Tutoriel Sage

Version 4.1.1

The Sage Group

14 August 2009

Table des matières

Sage est un logiciel mathématique libre destiné à la recherche et à l'enseignement en algèbre, géométrie, arithmétique, théorie des nombres, cryptographie, calcul scientifique et dans d'autres domaines apparentés. Le modèle de développement de Sage comme ses caractéristiques techniques se distinguent par un souci extrême d'ouverture, de partage, de coopération et de collaboration : notre but est de construire la voiture, non de réinventer la roue. L'objectif général de Sage est de créer une alternative libre viable à Maple, Mathematica, Magma et MATLAB.

Ce tutoriel est la meilleure façon de se familiariser avec Sage en quelques heures. Il est disponible en versions HTML et PDF, ainsi que depuis le notebook Sage (cliquez sur `Help`, puis sur `Tutorial` pour parcourir le tutoriel de façon interactive depuis Sage).

CHAPITRE 1

Introduction

Explorer ce tutoriel en entier devrait vous prendre au maximum trois à quatre heures. Vous pouvez le lire en version HTML ou PDF, ou encore l'explorer interactivement à l'intérieur de Sage en cliquant sur `Help` puis sur `Tutorial` depuis le *notebook* (il est possible que vous tombiez sur la version en anglais).

Sage est écrit en grande partie en Python, mais aucune connaissance de Python n'est nécessaire pour lire ce tutoriel. Par la suite, vous souhaiterez sans doute apprendre Python, et il existe pour cela de nombreuses ressources libres d'excellente qualité, dont [PyT] et [Dive]. Mais si ce que vous voulez est découvrir rapidement Sage, ce tutoriel est le bon endroit où commencer. Voici quelques exemples :

```
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 Installation

Si Sage n'est pas installé sur votre ordinateur, vous pouvez essayer quelques commandes en ligne à l'adresse http ://www.sagenb.org.

Des instructions pour installer Sage sur votre ordinateur sont disponibles dans le guide d'installation (*Installation Guide*), dans la section documentation de la page web principale de Sage [Sage]. Nous nous limiterons ici à deux remarques.

1. La version téléchargeable de Sage vient avec ses dépendances. Autrement dit, bien que Sage utilise Python, IPython, PARI, GAP, Singular, Maxima, NTL, GMP, etc., vous n'avez pas besoin de les installer séparément, ils sont fournis dans la distribution Sage. En revanche, pour utiliser certaines des fonctionnalités de Sage, par exemple Macaulay ou KASH, il vous faudra d'abord installer le paquet optionnel correspondant, ou du moins avoir le logiciel correspondant installé sur votre ordinateur. Macaulay et KASH sont disponibles pour SAGE sous forme de modules optionnels (pour une liste de tous les paquets optionnels disponibles, tapez `sage -optional` ou visitez la page *Download* (Téléchargement) du site web de Sage).
2. La version binaire pré-compilée de Sage (disponible sur le site web) est souvent plus facile et plus rapide à installer que la distribution en code source. Pour l'installer, décompressez l'archive et lancez simplement le programme `sage`.

1.2 Les différentes manières d'utiliser Sage

Il y a plusieurs façons d'utiliser Sage.
– **Interface graphique (« notebook ») :** voir la section sur le *Notebook* du manuel de référence, et *L'interface notebook* ci-dessous ;
– **Ligne de commande :** voir *La ligne de commande interactive* ;
– **Programmes :** en écrivant des programmes interprétés ou compilés en Sage (voir *Charger et attacher des fichiers Sage* et *Écrire des programmes compilés*) ;
– **Scripts :** en écrivant des programmes Python indépendants qui font appel à la bibliothèque Sage (voir *Scripts Python/Sage autonomes*).

1.3 Objectifs à long terme de Sage

– **Être utile :** le public visé par Sage comprend les étudiants (du lycée au doctorat), les enseignants et les chercheurs en mathématiques. Le but est de fournir un logiciel qui permette d'explorer toutes sortes de constructions mathématiques et de faire des expériences avec, en algèbre, en géométrie, en arithmétique et théorie des nombres, en analyse, en calcul numérique, etc. Sage facilite l'expérimentation interactive avec des objets mathématiques.
– **Être efficace :** c'est-à-dire rapide. Sage fait appel à des logiciels matures et soigneusement optimisés comme GMP, PARI, GAP et NTL, ce qui le rend très rapide pour certaines opérations.
– **Être libre/open-source :** le code source doit être disponible librement et lisible, de sorte que les utilisateurs puissent comprendre ce que fait le système et l'étendre facilement. Tout comme les mathématiciens acquièrent une compréhension plus profonde d'un théorème en lisant sa preuve soigneusement, ou simplement en la parcourant, les

personnes qui font des calculs devraient être en mesure de comprendre comment ceux-ci fonctionnent en lisant un code source documenté. Si vous publiez un article dans lequel vous utilisez Sage pour faire des calculs, vous avez la garantie que vos lecteurs auront accès librement à Sage et à son code source, et vous pouvez même archiver et redistribuer vous-même la version de Sage que vous utilisez.

– **Être facile à compiler :** le code source de Sage devrait être facile à compiler pour les utilisateurs de Linux, d'OS X et de Windows. Cela rend le système plus flexible pour les utilisateurs qui souhaiteraient le modifier.
– **Favoriser la coopération :** fournir des interfaces robustes à la plupart des autres systèmes de calcul formel, notamment PARI, GAP, Singular, Maxima, KASH, Magma, Maple et Mathematica. Sage cherche à unifier et étendre les logiciels existants.
– **Être bien documenté :** tutoriel, guide du programmeur, manuel de référence, guides pratiques, avec de nombreux exemples et une discussion des concepts mathématiques sous-jacents.
– **Être extensible :** permettre de définir de nouveaux types de données ou des types dérivés de types existants, et d'utiliser du code écrit dans différents langages.
– **Être convivial :** il doit être facile de comprendre quelles fonctionnalités sont disponibles pour travailler avec un objet donné, et de consulter la documentation et le code source. Également, arriver à un bon niveau d'assistance utilisateur.

CHAPITRE 2

Visite guidée

Cette partie présente une sélection des possibilités actuellement offertes par Sage. Pour plus d'exemples, on se reportera à *Sage Constructions* (« Construction d'objets en Sage »), dont le but est de répondre à la question récurrente du type « Comment faire pour construire ... ? ».

On consultera aussi le *Sage Reference Manual* (« Manuel de référence pour Sage »), qui contient des milliers d'exemples supplémentaires. Notez également que vous pouvez explorer interactivement ce tutoriel — ou sa version en anglais — en cliquant sur `Help` à partir du notebook de Sage.

(Si vous lisez ce tutoriel à partir du *notebook* de Sage, appuyez sur `maj-enter` pour évaluer le contenu d'une cellule. Vous pouvez même éditer le contenu avant d'appuyer sur `maj-entrée`. Sur certains Macs, il vous faudra peut-être appuyer sur `maj-return` plutôt que `maj-entrée`).

2.1 Affectation, égalité et arithmétique

A quelques exceptions mineures près, Sage utilise le langage de programmation Python, si bien que la plupart des ouvrages d'introduction à Python vous aideront à apprendre Sage.

Sage utilise = pour les affectations. Il utilise ==, <=, >=, < et > pour les comparaisons.

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

Sage fournit toutes les opérations mathématiques de base :

```
sage: 2**3    #  ** désigne l'exponentiation
8
sage: 2^3     #  ^ est un synonyme de ** (contrairement à Python)
8
sage: 10 % 3  #  pour des arguments entiers, % signifie mod, i.e., le reste dans la division euclidie
```

```
1
sage: 10/4
5/2
sage: 10//4   #  pour des arguments entiers, // renvoie le quotient dans la division euclidienne
2
sage: 4 * (10 // 4) + 10 % 4 == 10
True
sage: 3^2*4 + 2%5
38
```

Le calcul d'une expression telle que `3^2*4 + 2%5` dépend de l'ordre dans lequel les opérations sont effectuées ; ceci est expliqué dans l'annexe *Priorité des opérateurs arithmétiques binaires* *Priorité des opérateurs arithmétiques binaires*.

Sage fournit également un grand nombre de fonctions mathématiques usuelles ; en voici quelques exemples choisis :

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

Comme le montre le dernier de ces exemples, certaines expressions mathématiques renvoient des valeurs 'exactes' plutôt que des approximations numériques. Pour obtenir une approximation numérique, on utilise au choix la fonction `n` ou la méthode `n` (chacun de ces noms possède le nom plus long `numerical_approx`, la fonction `N` est identique à `n`). Celles-ci acceptent, en argument optionnel, `prec`, qui indique le nombre de bits de précisions requis, et `digits`, qui indique le nombre de décimales demandées ; par défaut, il y a 53 bits de précision.

```
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 est doté d'un typage dynamique. Ainsi la valeur à laquelle fait référence une variable est toujours associée à un type donné, mais une variable donnée peut contenir des valeurs de plusieurs types Python au sein d'une même portée :

```
sage: a = 5   # a est un entier
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a = 5/3  # a est maintenant un rationnel...
sage: type(a)
<type 'sage.rings.rational.Rational'>
sage: a = 'hello'  # ...et maintenant une chaîne
sage: type(a)
<type 'str'>
```

Le langage de programmation C, qui est statiquement typé, est bien différent : une fois déclarée de type int, une variable ne peut contenir que des int au sein de sa portée.

Un entier Python dont l'écriture commence par un zéro peut susciter des confusions : il est considéré comme un nombre octal, c'est-à-dire un nombre en base huit.

```
sage: 011
9
sage: 8 + 1
9
sage: n = 011
sage: n.str(8)   # écriture en base 8 de n, sous forme de chaîne
'11'
```

Ceci est cohérent avec le langage de programmation C.

2.2 Obtenir de l'aide

Sage est doté d'une importante documentation intégrée, accessible en tapant (par exemple) le nom d'une fonction ou d'une constante suivi d'un point d'interrogation :

```
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)
        0.69314718055994530941723212             # 32-bit
        0.69314718055994530941723212145817656807   # 64-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 dispose aussi de la complétion de ligne de commande, accessible en tapant les quelques premières lettres du nom d'une fonction puis en appuyant sur la touche tabulation. Ainsi, si vous tapez `ta` suivi de `TAB`, Sage affichera `tachyon, tan, tanh, taylor`. C'est une façon commode de voir quels noms de fonctions et d'autres structures sont disponibles en Sage.

2.3 Fonctions, indentation et itération

Les définitions de fonctions en Sage sont introduites par la commande `def`, et la liste des noms des paramètres est suivie de deux points, comme dans :

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

Remarque : suivant la version du *notebook* que vous utilisez, il est possible que vous voyez trois points `...` au début de la deuxième ligne de l'exemple. Ne les entrez pas, ils servent uniquement à signaler que le code est indenté.

Les types des paramètres ne sont pas spécifiés dans la définition de la fonction. Il peut y avoir plusieurs paramètres, chacun accompagné optionnellement d'une valeur par défaut. Par exemple, si la valeur de `divisor` n'est pas donnée lors d'un appel à la fonction ci-dessous, la valeur par défaut `divisor=2` est utilisée.

```
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
```

Il est possible de spécifier un ou plusieurs des paramètres par leur nom lors de l'appel de la fonction ; dans ce cas, les paramètres nommés peuvent apparaître dans n'importe quel ordre :

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

En Python, contrairement à de nombreux autres langages, les blocs de code ne sont pas délimités par des accolades ou des mots-clés de début et de fin de bloc. Au lieu de cela, la structure des blocs est donnée par l'indentation, qui doit être la même dans tout le bloc. Par exemple, le code suivant déclenche une erreur de syntaxe parce que l'instruction `return` n'est pas au même niveau d'indentation que les lignes précédentes.

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

Une fois l'indentation corrigée, l'exemple fonctionne :

```
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]
```

Il n'y a pas besoin de placer des points-virgules en fin de ligne ; une instruction est en général terminée par un passage à la ligne. En revanche, il est possible de placer plusieurs instructions sur la même ligne en les séparant par des points-virgules :

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

Pour continuer une instruction sur la ligne suivante, placez une barre oblique inverse en fin de ligne :

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

Pour compter en Sage, utilisez une boucle dont la variable d'itération parcourt une séquence d'entiers. Par exemple, la première ligne ci-dessous a exactement le même effet que `for(i=0; i<3; i++)` en C++ ou en Java :

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

La première ligne ci-dessous correspond à `for(i=2;i<5;i++)`.

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

Le troisième paramètre contrôle le pas de l'itération. Ainsi, ce qui suit est équivalent à `for(i=1;i<6;i+=2)`.

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

Vous souhaiterez peut-être regrouper dans un joli tableau les résultats numériques que vous aurez calculés avec Sage. Une façon de faire commode utilise les chaînes de format. Ici, nous affichons une table des carrés et des cubes en trois colonnes, chacune d'une largeur de six caractères.

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

La structure de données de base de Sage est la liste, qui est — comme son nom l'indique — une liste d'objets arbitraires. Par exemple, la commande `range` que nous avons utilisée plus haut crée en fait une liste :

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

Voici un exemple plus compliqué de liste :

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

Comme dans de nombreux langages de programmation, les listes sont indexées à partir de 0.

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

La fonction `len(v)` donne la longueur de `v` ; `v.append(obj)` ajoute un nouvel objet à la fin de `v` ; et `del v[i]` supprime l'élément d'indice `i` de `v`.

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

Une autre structure de données importante est le dictionnaire (ou tableau associatif). Un dictionnaire fonctionne comme une liste, à ceci près que les indices peuvent être presque n'importe quels objets (les objets mutables sont interdits) :

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

Vous pouvez définir de nouveaux types de données en utilisant les classes. Encapsuler les objets mathématiques dans des classes représente une technique puissante qui peut vous aider à simplifier et organiser vos programmes Sage. Dans l'exemple suivant, nous définissons une classe qui représente la liste des entiers impairs strictement positifs jusqu'à n. Cette classe dérive du type interne `list`.

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

La méthode `__init__` est appelée à la création de l'objet pour l'initialiser ; la méthode `__repr__` affiche l'objet. À la seconde ligne de la méthode `__init__`, nous appelons le constructeur de la classe `list`. Pour créer un objet de classe `Evens`, nous procédons ensuite comme suit :

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

Notez que `e` s'affiche en utilisant la méthode `__repr__` que nous avons définie plus haut. Pour voir la liste de nombres sous-jacente, on utilise la fonction `list` :

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

Il est aussi possible d'accéder à l'attribut `n`, ou encore d'utiliser `e` en tant que liste.

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

2.4 Algèbre de base et calcul infinitésimal

Sage peut accomplir divers calculs d'algèbre et d'analyse de base : par exemple, trouver les solutions d'équations, dériver, intégrer, calculer des transformées de Laplace. Voir la documentation *Sage Constructions* pour plus d'exemples.

2.4.1 Résolution d'équations

La fonction `solve` résout des équations. Pour l'utiliser, il convient de spécifier d'abord les variables. Les arguments de `solve` sont alors une équation (ou un système d'équations) suivie des variables à résoudre.

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

On peut résoudre une équation en une variable en fonction des autres :

```
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)]
```

On peut également résoudre un système à plusieurs variables :

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

L'exemple suivant, qui utilise Sage pour la résolution d'un système d'équations non-linéaires, a été proposé par Jason Grout. D'abord, on résout le système de façon symbolique :

```
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]]
```

Pour une résolution numérique, on peut utiliser à la place :

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

(La fonction `n` affiche une approximation numérique ; son argument indique le nombre de bits de précision.)

2.4.2 Dérivation, intégration, etc.

Sage est capable de dériver et d'intégrer de nombreuses fonctions. Par exemple, pour dériver $\sin(u)$ par rapport à u, on procède comme suit :

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

Pour calculer la dérivée quatrième de $\sin(x^2)$:

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

Pour calculer la dérivée partielle de $x^2 + 17y^2$ par rapport à x et y respectivement :

```
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
```

Passons aux primitives et intégrales. Pour calculer $\int x\sin(x^2)\,dx$ et $\int_0^1 \frac{x}{x^2+1}\,dx$

```
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)
```

Pour calculer la décomposition en éléments simples de $\frac{1}{x^2-1}$:

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

2.4.3 Résolution des équations différentielles

On peut utiliser Sage pour étudier les équations différentielles ordinaires. Pour résoudre l'équation $x' + x - 1 = 0$:

```
sage: t = var('t')     # on définit une variable t
sage: function('x',t)   # on déclare x fonction de cette variable
x(t)
sage: DE = lambda y: diff(y,t) + y - 1
sage: desolve(DE(x(t)), [x(t),t])
(c + e^t)*e^(-t)
```

Ceci utilise l'interface de Sage vers Maxima [Max], aussi il se peut que la sortie diffère un peu des sorties habituelles de Sage. Dans notre cas, le résultat indique que la solution générale à l'équation différentielle est $x(t) = e^{-t}(e^t + c)$.

Il est aussi possible de calculer des transformées de Laplace. La transformée de Laplace de $t^2e^t - \sin(t)$ s'obtient comme suit :

```
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)
```

Voici un exemple plus élaboré. L'élongation à partir du point d'équilibre de ressorts couplés attachés à gauche à un mur

```
|------\/\/\/\/\---|masse1|----\/\/\/\/\/----|masse2|
        ressort1                 ressort2
```

est modélisée par le système d'équations différentielles d'ordre 2

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

où m_i est la masse de l'objet *i*, x_i est l'élongation à partir du point d'équilibre de la masse *i*, et k_i est la constante de raideur du ressort *i*.

Exemple : Utiliser Sage pour résoudre le problème ci-dessus avec $m_1 = 2$, $m_2 = 1$, $k_1 = 4$, $k_2 = 2$, $x_1(0) = 3$, $x_1'(0) = 0$, $x_2(0) = 3$, $x_2'(0) = 0$.

Solution : Considérons la transformée de Laplace de la première équation (avec les notations $x = x_1$, $y = x_2$) :

```
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),
```

La réponse n'est pas très lisible, mais elle signifie que

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

(où la transformée de Laplace d'une fonction notée par une lettre minuscule telle que $x(t)$ est désignée par la majuscule correspondante $X(s)$). Considérons la transformée de Laplace de la seconde équation :

```
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
```

Ceci signifie

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

Injectons les conditions initiales pour $x(0)$, $x'(0)$, $y(0)$ et $y'(0)$ et résolvons les deux équations qui en résultent :

```
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)]]
```

À présent, prenons la transformée de Laplace inverse pour obtenir la réponse :

```
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)
```

Par conséquent, la solution est

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

On peut en tracer le graphe paramétrique en utilisant

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

Les coordonnées individuelles peuvent être tracées en utilisant

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

(Pour plus d'information sur le tracé de graphe, voir *Graphiques*.)

RÉFÉRENCES : Nagle, Saff, Snider, Fundamentals of Differential Equations, 6th ed, Addison-Wesley, 2004. (see § 5.5).

2.4.4 Méthode d'Euler pour les systèmes d'équations différentielles

Dans l'exemple suivant, nous illustrons la méthode d'Euler pour des équations différentielles ordinaires d'ordre un et deux. Rappelons d'abord le principe de la méthode pour les équations du premier ordre. Etant donné un problème donné avec une valeur initiale sous la forme

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

nous cherchons une valeur approchée de la solution au point $x = b$ avec $b > a$.

Rappelons que par définition de la dérivée

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

où $h > 0$ est fixé et petit. Ceci, combiné à l'équation différentielle, donne $f(x, y(x)) \approx \frac{y(x+h)-y(x)}{h}$. Aussi $y(x+h)$ s'écrit :

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

Si nous notons $hf(x, y(x))$ le « terme de correction » (faute d'un terme plus approprié), et si nous appelons $y(x)$ « l'ancienne valeur de y » et $y(x + h)$ la « nouvelle valeur de y », cette approximation se réécrit

$$y_{nouveau} \approx y_{ancien} + h * f(x, y_{ancien}).$$

Divisions l'intervalle entre a et b en n pas, si bien que $h = \frac{b-a}{n}$. Nous pouvons alors remplir un tableau avec les informations utilisées dans la méthode.

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

Le but est est de remplir tous les trous du tableau, ligne après ligne, jusqu'à atteindre le coefficient « ??? », qui est l'approximation de $y(b)$ au sens de la méthode d'Euler.

L'idée est la même pour les systèmes d'équations différentielles.

Exemple : Rechercher une approximation numérique de $z(t)$ en $t = 1$ en utilisant 4 étapes de la méthode d'Euler, où $z'' + tz' + z = 0$, $z(0) = 1$, $z'(0) = 0$.

Il nous faut réduire l'équation différentielle d'ordre 2 à un système de deux équations différentielles d'ordre 1 (en posant $x = z$, $y = z'$) et appliquer la méthode d'Euler :

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

On en déduit $z(1) \approx 0.75$.

On peut également tracer le graphe des points (x, y) pour obtenir une image approchée de la courbe. La fonction `eulers_method_2x2_plot` réalise cela ; pour l'utiliser, il faut définir les fonctions f et g qui prennent un argument à trois coordonnées : (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)
```

Arrivé à ce point, `P` conserve en mémoire deux graphiques : `P[0]`, le graphe de x en fonction de t, et `P[1]`, le graphique de y par rapport à t. On peut tracer les deux graphiques simultanément par :

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

(Pour plus d'information sur le tracé de graphiques, voir *Graphiques*.)

2.4.5 Fonctions spéciales

Plusieurs familles de polynômes orthogonaux et fonctions spéciales sont implémentées via PARI [GAP] et Maxima [Max]. Ces fonctions sont documentées dans les sections correspondantes (*Orthogonal polynomials* et *Special func-*

tions, respectively) du manuel de référence de Sage (*Sage reference manual*).

```
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")  # les quelques derniers chiffres sont aléatoires
0.167089499251049...
```

Pour l'instant, ces fonctions n'ont été adaptées à Sage que pour une utilisation numérique. Pour faire du calcul formel, il faut utiliser l'interface Maxima directement, comme le présente l'exemple suivant :

```
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 Graphiques

Sage peut produire des graphiques en deux ou trois dimensions.

2.5.1 Graphiques en deux dimensions

En deux dimensions, Sage est capable de tracer des cercles, des droites, des polygones, des graphes de fonctions en coordonnées cartésiennes, des graphes en coordonnées polaires, des lignes de niveau et des représentations de champs de vecteurs. Nous présentons quelques exemples de ces objets ici. Pour plus d'exemples de graphiques avec Sage, on consultera *Résolution des équations différentielles*, *Maxima* et aussi la documentation "Sage Constructions".

La commande suivante produit un cercle jaune de rayon 1 centré à l'origine :

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

Il est également possible de produire un disque plein :

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

Il est aussi possible de créer un cercle en l'affectant à une variable ; ceci ne provoque pas son affichage.

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

Pour l'afficher, on utilise `c.show()` ou `show(c)`, comme suit :

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

Alternativement, l'évaluation de `c.save('filename.png')` enregistre le graphique dans le fichier spécifié.

Toutefois, ces « cercles » ressemblent plus à des ellipses qu'à des cercles, puisque les axes possèdent des échelles différentes. On peut arranger ceci :

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

La commande `show(c, aspect_ratio=1)` produit le même résultat. On peut enregistrer l'image avec cette option par la commande `c.save('filename.png', aspect_ratio=1)`.

Il est très facile de tracer le graphique de fonctions de base :

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

En spécifiant un nom de variable, on peut aussi créer des graphes paramétriques :

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

Différents graphiques peuvent se combiner sur une même image :

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

Une manière commode de tracer des formes pleines est de préparer une liste de points (`L` dans l'exemple ci-dessous) puis d'utiliser la commande `polygon` pour tracer la forme pleine dont le bord est formé par ces points. Par example, voici un deltoïde vert :

```
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: polygon(L, rgbcolor=(1/8,3/4,1/2))
```

Pour visualiser le graphique en masquant les axes, tapez `show(p, axes=false)`.

On peut ajouter un texte à un graphique :

```
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)
```

En cours d'analyse, les professeurs font souvent le dessin suivant au tableau : non pas une mais plusieurs branches de la fonction arcsin, autrement dit, le graphe d'équation $y = \sin(x)$ pour x entre -2π et 2π, renversé par symétrie par rapport à la première bissectrice des axes. La commande Sage suivante réalise cela :

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

Comme les valeurs prises par la fonction tangente ne sont pas bornées, pour utiliser la même astuce pour représenter la fonction arctangente, il faut préciser les bornes de la coordonnée x :

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

Sage sait aussi tracer des graphiques en coordonnées polaires, des lignes de niveau et (pour certains types de fonctions) des champs de vecteurs. Voici un exemple de lignes de niveau :

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

2.5.2 Graphiques en trois dimensions

Sage produit des graphes en trois dimensions en utilisant le package open source appelé [Jmol]. En voici quelques exemples :

Le parapluie de Whitney tracé en jaune http ://en.wikipedia.org/wiki/Whitney_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")
```

Une fois évaluée la commande `parametric_plot3d`, qui affiche le graphique, il est possible de cliquer et de le tirer pour faire pivoter la figure.

Le bonnet croisé (cf. http ://en.wikipedia.org/wiki/Cross-cap ou http ://www.mathcurve.com/surfaces/bonnetcroise/bonnetcroise.shtml) :

```
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")
```

Un tore tordu :

```
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")
```

2.6 Anneaux de base

Nous illustrons la prise en main de quelques anneaux de base avec Sage. Par exemple, `RationalField()` ou `QQ` désigneront dans ce qui suit au corps des nombres rationnels :

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

Le nombre décimal `1.2` est considéré comme un élément de `QQ`, puisqu'il existe une application de coercition entre les réels et les rationnels :

```
sage: 1.2 in QQ
True
```

Néanmoins, il n'y a pas d'application de coercition entre le corps fini à 3 éléments et les rationnels :

```
sage: c = GF(3)(1)    # c est l'élément 1 du corps fini à 3 éléments
sage: c in QQ
False
```

De même, bien entendu, la constante symbolique :math :π n'appartient pas aux rationnels :

```
sage: pi in QQ
False
```

Le symbole `I` représente la racine carrée de -1 ; `i` est synonyme de `I`. Bien entendu, `I` n'appartient pas aux rationnels :

```
sage: i  # i^2 = -1
I
sage: i in QQ
False
```

À ce propos, d'autres anneaux sont prédéfinis en Sage : l'anneau des entiers relatifs `ZZ`, celui des nombres réels `RR` et celui des nombres complexes `CC`. Les anneaux de polynômes sont décrits dans *Polynômes*.

Passons maintenant à quelques éléments d'arithmétique.

```
sage: a, b = 4/3, 2/3
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       # coercition automatique avant addition
0.766666666666667
sage: 0.1 + 2/3       # les règles de coercition sont symétriques en SAGE
0.766666666666667
```

Il y a une subtilité dans la définition des nombres complexes. Comme mentionné ci-dessus, le symbole `i` représente une racine carrée de -1, mais il s'agit d'une racine carrée *formelle* de -1. L'appel `CC(i)` renvoie la racine carrée complexe de -1.

```
sage: i = CC(i)       # nombre complexe en virgule flottante
sage: z = a + b*i
sage: z
1.33333333333333 + 0.666666666666667*I
sage: z.imag()        # partie imaginaire
0.666666666666667
sage: z.real() == a   # coercition automatique avant comparaison
True
sage: QQ(11.1)
111/10
```

2.7 Polynômes

Dans cette partie, nous expliquons comment créer et utiliser des polynômes avec Sage.

2.7.1 Polynômes univariés

Il existe trois façons de créer des anneaux de polynômes.

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

Ceci crée un anneau de polynômes et indique à Sage d'utiliser (la chaîne de caractère) 't' comme indéterminée lors de l'affichage à l'écran. Toutefois, ceci ne définit pas le symbole `t` pour son utilisation dans Sage. Aussi, il n'est pas possible de l'utiliser pour saisir un polynôme (comme $t^2 + 1$) qui appartient à `R`.

Une deuxième manière de procéder est

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

Ceci a les mêmes effets en ce qui concerne `t`.

Une troisième manière de procéder, très pratique, consiste à entrer

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

ou

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

ou même

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

L'effet secondaire de ces dernières instructions est de définir la variable `t` comme l'indéterminée de l'anneau de polynômes. Ceci permet de construire très aisément des éléments de `R`, comme décrit ci-après. (Noter que cette troisième manière est très semblable à la notation par constructeur de Magma et que, de même que dans Magma, ceci peut servir pour une très large classe d'objets.)

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

Quelle que soit la méthode utilisée pour définir l'anneau de polynômes, on récupère l'indéterminée comme le 0-ième générateur :

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

Notez que les nombres complexes peuvent être construits de façon similaire : les nombres complexes peuvent être vus comme engendrés sur les réels par le symbole `i`. Aussi, on dispose de :

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

Pour un anneau de polynômes, on peut obtenir à la fois l'anneau et son générateur ou juste le générateur au moment de la création de l'anneau comme suit :

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

Finalement, on peut faire de l'arithmétique dans $\mathbf{Q}[t]$.

```
sage: R, t = QQ['t'].objgen()
sage: f = 2*t^7 + 3*t^2 - 15/19
sage: f^2
4*t^14 + 12*t^9 - 60/19*t^7 + 9*t^4 - 90/19*t^2 + 225/361
sage: cyclo = R.cyclotomic_polynomial(7); cyclo
t^6 + t^5 + t^4 + t^3 + t^2 + t + 1
sage: g = 7 * cyclo * t^5 * (t^5 + 10*t + 2)
sage: g
7*t^16 + 7*t^15 + 7*t^14 + 7*t^13 + 77*t^12 + 91*t^11 + 91*t^10 + 84*t^9
       + 84*t^8 + 84*t^7 + 84*t^6 + 14*t^5
sage: F = factor(g); F
(7) * t^5 * (t^5 + 10*t + 2) * (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1)
sage: F.unit()
7
sage: list(F)
[(t, 5), (t^5 + 10*t + 2, 1), (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1, 1)]
```

On remarquera que la factorisation prend correctement en compte le coefficient dominant, et ne l'oublie pas dans le résultat.

S'il arrive que vous utilisiez intensivement, par exemple, la fonction `R.cyclotomic_polynomial` dans un projet de recherche quelconque, en plus de citer Sage, vous devriez chercher à quel composant Sage fait appel pour calculer en réalité ce polynôme cyclotomique et citer ce composant. Dans ce cas particulier, en tapant `R.cyclotomic_polynomial??` pour voir le code source, vous verriez rapidement une ligne telle que `f = pari.polcyclo(n)` ce qui signifie que PARI est utilisé pour le calcul du polynôme cyclotomique. Pensez à citer PARI dans votre travail.

La division d'un polynôme par un autre produit un élément du corps des fractions, que Sage crée automatiquement.

```
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
```

En utilisant des séries de Laurent, on peut calculer des développements en série dans le corps des fractions de `QQ[x]` :

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

Si l'on nomme les variables différemment, on obtient un anneau de polynômes univariés différent.

```
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
```

L'anneau est déterminé par sa variable. Notez que créer un autre anneau avec la même variable `x` ne renvoie pas de nouvel anneau.

```
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 permet aussi de travailler dans des anneaux de séries formelles et de séries de Laurent sur un anneau de base quelconque. Dans l'exemple suivant, nous créons un élément de $\mathbf{F}_7[[T]]$ et effectuons une division pour obtenir un élément de $\mathbf{F}_7((T))$.

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

On peut aussi créer des anneaux de séries formelles en utilisant des doubles crochets :

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

2.7.2 Polynômes multivariés

Pour travailler avec des polynômes à plusieurs variables, on commence par déclarer l'anneau des polynômes et les variables, de l'une des deux manières suivantes.

```
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
```

De même que pour les polynômes à une seule variable, les variantes suivantes sont autorisées :

```
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
```

Si l'on désire de simples lettres comme noms de variables, on peut utiliser les raccourcis suivants :

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

A présent, passons aux questions arithmétiques.

```
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
```

On peut aussi utiliser des notations plus mathématiques pour constuire un anneau de polynômes.

```
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,))
```

Sous Sage, les polynômes multivariés sont implémentés en représentation « distributive » (par opposition à récursive), à l'aide de dictionnaires Python. Sage a souvent recours à Singular [Si], par exemple, pour le calcul de pgcd ou de bases de Gröbner d'idéaux.

```
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
```

Créons ensuite l'idéal (f, g) engendré par f et g, en multipliant simplement `(f,g)` par `R` (nous pourrions aussi bien écrire `ideal([f,g])` ou `ideal(f,g)`).

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

En passant, la base de Gröbner ci-dessus n'est pas une liste mais une suite non mutable. Ceci signifie qu'elle possède un univers, un parent, et qu'elle ne peut pas être modifiée (ce qui est une bonne chose puisque changer la base perturberait d'autres routines qui utilisent la base de Gröbner).

```
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.
```

Un peu (comprenez : pas assez à notre goût) d'algèbre commutative est disponible en Sage. Ces routines font appel à Singular. Par exemple, il est possible de calculer la décomposition en facteurs premiers et les idéaux premiers associés de I :

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

2.8 Algèbre linéaire

Sage fournit les constructions standards d'algèbre linéaire, par exemple le polynôme caractéristique, la forme échelonnée, la trace, diverses décompositions, etc. d'une matrice.

La création de matrices et la multiplication matricielle sont très faciles et naturelles :

```
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]
```

Notez bien qu'avec Sage, le noyau d'une matrice A est le « noyau à gauche », c'est-à-dire l'espace des vecteurs w tels que $wA = 0$.

La résolution d'équations matricielles est facile et se fait avec la méthode `solve_right`. L'évaluation de `A.solve_right(Y)` renvoie une matrice (ou un vecteur) X tel que $AX = Y$:

```
sage: Y = vector([0, -4, -1])
sage: X = A.solve_right(Y)
sage: X
(-2, 1, 0)
sage: A * X   #vérifions la réponse...
(0, -4, -1)
```

Un antislash (contre-oblique) `\` peut être employé à la place de `solve_right` : il suffit d'écrire `A \ Y` au lieu de `A.solve_right(Y)`.

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

S'il n'y a aucune solution, Sage renvoie une erreur :

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

De même, il faut utiliser `A.solve_left(Y)` pour résoudre en X l'équation $XA = Y$.

Créons l'espace $\mathrm{Mat}_{3\times 3}(\mathbf{Q})$:

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

(Pour indiquer l'espace des matrices 3 par 4, il faudrait utiliser `MatrixSpace(QQ,3,4)`. Si le nombre de colonnes est omis, il est égal par défaut au nombre de lignes. Ainsi `MatrixSpace(QQ,3)` est un synonyme de `MatrixSpace(QQ,3,3)`.) L'espace des matrices possède une base que Sage enregistre sous forme de liste :

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

Nous créons une matrice comme un élément de `M`.

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

Puis, nous calculons sa forme échelonnée en ligne et son noyau.

```
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]
```

Puis nous illustrons les possibilités de calcul de matrices définies sur des corps finis :

```
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)]
```

Nous créons le sous-espace engendré sur $\mathbf{F}_2$ par les vecteurs lignes ci-dessus.

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

La base de S utilisée par Sage est obtenue à partir des lignes non-nulles de la matrice des générateurs de S réduite sous forme échelonnée en lignes.

2.8.1 Algèbre linéaire creuse

Sage permet de travailler avec des matrices creuses sur des anneaux principaux.

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

L'algorithme multi-modulaire présent dans Sage fonctionne bien pour les matrices carrées (mais moins pour les autres) :

```
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()
```

Notez que Python distingue les majuscules des minuscules :

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

Sage peut calculer des valeurs propres et des vecteurs propres :

```
sage: g = matrix(GF(7), [[5, 1], [4, 1]])
sage: g.eigenvalues()
[4, 2]
sage: g.eigenvectors_right() # renvoie (valeurs propres, [vecteurs propres], multiplicités algébrique
[(4, [
(1, 6)
], 1), (2, [
(1, 4)
], 1)]
```

Les valeurs propres et vecteurs propres peuvent aussi être calculés avec Maxima (voir *Maxima* ci-dessous).

2.9 Groupes finis, groupes abéliens

Sage permet de faire des calculs avec des groupes de permutation, des groupes classiques finis (tels que $SU(n, q)$), des groupes finis de matrices (avec vos propres générateurs) et des groupes abéliens (même infinis). La plupart de ces fonctionalités est implémentée par une interface vers GAP.

Par exemple, pour créer un groupe de permutation, il suffit de donner une liste de générateurs, comme dans l'exemple suivant.

```
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()           # sortie plus ou moins aléatoire (random)
[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()           # sortie aléatoire (random)
(1,5,3)(2,4)
sage: print latex(G)
\langle (3,4), (1,2,3)(4,5) \rangle
```

On peut obtenir la table des caractères (au format LaTeX) à partir de Sage :

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

Sage inclut aussi les groupes classiques ou matriciels définis sur des corps finis :

```
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()  # élément du groupe tiré au hasard (random)
[5 5 5 1]
[0 2 6 3]
[5 0 1 0]
[4 6 3 4]
sage: G.order()
276595200
```

On peut aussi effectuer des calculs dans des groupes abéliens (infinis ou finis) :

```
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
```

2.10 Théorie des nombres

Sage possède des fonctionnalités étendues de théorie des nombres. Par exemple, on peut faire de l'arithmétique dans $\mathbf{Z}/N\mathbf{Z}$ comme suit :

```
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 contient les fonctions standards de théorie des nombres. Par exemple,

```
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
```

Voilà qui est parfait !

La fonction `sigma(n,k)` de Sage additionne les k-ièmes puissances des diviseurs de `n` :

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

Nous illustrons à présent l'algorithme d'Euclide de recherche d'une relation de Bézout, l'indicatrice d'Euler ϕ et le théorème des restes chinois :

```
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
```

Voici une petite expérience concernant la conjecture de Syracuse :

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

Et finalement un exemple d'utilisation du théorème chinois :

```
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,)]
```

2.10.1 Nombres p-adiques

Le corps des nombres p-adiques est implémenté en Sage. Notez qu'une fois qu'un corps p-adique est créé, il n'est plus possible d'en changer la précision.

```
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)
```

Beaucoup de travail a été accompli afin d'implémenter l'anneau des entiers dans des corps p-adiques ou des corps de nombres distincts de $\mathbf{Q}$. Le lecteur intéressé est invité à poser ses questions aux experts sur le groupe Google `sage-support` pour plus de détails.

Un certain nombre de méthodes associées sont d'ores et déjà implémentées dans la classe NumberField.

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


sage: K.galois_group(type="pari")
Galois group PARI group [6, -1, 2, "S3"] of degree 3 of the Number Field
in a with defining polynomial x^3 + x^2 - 2*x + 8


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

2.11 Quelques mathématiques plus avancées

2.11.1 Géométrie algébrique

Il est possible de définir des variétés algébriques arbitraires avec Sage, mais les fonctionnalités non triviales sont parfois limitées aux anneaux sur $\mathbf{Q}$ ou sur les corps finis. Calculons par exemple la réunion de deux courbes planes affines, puis récupérons chaque courbe en tant que composante irréductible de la réunion.

```
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
]
```

Nous pouvons également trouver tous les points d'intersection des deux courbes en les intersectant et en calculant les composantes irréductibles.

```
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
]
```

Ainsi, par exemple, $(1, 0)$ et $(0, 1)$ appartiennent aux deux courbes (ce dont on pouvait directement s'apercevoir) ; il en va de même des points (quadratiques), dont la coordonnée en y satisfait à l'équation $2y^2 + 4y + 3 = 0$.

Sage peut calculer l'idéal torique de la cubique gauche dans l'espace projectif de dimension 3.

```
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
```

2.11.2 Courbes elliptiques

Les fonctionnalités relatives aux courbes elliptiques comprennent la plupart des fonctionnalités de PARI, l'accès aux données des tables en ligne de Cremona (ceci requiert le chargement d'une base de donnée optionnelle), les fonctionnalités de mwrank, c'est-à-dire la 2-descente avec calcul du groupe de Mordell-Weil complet, l'algorithme SEA, le calcul de toutes les isogénies, beaucoup de nouveau code pour les courbes sur $\mathbf{Q}$ et une partie du code de descente algébrique de Denis Simon.

La commande `EllipticCurve` permet de créer une courbe elliptique avec beaucoup de souplesse :
– EllipticCurve([a_1, a_2, a_3, a_4, a_6]) : renvoie la courbe elliptique

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

où les a_i's sont convertis par coercition dans le parent de a_1. Si tous les a_i ont pour parent $\mathbf{Z}$, ils sont convertis par coercition dans $\mathbf{Q}$.
– EllipticCurve([a_4, a_6]) : idem avec $a_1 = a_2 = a_3 = 0$.
– EllipticCurve(label) : Renvoie la courbe elliptique sur $\mathbf{Q}$ de la base de données de Cremona selon son nom dans la (nouvelle !) nomenclature de Cremona. Les courbes sont étiquetées par une chaîne de caractère telle que `"11a"` ou `"37b2"`. La lettre doit être en minuscule (pour faire la différence avec l'ancienne nomenclature).
– EllipticCurve(j) : renvoie une courbe elliptique de j-invariant j.
– EllipticCurve(R, [a_1, a_2, a_3, a_4, a_6]) : Crée la courbe elliptique sur l'anneau R donnée par les coefficients a_i comme ci-dessus.
Illustrons chacune de ces constructions :

```
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
```

Le couple $(0, 0)$ est un point de la courbe elliptique E définie par $y^2 + y = x^3 - x$. Pour créer ce point avec Sage, il convient de taper `E([0,0])`. Sage peut additionner des points sur une telle courbe elliptique (rappelons qu'une courbe elliptique possède une structure de groupe additif où le point à l'infini représente l'élément neutre et où trois points alignés de la courbe sont de somme nulle) :

```
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
```

Les courbes elliptiques sur les nombres complexes sont paramétrées par leur j-invariant. Sage calcule le j-invariant comme suit :

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

Si l'on fabrique une courbe avec le même j-invariant que celui de E, elle n'est pas nécessairement isomorphe à E. Dans l'exemple suivant, les courbes ne sont pas isomorphes parce que leur conducteur est différent.

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

Toutefois, le twist de F par 2 donne une courbe isomorphe.

```
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
```

On peut calculer les coefficients a_n de la série-L ou forme modulaire $\sum_{n=0}^{\infty} a_n q^n$ attachée à une courbe elliptique. Le calcul s'effectue en utilisant la bibliothèque PARI écrite en C :

```
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)
```

Il faut à peine quelques secondes pour calculer tous les coefficients a_n pour $n \leq 10^5$:

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

Les courbes elliptiques peuvent être construites en utilisant leur nom dans la nomenclature de Cremona. Ceci charge par avance la courbe elliptique avec les informations la concernant, telles que son rang, son nombre de Tamagawa, son régulateur, 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
```

On peut aussi accéder à la base de données de Cremona directement.

```
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]}
```

Les objets extraits de la base de données ne sont pas de type `EllipticCurve`, mais de simples entrées de base de données formées de quelques champs. Par défaut, Sage est distribué avec une version réduite de la base de données de Cremona qui ne contient que des informations limitées sur les courbes elliptiques de conducteur ≤ 10000. Il existe également en option une version plus complète qui contient des données étendues portant sur toute les courbes de conducteur jusqu'à 120000 (à la date d'octobre 2005). Une autre - énorme (2GB) - base de données optionnelle, fournie dans un package séparé, contient des centaines de millions de courbes elliptiques de la bases de donnée de Stein-Watkins.

2.11.3 Caractères de Dirichlet

Un *caractère de Dirichlet* est une extension d'un homomorphisme $(\mathbf{Z}/N\mathbf{Z})^* \to R^*$, pour un certain anneau R, à l'application $\mathbf{Z} \to R$ obtenue en envoyant les entiers x tels que $\gcd(N, x) > 1$ vers 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
```

Une fois le groupe créé, on crée aussitôt un élément et on calcule avec lui.

```
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
```

Il est possible aussi de calculer l'action d'un groupe de Galois $\mathrm{Gal}(\mathbf{Q}(\zeta_N)/\mathbf{Q})$ sur l'un de ces caractères, de même qu'une décomposition en produit direct correspondant à la factorisation du module.

```
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
]
```

Construisons à present le groupe de caractères de Dirichlet modulo 20, mais à valeur dans $\mathbf{Q}(i)$:

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

Nous calculons ensuite différents invariants de `G` :

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

Dans cet exemple, nous créons un caractère de Dirichlet à valeurs dans un corps de nombres. Nous spécifions ci-dessous explicitement le choix de la racine de l'unité par le troisième argument de la fonction `DirichletGroup`.

```
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]]
```

Ici, `NumberField(x^4 + 1, 'a')` indique à Sage d'utiliser le symbole "a" dans l'affichage de ce qu'est `K` (un corps de nombre en "a" défini par le polynôme $x^4 + 1$). Le nom "a" n'est pas déclaré à ce point. Une fois que `a = K.0` (ou de manière équivalente `a = K.gen()`) est évalué, le symbole "a" représente une racine du polynôme générateur $x^4 + 1$.

2.11.4 Formes modulaires

Sage peut accomplir des calculs relatifs aux formes modulaires, notamment des calculs de dimension, d'espace de symboles modulaires, d'opérateurs de Hecke et de décomposition.

Il y a plusieurs fonctions disponibles pour calculer la dimension d'espaces de formes modulaires. Par exemple,

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

Nous illustrons ci-dessous le calcul des opérateurs de Hecke sur un espace de symboles modulaires de niveau 1 et de poids 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
```

Nous pouvons aussi créer des espaces pour $\Gamma_0(N)$ et $\Gamma_1(N)$.

```
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
```

Calculons quelques polynômes caractéristiques et développements en série de Fourier.

```
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)
]
```

On peut même calculer des espaces de formes modulaires avec caractères.

```
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)
]
```

Voici un autre exemple montrant comment Sage peut calculer l'action d'un opérateur de Hecke sur un espace de formes modulaires.

```
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
```

Notons T_p les opérateurs de Hecke usuels (p premier). Comment agissent les opérateurs de Hecke T_2, T_3, T_5 sur l'espace des symboles modulaires ?

```
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]
```

CHAPITRE 3

La ligne de commande interactive

Dans la plus grande partie de ce tutoriel, nous supposons que vous avez lancé l'interpréteur Sage avec la commande `sage`. Cela démarre une version adaptée du shell (interpréteur de commandes) IPython et importe un grand nombre de fonctions et de classes qui sont ainsi prêtes à l'emploi depuis l'invite de commande. D'autres personnalisations sont possibles en éditant le fichier `$SAGE_ROOT/ipythonrc`. Au démarrage, le shell Sage affiche un message de ce genre :

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


sage:
```

Pour quitter Sage, tapez Ctrl-D ou tapez `quit` ou `exit`.

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

L'indication *wall time* donne le temps écoulé à votre montre (ou l'horloge suspendue au mur) pendant l'exécution. C'est une donnée pertinente car le temps processeur (*CPU time*) ne tient pas compte du temps utilisé par les sous-processus comme GAP et Singular.

(Il vaut mieux éviter de tuer un processus Sage depuis un terminal avec `kill -9`, car il est possible que Sage ne tue pas ses processus enfants, par exemple des processus Maple qu'il aurait lancés, ou encore qu'il ne nettoie pas les fichiers temporaires de `$HOME/.sage/tmp`.)

3.1 Votre session Sage

Une session est la suite des entrées et sorties qui interviennent entre le moment où vous démarrez Sage et celui où vous le quittez. Sage enregistre un journal de toutes les entrées via IPython. Si vous utilisez le shell interactif (par opposition à l'interface *notebook*), vous pouvez taper `%hist` à n'importe quel moment pour obtenir la liste de toutes les lignes de commandes entrées depuis le début de la session. Tapez `?` à l'invite de commande Sage pour plus d'informations sur IPython. Par exemple : « IPython fournit des invites de commande numérotées [...] avec un cache des entrées-sorties. Toutes les entrées sont sauvegardées et peuvent être rappelées comme variables (en plus de la navigation habituelle

dans l'historique avec les flèches du clavier). Les variables GLOBALES suivantes existent toujours (ne les écrasez pas !) » :

```
_ : dernière entrée (shell et notebook)
__ : avant-dernière entrée (shell uniquement)
_oh : liste de toutes les entrées précédentes (shell uniquement)
```

Voici un exemple :

```
sage: factor(100)
 _1 = 2^2 * 5^2
sage: kronecker_symbol(3,5)
 _2 = -1
sage: %hist   #Fonctionne depuis le shell mais pas depuis le 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
```

Dans la suite de ce tutorial et le reste de la documentation de Sage, nous omettrons la numérotation des sorties.

Il est possible de créer (pour la durée d'une session) une macro qui rappelle une liste de plusieurs lignes d'entrée.

```
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
```

Depuis le shell interactif Sage, il est possible d'exécuter une commande Unix en la faisant précéder d'un point d'exclamation !. Par exemple,

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

renvoie la liste des fichiers du répertoire courant.

Dans ce contexte, le `PATH` commence par le répertoire des binaires de Sage, de sorte que les commandes `gp`, `gap`, `singular`, `maxima`, etc. appellent les versions inclues dans Sage.

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

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

3.2 Journal des entrées-sorties

Enregistrer le journal d'une session Sage n'est pas la même chose que sauvegarder la session (voir *Enregister et recharger des sessions entières* pour cette possibilité). Pour tenir un journal des entrées (et optionnellement des sorties) de Sage, utilisez la commande `logstart`. Tapez `logsatart ?` pour plus d'informations. Cette commande permet d'enregistrer toutes les entrées que vous tapez, toutes les sorties, et de rejouer ces entrées dans une session future (en rechargeant le fichier journal).

```
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)]
```

Si vous utilisez le terminal Konsole de KDE, vous pouvez aussi sauver votre session comme suit : après avoir lancé Sage dans la `konsole`, ouvrez le menu « Configuration » et choisissez « Historique... » puis comme nombre de lignes « Illimité ». Ensuite, lorsque vous souhaitez enregistrer l'état de votre session, sélectionnez « Enregistrer l'historique sous... » dans le menu « Édition » et entrez le nom d'un fichier où enregistrer le texte de votre session. Une fois le fichier sauvegardé, vous pouvez par exemple l'ouvrir dans un éditeur comme xemacs et l'imprimer.

3.3 Coller du texte ignore les invites

Imaginons que vous lisiez une session Sage ou Python et que vous vouliez copier-coller les calculs dans Sage. Le problème est qu'il y a des invites `>>>` ou `sage :` en plus des entrées. En fait, vous pouvez tout à fait copier un exemple complet, invites comprises : par défaut, l'analyseur syntaxique de Sage supprime les `>>>` et `sage :` en début de ligne avant de passer la ligne à Python. Par exemple, les lignes suivantes sont interprétées correctement :

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

3.4 Mesure du temps d'exécution d'une commande

Si une ligne d'entrée commence par `%time`, le temps d'exécution de la commande correspondante est affiché après la sortie. Nous pouvons par exemple comparer le temps que prend le calcul d'une certaine puissance entière par diverses méthodes. Les temps de calcul ci-dessous seront sans doute très différents suivant l'ordinateur, voire la version de Sage utilisés. Premièrement, en pur Python :

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

Le calcul a pris 0.66 seconde, pendant un intervalle de *wall time* (le temps de votre montre) lui aussi de 0.66 seconde. Si d'autres programmes qui s'exécutent en même temps que Sage chargent l'ordinateur avec de gros calculs, le *wall time* peut être nettement plus important que le temps processeur.

Chronométrons maintenant le calcul de la même puissance avec le type Integer de Sage, qui est implémenté (en Cython) en utilisant la bibliothèque GMP :

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

Avec l'interface à la bibliothèque C PARI :

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

GMP est plus rapide, mais de peu (ce n'est pas une surprise, car la version de PARI incluse dans Sage utilise GMP pour l'arithmétique entière).

Il est aussi possible de chronométrer tout un bloc de commandes avec la commande `cputime`, comme dans l'exemple suivant :

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

La commande `walltime` fonctionne comme `cputime`, à ceci près qu'elle mesure le temps total écoulé « à la montre ».

Nous pouvons aussi faire faire le calcul de puissance ci-dessus à chacun des systèmes de calcul formel inclus dans Sage. Dans chaque cas, nous commençons par lancer une commande triviale dans le système en question, de façon à démarrer son serveur. La mesure la plus pertinente est le *wall time*. Cependant, si la différence entre celui-ci et le temps processeur est importante, cela peut indiquer un problème de performance qui mérite d'être examiné.

```
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
```

Nous voyons que GAP et Maxima sont les plus lents sur ce test (lancé sur la machine `sage.math.washington.edu`). Mais en raison du surcoût de l'interface pexpect, la comparaison avec Sage, qui est le plus rapide, n'est pas vraiment équitable.

3.5 Erreurs et exceptions

Quand quelque chose ne marche pas, cela se manifeste habituellement par une « exception » Python. Python essaie de plus de donner une idée de ce qui a pu déclencher l'exception. Bien souvent, il affiche le nom de l'exception (par exemple `NameError` ou `ValueError`, voir le manuel de référence de Python [Py] pour une liste complète). Par exemple :

```
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
```

Le débogueur interactif est parfois utile pour comprendre ce qu'il s'est passé. Il s'active ou se désactive avec `%pdb` (et est désactivé par défaut). L'invite `ipdb>>` du débogueur apparaît si une exception a lieu alors que celui-ci est actif. Le débogueur permet d'afficher l'état de n'importe quelle variable locale et de monter ou descendre dans la pile d'exécution. Par exemple :

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

ipdb>
```

Pour obtenir une liste des commandes disponibles dans le débogueur, tapez `?` à l'invite `ipdb>` :

```
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
```

Tapez Ctrl-D ou `quit` pour revenir à Sage.

3.6 Recherche en arrière et complétion de ligne de commande

Commençons par créer l'espace vectoriel de dimension trois $V = \mathbf{Q}^3$ comme suit :

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

Nous pouvons aussi utiliser la variante plus concise :

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

Tapez ensuite le début d'une commande, puis `Ctrl-p` (ou flèche vers le haut) pour passer en revue les lignes qui commencent par les mêmes lettres parmi celles que vous avez entrées jusque-là. Cela fonctionne même si vous avez quitté et relancé Sage entre-temps. Vous pouvez aussi rechercher une portion de commande en remontant dans l'historique avec `Ctrl-r`. Toutes ces fonctionnalités reposent sur la bibliothèque `readline`, qui existe pour la plupart des variantes de Linux.

La complétion de ligne de commande permet d'obtenir facilement la liste des fonctions membres de V : tapez simplement `V.` puis appuyez sur la touche tabulation.

```
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
```

Si vous tapez les quelques premières lettres d'un nom de fonction avant d'appuyer sur `tab`, vous n'obtiendrez que les fonctions qui commencent par ces quelques lettres :

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

Si vous cherchez à savoir ce que fait une fonction, par exemple la fonction coordinates, `V.coordinates ?` affiche un message d'aide et `V.coordinates ? ?` le code source de la fonction, comme expliqué dans la section suivante.

3.7 Aide en ligne

Sage dispose d'un système d'aide intégré. Pour obtenir la documentation d'une fonction, tapez son nom suivi d'un point d'interrogation.

```
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]
```

Comme nous pouvons le voir ci-dessus, la sortie indique le type de l'objet, le nom du fichier où il est défini, et donne une description de l'effet de la fonction, avec des exemples que vous pouvez copier dans votre session Sage. Pratiquement tous ces exemples sont automatiquement testés régulièrement pour s'assurer qu'ils se comportent exactement comme indiqué.

Une autre fonctionnalité, nettement dans l'esprit du caractère ouvert de Sage, est que lorsque `f` est une fonction Python, taper `f ? ?` affiche son code source. Par exemple,

```
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()
```

Nous voyons que la fonction `coordinates` ne fait qu'appeler `coordinate_vector` et transformer le résultat en une liste. Mais alors, que fait la fonction `coordinate_vector` ?

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

La fonction `coordinate_vector` convertit son entrée en un élément de l'espace ambiant, ce qui a pour effet de calculer le vecteur des coefficients de v dans V. L'espace V est déjà « l'espace ambiant » puisque c'est simplement $\mathbf{Q}^3$. Il y a aussi une fonction `coordinate_vector` différente pour les sous-espaces. Créons un sous-espace et examinons-là :

```
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)
```

(Si vous pensez que cette implémentation est inefficace, venez nous aider à optimiser l'algèbre linéaire !)

Vous pouvez aussi taper `help(commande)` ou `help(classe)` pour appeler une sorte de page de manuel relative à une commande ou une classe.

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

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

Pour quitter la page d'aide, appuyez sur `q`. Votre session revient à l'écran comme elle était : contrairement à la sortie de `fonction?`, celle de `help` n'encombre pas votre session. Une possibilité particulièrement utile est de consulter l'aide d'un module entier avec `help(nom_du_module`. Par exemple, les espaces vectoriels sont définis dans `sage.modules.free_module`, et on accède à la documentation de ce module en tapant `help(sage.modules.free_module)`. Lorsque vous lisez une page de documentation avec la commande `help`, vous pouvez faire des recherches en avant en tapant `/` et en arrière en tapant `?`.

3.8 Enregistrer et charger des objets individuellement

Imaginons que nous calculions une matrice, ou pire, un espace compliqué de symboles modulaires, et que nous souhaitions les sauvegarder pour un usage futur. Les systèmes de calcul formel ont différentes approches pour permettre cela.

1. **Sauver la partie :** il n'est possible de sauver que la session entière (p.ex. GAP, Magma).
2. **Format d'entrée/sortie unifié :** chaque objet est affiché sous une forme qui peut être relue (GP/PARI).
3. **Eval :** permettre d'évaluer facilement du code arbitraire dans l'interpréteur (p.ex. Singular, PARI).

Utilisant Python, Sage adopte une approche différente, à savoir que tous les objets peuvent être sérialisés, i.e. transformés en chaînes de caractères à partir desquelles ils peuvent être reconstruits. C'est une méthode semblable dans l'esprit à l'unification des entrées et sorties de PARI, avec l'avantage que l'affichage normal des objets n'a pas besoin d'être trop compliqué. En outre, cette fonction de sauvegarde et de relecture des objets ne nécessite (dans la plupart des cas) aucune programmation supplémentaire : il s'agit simplement une fonctionnalité de Python fournie par le langage depuis la base.

Quasiment n'importe quel objet Sage `x` peut être enregistré sur le disque, dans un format compressé, avec `save(x, nom_de_fichier)` (ou dans bien des cas `x.save(nom_de_fichier)`). Pour recharger les objets, on utilise `load(nom_de_fichier)`.

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

Quittez puis redémarrez maintenant Sage. Vous pouvez récupérer `A` :

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

Vous pouvez faire de même avec des objets plus compliqués, par exemple des courbes elliptiques. Toute l'information en cache sur l'objet est stockée avec celui-ci :

```
sage: E = EllipticCurve('11a')
sage: v = E.anlist(100000)               # prend un moment
sage: save(E, 'E')
sage: quit
```

Ainsi, la version sauvegardée de `E` prend 153 kilo-octets car elle contient les 100000 premiers 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)               # instantané !
```

(En Python, les sauvegardes et rechargements s'effectuent à l'aide du module `cPickle`. En particulier, on peut sauver un objet Sage `x` par la commande `cPickle.dumps(x, 2)`. Attention au `2` !)

Sage n'est pas capable de sauvegarder les objets créés dans d'autres systèmes de calcul formel comme GAP, Singular, Maxima etc. : au rechargement, ils sont dans un état marqué « invalide ». Concernant GAP, un certain nombre d'objets sont affichés sous une forme qui permet de les reconstruire, mais d'autres non, aussi la reconstruction d'objets GAP à partir de leur affichage est intentionnellement interdite.

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

```
...
ValueError: The session in which this object was defined is no longer
running.
```

Les objets GP/PARI, en revanche, peuvent être sauvegardés et rechargés, puisque la forme imprimée d'un objet suffit à reconstruire celui-ci.

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

Un objet sauvegardé peut être rechargé y compris sur un ordinateur doté d'une architecture ou d'un système d'exploitation différent. Ainsi, il est possible de sauvegarder une immense matrice sur un OS-X 32 bits, la recharger sur un Linux 64 bits, l'y mettre en forme échelon et rapatrier le résultat. Bien souvent, un objet peut même être rechargé avec une version de Sage différente de celle utilisée pour le sauver, pourvu que le code qui gère cet objet n'ait pas trop changé d'une version sur l'autre. Sauver un objet enregistre tous ses attributs ainsi que la classe à laquelle il appartient (mais pas son code source). Si cette classe n'existe plus dans une version ultérieure de Sage, l'objet ne peut pas y être rechargé. Mais il demeure possible de le charger dans l'ancienne version pour récupérer son dictionnaire (avec `x.__dict__`), sauver celui-ci, et le recharger dans la nouvelle version.

3.8.1 Enregistrer un objet comme texte

Une autre possibilité consiste à sauvegarder la représentation texte ASCII dans un fichier texte brut, ce qui se fait simplement en ouvrant le fichier en écriture et en y écrivant la représentation de l'objet (il est tout à fait possible d'écrire plusieurs objets). Une fois l'écriture terminée, nous refermons le fichier.

```
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 Enregister et recharger des sessions entières

Sage dispose de fonctions très souples de sauvegarde et relecture de sessions entières.

La commande `save_session(nom_de_session)` enregistre toutes les variables définies dans la session courante sous forme de dictionnaire dans le fichier `nom_de_session.sobj`. (Les éventuelles variables qui ne supportent pas la sauvegarde sont ignorées.) Le fichier `.sobj` obtenu peut être rechargé comme n'importe quel objet sauvegardé ; on obtient en le rechargeant un dictionnaire dont les clés sont les noms de variables et les valeurs les objets correspondants.

La commande `reload_session(nom_de_session)` charge toutes les variables sauvées dans `nom_de_session`. Cela n'efface pas les variables déjà définies dans la session courante : les deux sessions sont fusionnées.

Commençons par démarrer Sage et par définir quelques variables.

```
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
```

Nous sauvons maintenant notre session, ce qui a pour effet d'enregistrer dans un même fichier toutes les variables ci-dessus. Nous pouvons constater que le fichier fait environ 3 ko.

```
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
```

Enfin, nous redémarrons Sage, nous définissons une nouvelle variable, et nous rechargeons la session précédente.

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

Toutes les variables sauvegardées sont à nouveau disponibles. En outre, la variable `b` n'a pas été écrasée.

```
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 L'interface *notebook*

Pour démarrer le *notebook* Sage, tapez

```
sage: notebook()
```

sur la ligne de commande Sage. Cela démarre le serveur du *notebook* et ouvre votre navigateur web par défaut sur la page correspondante. Les fichiers d'état du serveur sont placés dans `$HOME/.sage/sage\_notebook`.

La variante

```
sage: notebook("repertoire")
```

lance un nouveau serveur *notebook* en utilisant les fichiers du répertoire donné à la place de `$HOME/.sage/sage_notebook`. Cela peut être utile si vous voulez gérer une collection de feuilles de travail attachées à un projet spécifique, ou encore lancer plusieurs instances du serveur en même temps.

Au démarrage, le *notebook* commence par créer les fichiers suivants dans `$HOME/.sage/sage_notebook` :

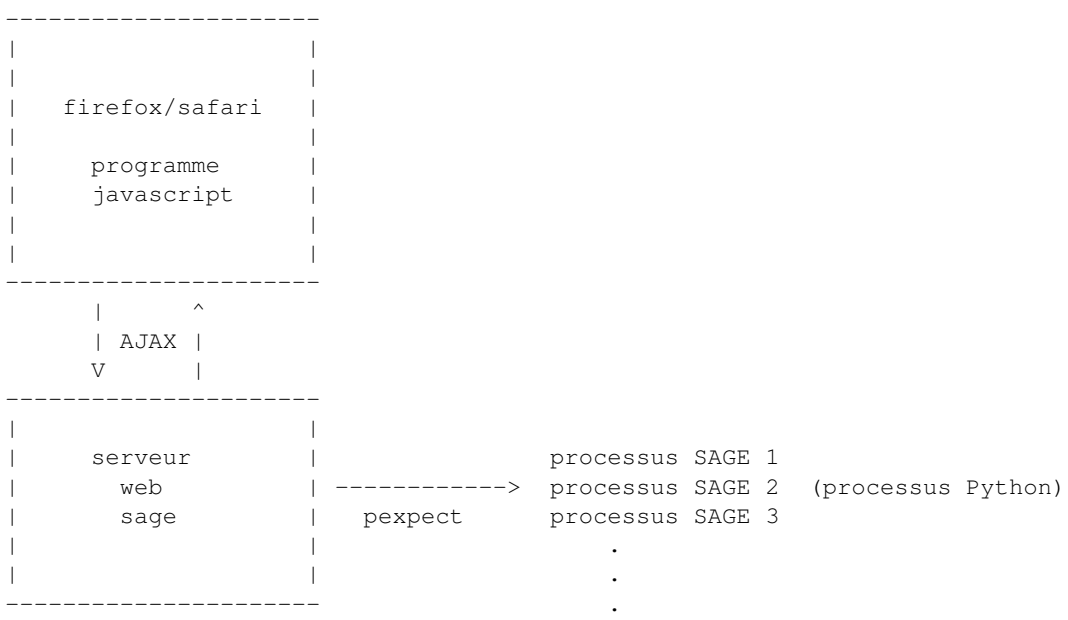
```
nb.sobj       (fichier objet Sage du notebook)
objects/      (sous-répertoire contenant les objets Sage)
worksheets/   (sous-répertoire contenant les feuilles de travail).
```

Une fois ces fichiers créés, le *notebook* démarre un serveur web.

Un « *notebook* » est une collection de comptes utilisateur, qui peuvent chacun posséder un nombre quelconque de feuilles de travail. Quand vous créez une nouvelle feuille de travail, les données correspondantes sont stockées dans un répertoire de la forme `worksheets/utilisateur/numéro`. Dans chacun de ces répertoires se trouve un fichier texte brut `worksheet.txt` qui contient tout ce qu'il faut pour reconstituer la feuille de travail s'il lui arrive quelque chose, si Sage rencontre un problème, ou quoi que ce soit de ce genre.

Dans Sage, vous pouvez taper `notebook ?` pour beaucoup plus d'informations sur comment démarrer un serveur.

Le schéma suivant présente l'architecture du *Notebook* Sage :



Dans le *notebook*, pour consulter l'aide d'une commande Sage `cmd`, tapez `cmd ?` dans le champ d'entrée des commandes puis tapez `<échap>` (et non `<maj-entrée>`).

CHAPITRE 4

Interfaces

L'un des aspects essentiels de Sage est qu'il permet d'effectuer des calculs utilisant des objets issus de nombreux systèmes de calcul formel de façon unifiée, avec une interface commune et un langage de programmation sain.

Les méthodes `console` et `interact` d'une interface avec un programme externe font des choses tout-à-fait différentes. Prenons l'exemple de GAP :

1. `gap.console()` : Cette commande ouvre la console GAP. Cela transfère le contrôle à GAP ; Sage n'est dans ce cas qu'un moyen commode de lancer des programmes, un peu comme le shell sous Unix.
2. `gap.interact()` : Cette commande permet d'interagir avec une instance de GAP en cours d'exécution, et éventuellement « remplie d'objets Sage ». Il est possible d'importer des objets Sage dans la session GAP (y compris depuis l'interface interactive), etc.

4.1 GP/PARI

PARI est un programme C compact, mature, fortement optimisé et spécialisé en théorie des nombres. Il possède deux interfaces très différentes utilisables depuis Sage :
– `gp` - l'interpréteur "**G** o **P** ARI", et
– `pari` - la bibliothèque C PARI.
Ainsi, les deux commandes suivantes font le même calcul de deux façons différentes. Les deux sorties ont l'air identiques, mais elles ne le sont pas en réalité, et ce qui ce passe en coulisses est radicalement différent.

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

Dans le premier exemple, on démarre une instance de l'interpréteur GP et on lui envoie la chaîne `'znprimroot(10007)'`. Il l'évalue, et affecte le résultat à une variable GP (ce qui occupe un espace qui ne sera pas libéré dans la mémoire du processus fils GP). La valeur de la variable est ensuite affichée. Dans le second cas, nul programme séparé n'est démarré, la chaîne `'znprimroot(10007)'` est évaluée par une certaine fonction de la bibliothèque C PARI. Le résultat est stocké sur le tas de l'interpréteur Python, et la zone de mémoire utilisée est libérée lorsque son contenu n'est plus utilisé. Les objets renvoyés par ces deux commandes sont de types différents :

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

Alors, laquelle des intrefaces utiliser ? Tout dépend de ce que vous cherchez à faire. L'interface GP permet de faire absolument tout ce que vous pourriez faire avec la ligne de commande GP/PARI habituelle, puisqu'elle fait appel à celle-ci. En particulier, vous pouvez l'utiliser pour charger et exécuter des scripts PARI compliqués. L'interface PARI (via la bibliothèque C) est nettement plus restrictive. Tout d'abord, toutes les méthodes ne sont pas implémentées. Deuxièmement, beaucoup de code utilisant par exemple l'intégration numérique ne fonctionne pas via l'interface PARI. Ceci dit, l'interface PARI est souvent considérablement plus rapide et robuste que l'interface GP.

(Si l'interface GP manque de mémoire pour évaluer une ligne d'entrée donnée, elle double silencieusement la taille de la pile et réessaie d'évaluer la ligne. Ainsi votre calcul ne plantera pas même si vous n'avez pas évalué convenablement l'espace qui nécessite. C'est une caractéristique commode que l'interpréteur GP habituel ne semble pas fournir. L'interface PARI, quant à elle, déplace immédiatement les objets créés en-dehors de la pile de PARI, de sorte que celle-ci ne grossit pas. Cependant, la taille de chaque objet est limitée à 100 Mo, sous peine que la pile ne déborde à la création de l'objet. Par ailleurs, cette copie supplémentaire a un léger impact sur les performances.)

En résumé, Sage fait appel à la bibliothèque C PARI pour fournir des fonctionnalités similaires à celle de l'interpréteur PARI/GP, mais depuis le langage Python, et avec un gestionnaire de mémoire plus perfectionné.

Commençons par créer une liste PARI à partir d'une liste Python.

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

En Sage, les objets PARI sont de type `py_pari.gen`. Le type PARI de l'objet sous-jacent est donné par la méthode `type`.

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

Pour créer une courbe elliptique en PARI, on utiliserait `ellinit([1,2,3,4,5])`. La syntaxe Sage est semblable, à ceci près que `ellinit` devient une méthode qui peut être appelée sur n'importe quel objet PARI, par exemle notre `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]
```

À présent que nous disposons d'une courbe elliptique, faisons quelques calculs avec.

```
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

Pour les mathématiques discrètes effectives et principalement la théorie des groupes, Sage utilise GAP 4.4.10.

Voici un exemple d'utilisation de la fonction GAP `IdGroup`, qui nécessite une base de données optionnelle de groupes de petit ordre, à installer séparément comme décrit plus bas.

```
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()     # nécessite le paquet facultatif database_gap (optional)
[ 120, 34 ]
sage: G.Order()
120
```

On peut faire le même calcul en SAGE sans invoquer explicitement l'interface GAP comme suit :

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

(Certaines fonctionnalités de GAP nécessitent l'installation de deux paquets facultatifs. Saisissez `sage -optional` pour consulter la liste des paquets facultatifs, et choisissez celui dont le nom ressemble à `gap\_packages-x.y.z`, puis installez-le par `sage -i gap\_packages-x.y.z`. Faites de même avec `database\_gap-x.y.z`. D'autres paquets GAP, non couverts par la licence GPL, peuvent être téléchargés depuis le site web de GAP [GAPkg] et installés en les désarchivant dans `$SAGE_ROOT/local/lib/gap-4.4.10/pkg`.)

4.3 Singular

Singular fournit une bibliothèque consistante et mature qui permet, entre autres, de calculer des pgcd de polynômes de plusieurs variables, des factorisations, des bases de Gröbner ou encore des bases d'espaces de Riemann-Roch de courbes planes. Considérons la factorisation de polynômes de plusieurs variables à l'aide de l'interface à Singular fournie par Sage (n'entrez pas les `...`) :

```
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')
```

Maintenant que nous avons défini f, affichons-le puis factorisons-le.

```
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^
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
```

Comme avec GAP dans la section *GAP*, nous pouvons aussi calculer la factorisation sans utiliser explicitement l'interface Singular (Sage y fera tout de même appel en coulisses pour le calcul).

```
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

Le système de calcul formel Maxima est fourni avec Sage accompagné de clisp, une version du langage Lisp, et d'openmath, un programme de tracé de courbes en Tcl/Tk utilisé par Maxima. En revanche, gnuplot (que Maxima utilise par défaut pour tracer des graphiques) n'est distribué que comme paquet optionnel de Sage. Maxima fournit notamment des routines de calcul sur des expressions formelles. Il permet de calculer des dérivées, primitives et intégrales, de résoudre des équations différentielles d'ordre 1 et souvent d'ordre 2, et de résoudre par transformée de Laplace les équations différentielles linéaires d'ordre quelconque. Maxima dispose aussi d'un grand nombre de fonctions spéciales, permet de tracer des graphes de fonctions via gnuplot, et de manipuler des matrices (réduction en lignes, valeurs propres, vecteurs propres...) ou encore des équations polynomiales.

Utilisons par exemple l'interface Sage/Maxima pour construire la matrice dont le coefficient d'indice i, j vaut i/j, pour $i, j = 1, \ldots, 4$.

```
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]]
```

Un deuxième exemple :

```
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]))); lambda1 = eigA[0][0][0]
sage: v2 = V(sage_eval(repr(eigA[2]))); lambda2 = eigA[0][0][1]
sage: v3 = V(sage_eval(repr(eigA[3]))); 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
```

Voici enfin quelques exemples de tracés de graphiques avec `openmath` depuis Sage. Un grand nombre de ces exemples sont des adaptations de ceux du manuel de référence de Maxima.

Tracé en 2D de plusieurs fonctions (n'entrez pas les `...`) :

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

Un graphique 3D interactif, que vous pouvez déplacer à la souris (n'entrez pas les `...`) :

```
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
```

Le célèbre ruban de Möbius (n'entrez pas les `...`) :

```
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
```

Et la fameuse bouteille de Klein (n'entrez pas les `...`) :

```
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
```

CHAPITRE 5

Programmation

5.1 Charger et attacher des fichiers Sage

Nous décrivons maintenant la manière de charger dans Sage des programmes écrits dans des fichiers séparés. Créons un fichier appelé `example.sage` avec le contenu suivant :

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

Nous pouvons lire et exécuter le contenu du fichier `example.sage` en utilisant la commande `load`.

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

Nous pouvons aussi attacher un fichier Sage à la session en cours avec la commande `attach` :

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

L'effet de cette commande est que si nous modifions maintenant `example.sage` et entrons une ligne vierge (i.e. appuyons sur `entrée`), le contenu de `example.sage` sera automatiquement rechargé dans Sage.

Avec `attach`, le fichier est rechargé automatiquement dans Sage à chaque modification, ce qui est pratique pour déboguer du code ; tandis qu'avec `load` il n'est chargé qu'une fois.

Lorsque Sage lit `example.sage`, il le convertit en un programme Python qui est ensuite exécuté par l'interpréteur Python. Cette conversion est minimale, elle se résume essentiellement à encapsuler les littéraux entiers dans des `Integer()` et les littéraux flottants dans des `RealNumber()`, à remplacer les `^` par des `**`, et à remplacer par exemple `R.2` par `R.gen(2)`. La version convertie en Python de `example.sage` est placée dans le même répertoire sous le nom `example.sage.py`. Elle contient le code suivant :

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

On voit que les littéraux entiers ont été encapsulés et que le `^` a été remplacé par `**` (en effet, en Python, `^` représente le ou exclusif et `**` l'exponentiation).

(Ce prétraitement est implémenté dans le fichier `sage/misc/interpreter.py`.)

On peut coller dans Sage des blocs de code de plusieurs lignes avec des indentations pourvu que les blocs soient délimités par des retours à la ligne (cela n'est pas nécessaire quand le code est dans un fichier). Cependant, la meilleure façon d'entrer ce genre de code dans Sage est de l'enregistrer dans un fichier et d'utiliser la commande `attach` comme décrit ci-dessus.

5.2 Écrire des programmes compilés

Dans les calculs mathématiques sur ordinateur, la vitesse a une importance cruciale. Or, si Python est un langage commode et de très haut niveau, certaines opérations peuvent être plus rapides de plusieurs ordres de grandeur si elles sont implémentées sur des types statiques dans un langage compilé. Certaines parties de Sage auraient été trop lentes si elles avaient été écrites entièrement en Python. Pour pallier ce problème, Sage supporte une sorte de « version compilée » de Python appelée Cython (voir [Cython] et [Pyrex]). Cython ressemble à la fois à Python et à C. La plupart des constructions Python, dont la définition de listes par compréhension, les expressions conditionnelles, les constructions comme += sont autorisées en Cython. Vous pouvez aussi importer du code depuis d'autres modules Python. En plus de cela, vous pouvez déclarer des variables C arbitraires, et faire directement des appels arbitraires à des bibliothèques C. Le code Cython est converti en C et compilé avec le compilateur C.

Pour créer votre propre code Sage compilé, donnez à votre fichier source l'exension `.spyx` (à la place de `.sage`). Avec l'interface en ligne de commande, vous pouvez charger ou attacher des fichiers de code compilé exactement comme les fichiers interprétés. Pour l'instant, le *notebook* ne permet pas d'attacher des fichiers compilés. La compilation proprement dite a lieu « en coulisse », sans que vous ayez à la déclencher explicitement. Le fichier `$SAGE_ROOT/examples/programming/sagex/factorial.spyx` donne un exemple d'implémentation compilée de la fonction factorielle, qui appelle directement la bibliothèque GMP. Pour l'essayer, placez-vous dans le répertoire `$SAGE_ROOT/examples/programming/sagex/` et entrez les commandes suivantes :

```
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
```

Le suffixe `L` ci-dessus dénote un entier long Python (voir *Le préprocesseur Sage et les différences entre Sage et Python*).

Notez que si vous quittez et relancez Sage, `factorial.spyx` sera recompilé. La bibliothèque d'objets partagés compilés se trouve dans `$HOME/.sage/temp/hostname/pid/spyx`. Ces fichiers sont supprimés lorsque vous quittez Sage.

Attention, le prétraitement des fichiers Sage mentionné plus haut N'EST PAS appliqué aux fichiers spyx, ainsi, dans un fichier spyx, `1/3` vaut 0 et non le nombre rationnel 1/3. Pour appeler une fonction `foo` de la bibliothèque Sage depuis un fichier spyx, importez `sage.all` et appelez `sage.all.foo`.

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

5.2.1 Appeler des fonctions définies dans des fichiers C séparés

Il n'est pas difficile non plus d'accéder à des fonctions écrites en C, dans des fichiers *.c séparés. Créez dans un même répertoire deux fichiers `test.c` et `test.spyx` avec les contenus suivants :

Le code C pur : `test.c`

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

Le code Cython : `test.spyx` :

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

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

Vous pouvez alors faire :

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

Si la compilation du code C généré à partir d'un fichier Cython nécessite une bibliothèque supplémentaire `foo`, ajoutez au source Cython la ligne `clib foo`. De même, il est possible d'ajouter un fichier C supplémentaire `bar` aux fichiers à compiler avec la déclaration `cfile bar`.

5.3 Scripts Python/Sage autonomes

Le script autonome suivant, écrit en Sage, permet de factoriser des entiers, des polynômes, etc. :

```
#!/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]))
```

Pour utiliser ce script, votre répertoire `SAGE_ROOT` doit apparaître dans la variable d'environnement PATH. Supposons que le script ci-dessus soit appelé `factor`, il peut alors être utilisé comme dans l'exemple suivant :

```
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 Types de données

Chaque objet Sage a un type bien défini. Python dispose d'une vaste gamme de types intégrés et la bibliothèque Sage en fournit de nombreux autres. Parmi les types intégrés de Python, citons les chaînes, les listes, les n-uplets, les entiers et les flottants :

```
sage: s = "sage"; type(s)
<type 'str'>
sage: s = 'sage'; type(s)      # guillemets simples ou doubles
<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 ajoute de nombreux autres types. Par exemple, les espaces vectoriels :

```
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'>
```

Seules certaines fonctions peuvent être appelées sur `V`. Dans d'autres logiciels mathématiques, cela se fait en notation « fonctionnelle », en écrivant `foo(V,...)`. En Sage, certaines fonctions sont attachés au type (ou classe) de l'objet et appelées avec une syntaxe « orientée objet » comme en Java ou en C++, par exemple `V.foo(...)`. Cela évite de polluer l'espace de noms global avec des dizaines de milliers de fonctions, et cela permet d'avoir plusieurs fonctions appelées `foo`, avec des comportements différents, sans devoir se reposer sur le type des arguments (ni sur des instructions case) pour décider laquelle appeler. De plus, une fonction dont vous réutilisez le nom demeure disponible : par exemple, si vous appelez quelque chose `zeta` et si ensuite vous voulez calculer la valeur de la fonction zêta de Riemann au point 0.5, vous pouvez encore écrire `s=.5; s.zeta()`.

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

La notation fonctionnelle usuelle est aussi acceptée dans certains cas courants, par commodité et parce que certaines expressions mathématiques ne sont pas claires en notation orientée objet. Voici quelques exemples.

```
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
```

Pour obtenir la liste de toutes les fonctions membres de A, utilisez la complétion de ligne de commande : tapez `A.`, puis appuyez sur la touche `[tab]` de votre clavier, comme expliqué dans la section *Recherche en arrière et complétion de ligne de commande*.

5.5 Listes, n-uplets et séquences

Une liste stocke des éléments qui peuvent être de type arbitraire. Comme en C, en C++ etc. (mais au contraire de ce qu'il se passe dans la plupart des systèmes de calcul formel usuels) les éléments de la liste sont indexés à partir de 0 :

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

Lors d'un accès à une liste, l'index n'a pas besoin d'être un entier Python. Un entier (Integer) Sage (ou un Rational, ou n'importe quoi d'autre qui a une méthode `__index__`) fait aussi l'affaire.

```
sage: v = [1,2,3]
sage: v[2]
3
sage: n = 2       # Integer (entier Sage)
sage: v[n]        # ça marche !
3
sage: v[int(n)]   # Ok aussi
3
```

La fonction `range` crée une liste d'entiers Python (et non d'entiers Sage) :

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

Cela est utile pour construire des listes par compréhension :

```
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]
```

Pour plus d'information sur les compréhensions, voir [PyT].

Une fonctionnalité merveilleuse est l'extraction de tranches d'une liste. Si `L` est une liste, `L[m :n]` renvoie la sous-liste de `L` formée des éléments d'indices m à $n-1$ inclus :

```
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]
```

Les n-uplets ressemblent aux listes, à ceci près qu'ils sont non mutables, ce qui signifie qu'ils ne peuvent plus être modifiés une fois créés.

```
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
```

Les séquences sont un troisième type Sage analogue aux listes. Contrairement aux listes et aux n-uplets, il ne s'agit pas d'un type interne de Python. Par défaut, les séquences sont mutables, mais on peut interdire leur modification en utilisant la méthode `set_immutable` de la classe `Sequence`, comme dans l'exemple suivant. Tous les éléments d'une séquence ont un parent commun, appelé l'univers de la séquence.

```
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.
```

Les séquences sont des objets dérivés des listes, et peuvent être utilisées partout où les listes peuvent l'être :

```
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'>
```

Autre exemple : les bases d'espaces vectoriels sont des séquences non mutables, car il ne faut pas les modifier.

```
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 Dictionnaires

Un dictionnaire (parfois appelé un tableau associatif) est une correspondance entre des objets « hachables » (par exemple des chaînes, des nombres, ou des n-uplets de tels objets, voir http ://docs.python.org/tut/node7.html et http ://docs.python.org/lib/typesmapping.html dans la documentation de Python pour plus de détails) vers des objets arbitraires.

```
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
```

La troisième clé utilisée ci-dessus, l'anneau des entiers relatifs, montre que les indices d'un dictionnaire peuvent être des objets compliqués.

Un dictionnaire peut être transformé en une liste de couples clé-objet contenant les mêmes données :

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

Le parcours itératifs des paires d'un dictionnaire est un idiome de programmation fréquent :

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

Comme le montre la dernière sortie ci-dessus, un dictionnaire stocke ses éléments sans ordre particulier.

5.7 Ensembles

Python dispose d'un type ensemble intégré. Sa principale caractéristique est qu'il est possible de tester très rapidement si un élément appartient ou non à un ensemble. Le type ensemble fournit les opérations ensemblistes usuelles.

```
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 a son propre type ensemble, qui est (dans certains cas) implémenté au-dessus du type Python, mais offre quelques fonctionnalités supplémentaires utiles à Sage. Pour créer un ensemble Sage, on utilise `Set(...)`. Par exemple,

```
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 Itérateurs

Les itérateurs sont un ajout récent à Python, particulièrement utile dans les applications mathématiques. Voici quelques exemples, consultez [PyT] pour plus de détails. Fabriquons un itérateur sur les carrés d'entiers positifs jusqu'à 10000000.

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

Nous créons maintenant un itérateur sur les nombres premiers de la forme $4p + 1$ où p est lui aussi premier, et nous examinons les quelques premières valeurs qu'il prend.

```
sage: w = (4*p + 1 for p in Primes() if is_prime(4*p+1))
sage: w
<generator object <genexpr> at 0x...>
sage: w.next()
13
```

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

Certains anneaux, par exemple les corps finis et les entiers, disposent d'itérateurs associés :

```
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 Boucles, fonctions, structures de contrôle et comparaisons

Nous avons déjà vu quelques exemples courants d'utilisation des boucles `for`. En Python, les boucles `for` ont la structure suivante, avec une indentation :

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

0
1
2
3
4
```

Notez bien les deux points à la fin de l'instruction for (il n'y a pas de « do » ou « od » comme en Maple ou en GAP) ainsi que l'indentation du corps de la boucle, formé de l'unique instruction `print(i)`. Cette indentation est significative, c'est elle qui délimite le corps de la boucle. Depuis la ligne de commande Sage, les lignes suivantes sont automatiquement indentées quand vous appuyez sur `entrée` après un signe « : », comme illustré ci-dessous.

```
sage: for i in range(5):
...       print(i)  # appuyez deux fois sur entrée ici
0
1
2
3
4
```

Le signe = représente l'affectation. L'opérateur == est le test d'égalité.

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

```
11
13
14
```

Retenez bien que l'indentation détermine la structure en blocs des instructions `if`, `for` et `while` :

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

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

Naturellement, l'exemple précédent n'est pas une implémentation efficace du symbole de Legendre ! Il est simplement destiné à illustrer différents aspects de la programmation Python/Sage. La fonction {kronecker} fournie avec Sage calcule le symbole de Legendre efficacement, en appelant la bibliothèque C de PARI.

Remarquons aussi que les opérateurs de comparaison numériques comme ==, ! =, <=, >=, >, < convertissent automatiquement leurs deux membres en des nombres du même type lorsque c'est possible :

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

Deux objets quelconques ou presque peuvent être comparés, sans hypothèse sur l'existence d'un ordre total sous-jacent.

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

Pour évaluer des inégalités symboliques, utilisez `bool` :

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

Lorsque l'on cherche à comparer des objets de types différents, Sage essaie le plus souvent de trouver une coercition canonique des deux objets dans un même parent. Si cela réussit, la comparaison est faite entre les objets convertis ; sinon, les objets sont simplement considérés comme différents. Pour tester si deux variables font référence au même objet, on utilise l'opérateur `is`.

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

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

Dans les deux lignes suivantes, la première égalité est fausse parce qu'il n'y a pas de morphisme canonique $\mathbf{Q} \to \mathbf{F}_5$, et donc pas de manière canonique de comparer l'élément 1 de $\mathbf{F}_5$ à $1 \in \mathbf{Q}$. En revanche, il y a une projection canonique $\mathbf{Z} \to \mathbf{F}_5$, de sorte que la deuxième comparaison renvoie « vrai ». Remarquez aussi que l'ordre des membres de l'égalité n'a pas d'importance.

```
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
```

ATTENTION : La comparaison est plus restrictive en Sage qu'en Magma, qui considère $1 \in \mathbf{F}_5$ comme égal à $1 \in \mathbf{Q}$.

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

5.10 Profilage (profiling)

Auteur de la section : Martin Albrecht (malb@informatik.uni-bremen.de)

> "Premature optimization is the root of all evil." - Donald Knuth (« L'optimisation prématurée est la source de tous les maux. »)

Il est parfois utile de rechercher dans un programme les goulets d'étranglements qui représentent la plus grande partie du temps de calcul : cela peut donner une idée des parties à optimiser. Cette opération s'appelle profiler le code. Python, et donc Sage, offrent un certain nombre de possibilités pour ce faire.

La plus simple consiste à utiliser la commande `prun` du shell interactif. Elle renvoie un rapport qui résume les temps d'exécution des fonctions les plus coûteuses. Pour profiler, par exemple, le produit de matrices à coefficients dans un corps fini (qui, dans Sage 1.0, est lent), on entre :

```
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)
  ...
```

Ici, `ncalls` désigne le nombre d'appels, `tottime` le temps total passé dans une fonction (sans compter celui pris par les autres fonctions appelées par la fonction en question), `percall` est le rapport `tottime` divisé par `ncalls`. `cumtime` donne le temps total passé dans la fonction en comptant les appels qu'elle effectue, la deuxième colonne `percall` est le quotient de `cumtime` par le nombre d'appels primitifs, et `filename :lineno(function)` donne pour chaque fonction le nom de fichier et le numéro de la ligne où elle est définie. En règle générale, plus haut la fonction apparaît dans ce tableau, plus elle est coûteuse — et donc intéressante à optimiser.

Comme d'habitude, `prun ?` donne plus d'informations sur l'utilisation du profileur et la signification de sa sortie.

Il est possible d'écrire les données de profilage dans un objet pour les étudier de plus près :

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

Remarque : entrer `stats = prun -r A\*A` à la place des deux premières lignes ci-dessus provoque une erreur de syntaxe, car prun n'est pas une fonction normale mais une commande du shell IPython.

Pour obtenir une jolie représentation graphique des données de profilage, vous pouvez utiliser le profileur hotshot, un petit script appelé `hotshot2cachetree` et (sous Unix uniquement) le programme `kcachegrind`. Voici le même exemple que ci-dessus avec le profileur hotshot :

```
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()
```

À ce stade le résultat est dans un fichier `pythongrind.prof` dans le répertoire de travail courant. Convertissons-le au format cachegrind pour le visualiser.

Dans le shell du système d'exploitation, tapez

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

Le fichier `cachegrind.out.42` peut maintenant être examiné avec `kcachegrind`. Notez qu'il est important de respecter la convention de nommage `cachegrind.out.XX`.

CHAPITRE 6

Calcul distribué

Sage contient une puissante infrastructure de calcul distribué appelé Distributed Sage (`dsage`).

6.1 Vue d'ensemble

Distributed Sage est une infrastructure qui permet de faire du calcul distribué à partir de l'intérieur de Sage. Il contient un serveur, un client, des programmes esclaves, et un ensemble de classes que l'on peut dériver pour écrire des jobs de calcul distribué. Il est destiné en priorité aux calculs distribués « grossièrement », avec peu de communications entre les jobs (« grille de calcul »).

Distributed Sage est divisé en trois parties :

1. Le **serveur** est responsable de la distribution des jobs, de leur soumission et de leur collecte. Il dispose d'une interface web qui permet de surveiller l'état des jobs et d'effectuer diverses autres tâches d'aministration.
2. Le **client** a pour rôle de soumettre des jobs au serveur et de récupérer les résultats.
3. Les **esclaves** sont les programmes qui font le calcul proprement dit.

6.2 Prise en main

Voici quelques exemples qui montrent comment démarrer avec `dsage`.

6.2.1 Exemple 1

1. Lancez `dsage.setup()`. Cela va configurer la base de données SQLite et créer une paire de clés (i.e. une clé publique et une clé privée) qui seront utilisées pour la communication SSL. Un utilisateur par défaut sera créé, avec comme nom (par défaut) votre nom d'utilisateur courant.
2. Lancez ensuite `d = dsage.start_all()`. Cette commande démarre le serveur, le serveur web et 2 esclaves. Elle renvoie un objet (`d`) qui représente une connexion vers le serveur. Désormais, l'essentiel de vos interactions avec `dsage` passeront par cet objet `d`.
3. Ouvrez votre navigateur web à l'adresse http ://localhost :8082 pour accéder à l'interface web de `dsage`. À partir de celle-ci, vous pourrez voir le statut de vos jobs, les esclaves connectés et diverses autres informations importantes sur votre serveur `dsage`.
4. Commençons par un exemple simple. Tapez `job = d('2+2')`. En consultant l'interface web, vous devriez maintenant voir un nouveau job dans le tableau. Un de vos processus esclaves va maintenant récupérer le job,

l'exécuter et vous renvoyer le résultat. Il est possible qu'il ne soit pas encore visible car pour un calcul simple comme celui-là, c'est le surcoût correspondant à la communication par le réseau qui domine le temps de calcul. Si vous souhaitez attendre que votre job se termine, appelez `job.wait()`, qui bloquera jusqu'à ce que le job soit terminé. Vous pourrez ensuite inspecter `job.result` pour lire le résultat. N'importe quel calcul peut être fait de cette façon en appelant `d`.

6.2.2 Exemple 2

Cet exemple montre comment utiliser la classe `DistributedFactor` intégrée à `dsage`. DistributedFactor tente de factoriser un nombre par une combinaison de l'algorithme ECM (factorisation par les courbes elliptiques), du crible quadratique et de l'élimination des petits facteurs par divisions successives.

1. Lancez `d = dsage.start_all()` si vous n'avez pas encore démarré votre session `dsage`. Sinon, vous pouvez continuer à utiliser l'instance précédente de `d`.
2. Démarrez le job de factorisation distribuée par la commande `job_factorisation = DistributedFactor(d, nombre)`. Vous pouvez prendre des nombres assez grands, essayez par exemple $2^{360} - 1$. Pour voir si la factorisation est terminée, consultez l'attribut `job_factorisation.done`. Une fois le job terminé, les facteurs premiers trouvés sont disponibles dans `job_factorisation.prime_factors`.

6.3 Fichiers

`dsage` garde quelques fichiers dans `$SAGE_ROOT/.sage/dsage` :

1. `pubcert.pem` et `cacert.pem` : la clé publique et la clé privée utilisées par le serveur pour les communications SSL.
2. `dsage_key.pub` et `dsage_key` : les clés utilisées pour authentifier l'utilisateur.
3. `db/` : sous-répertoire qui contient la base de données `dsage`.
4. `*.log` : journaux du serveur et des esclaves.
5. `tmp_worker_files/` : sous-répertoire où les esclaves sauvegardent les jobs qu'ils ont traités.

CHAPITRE 7

Postface

7.1 Pourquoi Python ?

7.1.1 Les avantages de Python

Le langage d'implémentation de la base de Sage est le langage Python (voir [Py]), même si le code qui doit s'exécuter rapidement est écrit dans un langage compilé. Python présente plusieurs avantages :

– L'**enregistrement d'objets** est très facile en Python. Il existe en Python un vaste support pour enregistrer (presque) n'importe quel objet dans des fichiers sur le disque ou dans une base de données.
– Python fournit d'excellents outils pour la **documentation** des fonctions et des packages du code source, ce qui comprend l'extraction automatique de documentation et le test automatique de tous les exemples. Ces tests automatiques sont exécutés régulièrement de façon automatique, ce qui garantit que les exemples donnés dans la documentation fonctionnent comme indiqué.
– **Gestion de la mémoire :** Python possède désormais un gestionnaire de mémoire bien pensé et robuste ainsi qu'un ramasse-miettes (*garbage collector*) qui traite correctement les références circulaires et tient compte des variables locales dans les fichiers.
– **Énormément de packages** d'ores et déjà disponibles pour Python pourraient se révéler d'un grand intérêt pour les utilisateurs de Sage : analyse numérique et algèbre linéaire, visualisation 2D et 3D, réseau (pour le calcul distribué, la mise en place de serveurs - par exemple twisted), bases de données, etc.
– **Portabilité :** Python se compile sans difficulté et en quelques minutes sur la plupart des plates-formes.
– **Gestion des exception :** Python possède un système sophistiqué et bien pensé de gestion des exceptions, grâce auquel les programmes peuvent rétablir leur fonctionnement normal même si des erreurs surviennent dans le code qu'ils appellent.
– **Débogueur :** Python comprend un débogueur. Ainsi, quand un programme échoue pour une raison quelconque, l'utilisateur peut consulter la trace complète de la pile d'exécution, inspecter l'état de toutes les variables pertinentes et se déplacer dans la pile.
– **Profileur :** Il existe un profileur Python, qui exécute le code et renvoie un rapport qui détaille combien de fois et pendant combien de temps chaque fonction a été appelée.
– **Un langage :** Au lieu d'écrire un **nouveau langage** pour les mathématiques comme cela a été fait pour Magma, Maple, Mathematica, Matlab, GP/PARI, GAP, Macaulay 2, Simath, etc., nous utilisons le langage Python, qui est un langage de programmation répandu, activement développé et optimisé par des centaines de développeurs qualifiés. Python, avec son processus de développement éprouvé, fait partie des success stories majeures de l'open source (see [PyDev]).

7.1.2 Le préprocesseur Sage et les différences entre Sage et Python

Certains aspects mathématiques de Python peuvent induire des confusions. Aussi, Sage se comporte différemment de Python à plusieurs égards.

– **Notation de l'exponentiation :** `**` au lieu de `^`. En Python, `^` désigne le "xor" (ou exclusif bit à bit) et non l'exponentiation. Ainsi en Python, on a

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

Cette utilisation de `^` peut paraître étrange et surtout inefficace pour une utilisation purement mathématique puisque le ou exclusif n'est que rarement utilisé. Par commodité, Sage prétraite chaque ligne de commande avant de la transmettre à Python, en remplaçant par exemple les apparitions de `^` (en-dehors des chaînes de caractères) par des `**` :

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

– **Division entière :** L'expression Python `2/3` ne se comporte pas de la manière à laquelle s'attendraient des mathématiciens. En Python, si `m` et `n` sont de type int, alors `m/n` est aussi de type int, c'est le quotient entier de `m` par `n`. Par conséquent, `2/3=0`. Il y a eu dans la communauté Python des débats sur une éventuelle modification du langage de sorte que `2/3` renvoie un flottant `0.6666...` et que ce soit `2//3` qui renvoie `0`.
Dans l'interpréteur Sage, nous réglons cela en encapsulant automatiquement les entiers litéraux par `Integer( )` et en faisant de la division un constructeur pour les nombres rationnels. Par exemple :

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

– **Entiers longs :** Python possède nativement un support pour les entiers de précision arbitraire, en plus des int du langage C. Les entiers longs Python sont significativement plus lents que ceux que GMP fournit et sont marqués à l'affichage par un `L` qui les distingue des int (il est pas prévu de changer cela à court terme). Sage implémente les entiers en précision arbitraire en utilisant la bibliothèque C GMP. Les entiers longs GMP utilisés par Sage s'affichent sans le `L`.

Plutôt que de modifier l'interpréteur Python (comme l'ont fait certaines personnes pour leurs projets internes), nous utilisons le langage Python exactement comme il est et rajoutons un pré-parseur pour IPython de sorte que la ligne de commande de IPython se comporte comme l'attend un mathématicien. Ceci signifie que tout code Python existant peut être utilisé sous Sage. Toutefois, il faut toujours respecter les règles standards de Python lorsque l'on écrit des packages à importer dans Sage.

(Pour installer une bibliothèque Python, trouvée sur Internet par exemple, suivez les instructions mais exécutez `sage -python` au lieu de `python`. La plupart du temps, ceci signifie concrètement qu'il faut taper `sage -python setup.py install`.)

7.2 Comment puis-je contribuer ?

Si vous souhaitez contribuer au developpement de Sage, votre aide sera grandement appréciée ! Cela peut aller de contributions substantielles en code au signalement de bogues en passant par l'enrichissement de la documentation.

Parcourrez la page web de Sage pour y trouver les informations pour les développeurs. Entre autres choses, vous trouverez une longue liste de projets en lien avec Sage rangés par priorité et catégorie. Le "Sage Programming Guide" contient également des informations utiles. Vous pouvez aussi faire un tour sur le groupe Google `sage-devel`.

7.3 Comment citer Sage ?

Si vous écrivez un article qui utilise Sage, merci d'y préciser les calculs faits avec Sage en citant

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

dans votre bibliographie (en remplaçant 2.6 par la version de Sage que vous avez utilisée). De plus, pensez à rechercher les composantes de Sage que vous avez utilisés pour vos calculs, par exemple PARI, GAP, Singular, Maxima et citez également ces systèmes. Si vous vous demandez quel logiciel votre calcul utilise, n'hésitez pas à poser la question sur le groupe Google `sage-devel`. Voir *Polynômes univariés* pour une discussion plus approfondie de ce point.

Si vous venez de lire d'une traite ce tutoriel et que vous avez une idée du temps qu'il vous a fallu pour le parcourir, merci de nous le faire savoir sur le groupe Google `sage-devel`.

Amusez-vous bien avec Sage !

CHAPITRE 8

Annexe

8.1 Priorité des opérateurs arithmétiques binaires

Combien font `3^2*4 + 2%5` ? Le résultat (38) est déterminé par le « tableau de priorité des opérateurs » suivant. Il est dérivé de celui donné § 5.14 du manuel de référence de Python (*Python Language Reference Manual*, de G. Rossum et F. Drake.) Les opérations sont données par priorités croissantes.

Opérateur	Description
or	ou booléen
and	et booléen
not	négation booléenne
in, not in	appartenance
is, is not	test d'identité
>, <=, >, >=, ==, !=, <>	comparaisons
+, -	addition, soustraction
*, /, %	multiplication, division, reste
**, ^	exponentiation

Ainsi, pour calculer `3^2*4 + 2%5`, Sage « met les parenthèses » comme suit : `((3^2)*4) + (2%5)`. Il calcule donc d'abord `3^2`, ce qui fait `9`, puis `(3^2)*4` et `2%5`, et enfin ajoute les valeurs de ces deux dernières expressions.

CHAPITRE 9

Bibliographie

CHAPITRE 10

Index et tables

– *Index*
– *Index du module*
– *Page de recherche*

Bibliographie

[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.