.5 列表，元素和序列

列表类型保存任意类型的元素。和 C，C++ 等等一样（但是和多数标准的计算机代数系统不一样），列表中的元素下标从 0 开始计数。

```
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)
<type 'list'>
sage: v[0]
2
sage: v[2]
5
```

(索引一个列表时, 如果下标不是 Python 整数类型也是可以的!) Sage 整数 (或有理数, 或其他有 __index__ 方法的对象) 都可以正常索引。

```
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
```

range 函数新建一个 Python 整数 (不是 Sage 整数) 的列表:

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

使用包含列表 (list comprehension) 的方式构造列表是非常有用的:

```
sage: L = [factor(n) for n in range(1, 15)]
sage: print 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.Factorization'>
sage: [factor(n) for n in range(1, 15) if is_odd(n)]
[1, 3, 5, 7, 3^2, 11, 13]
```

更多关于如何使用包含列表创建列表的内容, 请参见 [PyT].

列表切片 (list slicing) 是一个有趣的功能。若 L 是一个列表, 那么 L[m:n] 返回 L 的一个子列表, 从第 m 个元素开始到第 $n-1$ 个元素, 象下面这样。

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

元组 (tuple) 与列表类似, 除了它们是固定的, 也就是说, 一旦它们被建立, 就不能再修改。

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

序列 (sequence) 是第三种面向列表的 Sage 类型。与列表和元组不同, 序列不是 Python 内置的类型。序列默认是可修改的, 但是可以使用 `Sequence` 类中的 `set_immutable` 方法设置成不能修改的, 如下例所示。一个序列中的所有元素都有共同的祖先 (parent), 称为序列的领域 (sequences universe).

```
sage: v = Sequence([1,2,3,4/5])
sage: v
[1, 2, 3, 4/5]
sage: type(v)
<class 'sage.structure.sequence.Sequence'>
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
...
ValueError: object is immutable; please change a copy instead.
```

序列是从列表继承下来的, 可用于任何列表可用的地方:

```
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))
<type 'list'>
```

另外一个例子, 向量空间的基是不可修改的序列, 因为你不能修改它们。

```
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'>
sage: B[0] = B[1]
...
ValueError: object is immutable; please change a copy instead.
sage: B.universe()
Vector space of dimension 3 over Rational Field
```

5.6 字典

字典 (dictionary)(有时也称为联合数组) 是从“可乱序”(hashable) 的对象 (如字符串, 数字和元组等, 详细信息参见 Python 文档 http://docs.python.org/tut/node7.html 和 http://docs.python.org/lib/typesmapping.html) 到任意对象的映射。

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

第三个键值展示了字典的索引可以是复杂的对象, 如, 整数环。

你可以将上面的字典转换为同样内容的列表:

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

一个常用的操作是在字典的对中遍历:

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

字典是没有顺序的, 如最后一个例子所示。

5.7 集合

Python 有内置的集合类型。主要的功能是迅速的判断一个元素是否属于集合，以及标准的集合运算。

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

Sage 也有自己的集合类型，多数情况下是用 Python 内置的集合类型实现的，但是有一些额外的 Sage 相关的函数。新建一个 Sage 的集合使用 `Set(...)`.

```
sage: X = Set([1,19,'a']);   Y = Set([1,1,1, 2/3])
sage: X
{'a', 1, 19}
sage: Y
{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
```

5.8 迭代器

迭代器是最近才加入 Python 中的，在数学应用中特别有用。下面是几个例子，更多内容请参见 [PyT]. 我们生成一个在不超过 10000000 的非负整数的平方数上的迭代器。

```
sage: v = (n^2 for n in xrange(10000000))
sage: v.next()
0
sage: v.next()
1
sage: v.next()
4
```

我们新建一个在形如 $4p+1$ (p 为素数) 的素数上的迭代器, 并观察前面几个。

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

特定的环, 如有限域和整数环, 其上都有迭代器:

```
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: W.next()
(0, 0)
sage: W.next()
(0, 1)
sage: W.next()
(0, -1)
```

5.9 循环, 函数, 控制语句和比较

我们已经见到几个 `for` 循环常见用法的例子。Python 中, `for` 循环是缩进的, 如

```
>>> for i in range(5):
...       print(i)

0
1
2
3
4
```

注意 for 语句结尾处的冒号 (不象 GAP 或 Maple, 这里没有“do”或“od”), 和循环体 (也就是 `print(i)`) 的缩进。缩进是非常重要的。输入“:”后按 `enter`, Sage 会自动为你缩进, 象下面这样。

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

```
2
3
4
```

符号 = 用于赋值。符号 == 用于比较是否相等:

```
sage: for i in range(15):
...       if gcd(i,15) == 1:
...           print(i)
1
2
4
7
8
11
13
14
```

要记住, 缩进决定了 if, for 和 while 语句的结构:

```
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
```

当然, 这不是 Legendre 符号的一个有效实现, 只是为了展示 Python/Sage 编程的某些方面。Sage 中函数{kronecker} 通过 C 语言库调用 PARI 来有效的计算 Legendre 符号。

最后我们看一下比较运算符, 如 ==, !=, <=, >=, >, <, 如果可能的话, 会自动将两边的数据转换为同一类型:

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

几乎任意两个对象都可以进行比较, 没有假设对象是全序的。

```
sage: 2 < CC(3.1,1)
True
sage: 5 < VectorSpace(QQ,3)   # output can be somewhat random
True
```

对于符号不等式，使用 bool 函数：

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

Sage 中比较两个不同类型的对象时，多数情况下 Sage 会尝试强制将对象转换为规范的共同父类。如果成功转换，比较就在转换后的对象间进行；如果转换不成功，对象被认为不相等。要测试两个变量是否指向同一个对象，使用 `is`. 如：

```
sage: 1 is 2/2
False
sage: 1 is 1
False
sage: 1 == 2/2
True
```

下面两行中，头一行不相等是因为没有规范映射 $\mathbf{Q} \to \mathbf{F}_5$, 从而没有规范的方法比较 $\mathbf{F}_5$ 中的 1 和 $\mathbf{Q}$ 中的 1. 相反，存在规范映射 $\mathbf{Z} \to \mathbf{F}_5$, 所以第二个比较是 `True`. 注意到顺序不影响结果。

```
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
```

警告: Sage 中的比较比 Magma 中更严格，Magma 中 $\mathbf{F}_5$ 中的 1 与 $\mathbf{Q}$ 中的 1 相等。

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

5.10 性能分析

本节作者: Martin Albrecht (malb@informatik.uni-bremen.de)

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

有时检查代码的瓶颈对于了解哪一部分占用了最多的计算时间是很有用的。这可以帮助确定最需要优化哪一部分。Python 和 Sage 提供几种性能分析选项。

最简单的是在交互命令行中使用 `prun` 命令。它返回一个描述每个函数占用多少计算时间的摘要。例如，要分析有限域上的矩阵乘法，可以这样做：

```
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)
  ...
```

这里 `ncals` 是调用的次数，`tottime` 是给定函数所用的总时间 (不包括调用子函数的时间)，`percal` 是 `tottime` 除以 `ncals` 的商. `cumtime` 是函数用的时间和所有子函数用的时间 (即从调用开始到退出的时间)，`percall` 是 `cumtime` 除以基本调用次数的商，`filename:lineno(function)` 提供了每个函数的相关信息。经验规律是：列表中函数排的越靠前，所花费的时间越多，也就越需要优化。

通常，`prun?` 会提供如何使用性能分析器的详细信息，并解析输出结果的含义。

分析的数据可以写入一个对象，这样可以就近检查：

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

注意：输入 `stats = prun -r A\*A` 会显示语法错误，因为 prun 是 IPython shell 的命令，不是一个正常的函数。

要想得到漂亮的图形化分析结果，可以使用 hotshot 分析器。它是调用 `hotshot2cachetree` 和程序 `kachegrind` (仅在 Unix 下有效) 的一个小脚本。使用 hotshot 分析器分析同一个例子：

```
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()
```

结果保存在当前工作目录的 `pythongrind.prof` 文件中。它可以被转换为可以可视化的 cachegrind 格式。

在系统 shell 中输入

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

输出文件 `cachegrind.out.42` 可以由 `kcachegrind` 查看。请注意遵守命名习惯 `cachegrind.out.XX`.

CHAPTER

SIX

分布式计算

Sage 内置了强大的分布式计算框架“分布式 Sage”(dsage).

6.1 概要

分布式 Sage 是一个框架, 允许用户在 Sage 中进行分布式计算。它包括一个服务器, 客户端和若干计算节点, 以及可编写计算任务的类。它被设计成主要用于粗糙的分布式计算, 即, 任务之间联系不多的计算。有时候也被称为网格计算。

分布式 Sage 由 3 个部分组成:

1. **服务器** 的职责是任务的分配, 提交和收集。还包括一个 web 接口, 以便于监视计算任务和进行其他管理的工作。
2. **客户端** 的职责是提交新的任务给服务器并收集计算结果。
3. **计算节点** 执行实际的计算。

6.2 快速入门

这有几个例子展示如何使用 dsage.

6.2.1 例 1

1. 运行 `dsage.setup()`. 将设置 SQLite 数据库并为 SSL 通信生成公钥和私钥。还根据你当前的用户名添加一个默认的用户。
2. 运行 `d = dsage.start_all()`. 该命令启动服务器, web 服务器, 2 个计算节点, 并返回一个对象 (d), 它是一个与服务器的连接。从这儿开始, 你与 dsage 的交互任务主要通过 d 来完成。

3. 打开浏览器，转到 http://localhost:8082 查看 dsage 的 web 界面。在这里你可以看到计算任务的状态，计算节点的连接以及 dsage 服务器的其他一些重要信息。(译注：在我的机子上，每次的端口号都不一样，似乎也没有规律。可以使用 nc -zv localhost 1-65535 命令查看当前所有打开的端口，挨个尝试。)

4. 尝试一个简单的例子。输入 `job=d('2+2')`. 如果你看着 Web 界面，会看到表格中出现一个新的任务。现在一个计算节点接受这个任务，执行后将结果显示给你。要得到结果，输入 `job.result`. 可能现在还没有结果，因为对于简单的计算，网络通信占用了大量的计算时间。如果你希望等待你的任务结束，可以调用 `job.wait()`，这将阻断通信直到任务完成，那时可以用 `job.result` 查看结果。你可以用这种方法调用 `d` 进行任何的计算。

6.2.2 例 2

在这个例子中，我们演示如何使用 dsage 的内置类 DistributedFactor. DistributedFactor 混合使用 ECM 和 QSieve 算法进行因子分解，对小的因子进行试验分解。

1. 如果还没有启动 dsage 会话，就运行 `d = dsage.start_all()`，否则可以接着使用之前的 `d` 实例。

2. 使用 `factor_job = DistributedFactor(d, number)` 启动分布式分解任务。你可以选一个相当大的值，比如 $2^{360} - 1$. 可以检查 `factor_job.done` 属性来查看分解任务是否已经完成。如果已经完成，可以使用 `factor_job.prime_factors` 查看它找到的素因子。

6.2.3 例 3

这个例子演示分布式的 map 方法。你可以在 http://docs.python.org/lib/build-in-funcs.html 找到关于正规 map 方法的文档。语法是完全相同的。

首先，如果还没有启动 dsage 会话，就运行 `d = dsage.start_all()`，否则可以接着使用之前的 `d` 实例。

```
sage: def f(n): return n*n
sage: j = d.map(f, [25,12,25,32,12])
sage: jobs = d.block_on_jobs(j)
```

这将堵断通信直到 `f` 对每一个输入都运算完毕。

6.2.4 例 4

这个例子演示如何对一个 dsage 实例使用 `@parallel` 修饰。

首先，如果还没有启动 dsage 会话，就运行 `d = dsage.start_all()`，否则可以接着使用之前的 `d` 实例。

```
sage: P = parallel(p_iter = d.parallel_iter)
sage: @P
```

```
...   def f(n,m):  return n+m
sage: f([(1,2), (5, 10/3)])
```

6.3 文件

dsage 将新文件保存在 $SAGE_ROOT/.sage/dsage:

1. pubcert.pem 和 cacert.pem: 用于 SSL 通信的公钥和私钥。
2. dsage_key.pub 和 dsage_key: 用于用户授权的密钥。
3. 目录 db/: 包含 dsage 的数据库。
4. *.log 文件: 由服务器和计算节点生成的日志文件。
5. 目录 tmp_worker_files/: 计算节点在计算时, 将任务保存在这里。

CHAPTER

SEVEN

后记

7.1 为什么选择 Python?

7.1.1 Python 的优点

虽然为了提高速度的那些代码必须使用编译语言实现，但 Sage 的基本语言是 Python(参见 [Py])。Python 有几个优点：

- Python 对 **保存对象** 支持的很好。在 Python 中，可以将几乎所有的对象保存到磁盘文件或数据库中。
- 源码中函数和包有好的 **文档** 支持，包括自动提取文档和自动测试所有例子。例子被自动测试以保证可以正确工作。
- **内存管理**：Python 拥有良好的健壮的内存管理和回收机制，可以正确处理循环引用，并且在文件中允许使用局部变量。
- Python 有 **很多包** 可能是 Sage 用户很感兴趣的：数值分析和线性代数，二维和三维可视化，网络 (用于分布式计算和服务器)，数据库，等等。
- **可移植性**：在很多平台上，Python 可以很容易由源码编译。
- **异常处理**：Python 有精心设计的高级异常处理系统，当所调用的代码出错时也能恢复。
- **调试**：Python 包括一个调试器，当代码由于某种原因失败时，用户可以访问扩展堆栈，查看所有相关变量，以及上下移动堆栈。
- **性能分析器**：Python 的性能分析器可以运行代码并建立详细的报告，包括每个函数被调用了多少次，运行时间等。
- **统一语言**：不象 Magma, Maple, Mathematica, Matlab, GP/PARI, GAP, Macaular 2, Simath 等软件那样，为数学又新建一种语言，我们使用的 Python 语言是流行的计算机语言，由许多有经验的工程师开发和优化。Python 成熟的开发过程是开源软件成功的实例 (参见 [PyDev])。

7.1.2 预处理器：Sage 与 Python 的差别

Python 的某些数学方面容易让人弄混，因此 Sage 在这些方面与 Python 不同。

- **指数的记号：** ** 与 ^. Python 中，^ 是“xor”的意思，不是指数，所以在 Python 中有

```
>>> 2^8
10
>>> 3^2
1
>>> 3**2
9
```

^ 的这种用法看上去很奇怪，在数学研究中也没什么用，因为很少用“异或”函数。为了方便，Sage 在将命令传给 Python 之前，先进行预处理，将不是字符串中的 ^ 替换为 **.

```
sage: 2^8
256
sage: 3^2
9
sage: "3^2"
'3^2'
```

- **整数除法：** Python 表达式 `2/3` 不会产生数学家期待的结果。Python 中，如果 `m` 和 `n` 都是整数，那么 `m/n` 也是整数，即 `m` 除以 `n` 的商。所以 `2/3=0`. Python 社区中有一些关于修改 Python 的讨论，使得 `2/3` 的返回浮点数 `0.6666...`，而 `2//3` 返回 `0`.

  我们在 Sage 解释器中处理这个问题，将字面的整数转为 `Integer()`，并使除法作为有理数的构造器。如：

```
sage: 2/3
2/3
sage: (2/3).parent()
Rational Field
sage: 2//3
0
sage: int(2)/int(3)
0
```

- **长整数：** 除 C 的整型外，Python 本身支持任意精度的整数。这比 GMP 所提供的整型要慢的多，而且输出时末尾会有一个 `L` 以便与整型区分（这个性质短期内不会改变）。Sage 使用 GMP 的 C 语言库实现任意精度的整型，并且输出时没有 `L`.

我们不是去修改 Python 的解释器（有些人在内部项目中是这样做的），我们用的就是 Python 语言本身，只是为 IPython 增加了一个预处理器，使得命令行的工作方式与数学家的习惯一致。这意味着已有的 Python 代码都可以用于 Sage。然而在编写一个要导入 Sage 的包时，还是要遵守标准 Python 的规则。

(要安装一个 Python 库, 比如你从互联网上找的, 跟着指南做就行, 只是要执行 `sage -python` 而不是 `python`. 多数情况下是输入 `sage -python setup.py install`.)

7.2 我想做点贡献。怎么做？

如果你愿意为 Sage 做点贡献, 非常感谢! 提交代码, 添加文档, 报告 Bug, 都是欢迎的。

浏览 Sage 网站上关于开发人员的信息, 除了别的信息, 你可以找到一个 Sage 相关项目的长长的列表, 已根据类别和优先级排序。“Sage 编程指南”是有用的信息, 同样, 你也可以看看名为 `sage-devel` 的 Google 讨论组。

7.3 如何引用 Sage？

如果你的论文中用到了 Sage, 请在使用 Sage 计算的结果处引用 Sage, 参考文献中包含如下条目:

```
[SAGE], SAGE Mathematical Software, Version 2.6, http://www.sagemath.org
```

(将 2.6 替换为你所用的 Sage 版本)。并且要注意查看你的计算用到了 Sage 的哪些模块, 如, PARI? GAP? Singular? Maxima? 也要引用那些系统。如果你对用到了哪些软件有疑问, 可以在 `sage-devel` Google 讨论组中提问。关于这一点的更多讨论参见 一元多项式.

如果你直接读完该教程, 并且对于它花了你多少时间有一些概念, 请让我们知道, 可在 `Sage-devel` Google 讨论组中发贴。

祝使用 Sage 愉快!

CHAPTER

EIGHT

附录

8.1 二元数学运算符优先级

`3^2*4 + 2%5` 等于多少？由“运算符优先级表”得到的结果是 38. 下表基于 *Python 语言参考手册* 中的 § 5.14, 作者是 G. Rossum 和 F. Drake. 运算符是按优先级从低到高的顺序排列。

运算符	说明
or	逻辑或
and	逻辑与
not	逻辑非
in, not in	隶属关系
is, is not	类型检测
>, <=, >, >=, ==, !=, <>	比较
+, -	加、减
*, /, %	乘、除、余
**, ^	指数

所以，计算 `3^2*4 + 2%5` 时，Sage 是这样加括号的: `((3^2)*4) + (2%5)`. 这样，先计算 `3^2` 得 `9`, 再计算 `(3^2)*4` 和 `2%5`, 最后加起来。

CHAPTER

NINE

参考文献

CHAPTER

TEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

BIBLIOGRAPHY

[Dive] Dive into Python, Freely available online at http://diveintopython.org

[PyT] The Python Tutorial, http://www.python.org/

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

[GAP] The GAP Group, `GAP - Groups, Algorithms, and Programming`, http://www.gap-system.org

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

[Jmol] Jmol: an open-source Java viewer for chemical structures in 3D http://www.jmol.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 .

[Py] The Python language http://www.python.org/ , Reference Manual http://docs.python.org/ref/ref.html

[GAPkg] GAP Packages, http://www.gap-system.org/Packages/packages.html

[Cython] Cython, http://www.cython.org

[Pyrex] Pyrex, http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/

[PyT] The Python Tutorial, http://www.python.org/

[Py] The Python language http://www.python.org/ , Reference Manual http://docs.python.org/ref/ref.html

[PyDev] Guido, Some Guys, and a Mailing List: How Python is Developed, http://www.python.org/dev/dev_intro.html.

[Cyt] Cython, http://www.cython.org.

[Dive] Dive into Python, Freely available online at http://diveintopython.org.

[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/

[Py] The Python language http://www.python.org/ Reference Manual http://docs.python.org/ref/ref.html.

[PyDev] Guido, Some Guys, and a Mailing List: How Python is Developed, http://www.python.org/dev/dev_intro.html.

[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/ and http://sage.sf.net/.

[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.