Calcul mathématique avec Sage

Alexandre Casamayou, Guillaume Connan, Thierry Dumont, Laurent Fousse, Francois Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet, Nicolas M. Thiéry, Paul Zimmermann

Calcul mathématique avec Sage

Alexandre Casamayou Guillaume Connan Thierry Dumont
Laurent Fousse François Maltey Matthias Meulien
Marc Mezzarobba Clément Pernet Nicolas M. Thiéry
Paul Zimmermann

Version 1.0 — Juillet 2010

Des parties de cet ouvrage sont inspirées de l'ouvrage Calcul formel : mode d'emploi. Exemples en Maple de Philippe Dumas, Claude Gomez, Bruno Salvy et Paul Zimmermann [DGSZ95], diffusé sous la même licence, notamment les sections 1.6 et 2.1.2, et la section 2.3.5.

Une partie des exemples Sage de la section 12 sont tirés du tutoriel de `MuPAD-Combinat` [HT04] et `Sage-combinat`. Le dénombrement des arbres binaires complets de §12.1.2 est en partie inspiré d'un sujet de TP de Florent Hivert.

L'exercice 9 sur le problème de Gauss est tiré d'un problème de François Pantigny et l'exercice 17 sur l'effet Magnus est extrait d'un TD de Jean-Guy Stoliaroff.

Préface

Ce livre est destiné à tous ceux qui désirent utiliser efficacement un système de calcul mathématique, en particulier le logiciel Sage. Ces systèmes offrent une multitude de fonctionnalités, et trouver comment résoudre un problème donné n'est pas toujours facile. Un manuel de référence fournit une description analytique et en détail de chaque fonction du système ; encore faut-il savoir le nom de la fonction que l'on cherche ! Le point de vue adopté ici est complémentaire, en donnant une vision globale et synthétique, avec un accent sur les mathématiques sous-jacentes, les classes de problèmes que l'on sait résoudre et les algorithmes correspondants.

La première partie, plus spécifique au logiciel Sage, constitue une prise en main du système. Cette partie se veut accessible aux élèves de lycée, et donc a fortiori aux étudiants de BTS et de licence. Les autres parties s'adressent à des étudiants au niveau agrégation, et sont d'ailleurs organisées en suivant le programme de l'épreuve de modélisation de l'agrégation de mathématiques. Contrairement à un manuel de référence, les concepts mathématiques sont clairement énoncés avant d'illustrer leur mise en œuvre avec Sage. Ce livre est donc aussi un livre sur les mathématiques.

Pour illustrer cet ouvrage, le choix s'est porté naturellement vers Sage, car c'est un logiciel libre, que tout un chacun peut librement utiliser, modifier et redistribuer. Ainsi l'élève qui a appris Sage au lycée pourra l'utiliser quelle que soit sa voie professionnelle : en licence, Master, doctorat, en école d'ingénieur, en entreprise, ... Sage est un logiciel encore jeune par rapport aux logiciels concurrents, et malgré ses capacités déjà étendues, comporte encore de nombreuses *bogues*. Mais par sa communauté très active de développeurs, Sage évolue très vite. Chaque utilisateur de Sage peut rapporter une bogue — et éventuellement sa solution — sur `trac.sagemath.org` ou via la liste `sage-support`.

Pour rédiger ce livre, nous avons utilisé la version 4.4.4 de Sage. Néanmoins, les exemples doivent fonctionner avec toute version ultérieure. Par contre, certaines affirmations peuvent ne plus être vérifiées, comme par exemple le fait que Sage utilise Maxima pour évaluer des intégrales numériques.

Quand j'ai proposé en décembre 2009 à Alexandre Casamayou, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meu-

lien, Marc Mezzarobba, Clément Pernet et Nicolas Thiéry d'écrire un livre sur Sage, tous ont répondu présent, malgré une charge de travail déjà importante. Je tiens à les remercier, notamment pour le respect du planning serré que j'avais fixé.

Tous les auteurs remercient les personnes suivantes qui ont relu une version préliminaire de ce livre : Gaëtan Bisson, Françoise Jung ; ainsi qu'Emmanuel Thomé pour son aide précieuse lors de la réalisation de ce livre, et Sylvain Chevillard pour ses conseils typographiques.

En rédigeant ce livre, nous avons beaucoup appris sur Sage, nous avons bien sûr rencontré quelques bogues, dont certaines sont déjà corrigées. Nous espérons que ce livre sera utile à d'autres, lycéens, étudiants, professeurs, ingénieurs, chercheurs, ... Cette première version comportant certainement de nombreuses imperfections, nous attendons en retour du lecteur qu'il nous fasse part de toute erreur, critique ou suggestion pour une version ultérieure ; merci d'utiliser pour cela la page `sagebook.gforge.inria.fr`.

Villers-lès-Nancy, France
juillet 2010

Paul Zimmermann

Table des matières

Première partie

Prise en main du logiciel

1

Premiers pas avec Sage

Ce chapitre d'introduction présente la *tournure d'esprit* du logiciel mathématique Sage. Les autres chapitres de cette partie développent les notions de base de Sage : effectuer des calculs numériques ou symboliques en analyse, opérer sur des vecteurs et des matrices, écrire des programmes, manipuler des listes de données, construire des graphes, etc. Les parties suivantes de cet ouvrage approfondissent les branches des mathématiques dans lesquelles l'informatique fait preuve d'une grande efficacité.

1.1 Le logiciel Sage

1.1.1 Un outil pour les mathématiques

Sage est un logiciel qui implante des algorithmes mathématiques dans des domaines variés. En premier lieu, le système opère sur différentes sortes de nombres : les nombres entiers ou rationnels, les approximations numériques des réels et des complexes, avec une précision variable, et les entiers modulaires des corps finis. L'utilisateur peut très rapidement utiliser Sage comme n'importe quelle calculette scientifique éventuellement graphique.

Les collégiens apprennent le calcul algébrique pour résoudre des équations affines, développer, simplifier, et parfois factoriser des expressions. Sage effectue directement ces calculs dans le cadre bien plus complet des polynômes et des fractions rationnelles.

L'analyse consiste à étudier les fonctions trigonométriques et les autres fonctions usuelles comme la racine carrée, les puissances, l'exponentielle, le logarithme. Sage reconnaît ces expressions, et effectue dérivations, intégrations et calculs de limites ; il simplifie aussi les sommes, développe en séries,

résout certaines équations différentielles, etc.

Le logiciel Sage implante les résultats de l'algèbre linéaire sur les vecteurs, les matrices et les sous-espaces vectoriels. Le système permet aussi d'aborder, d'illustrer et de traiter de différentes façons les probabilités, les statistiques et les questions de dénombrement.

Ainsi les domaines mathématiques traités par Sage sont multiples, de la théorie des groupes à l'analyse numérique. Il peut représenter graphiquement les résultats obtenus sous la forme d'une image, d'un volume, ou d'une animation.

Utiliser un même logiciel pour aborder ces différents aspects des mathématiques libère le mathématicien, quel que soit son niveau, des problèmes de transfert de données entre les logiciels et de l'apprentissage des syntaxes de plusieurs langages informatiques. Sage s'efforce d'être homogène entre ces différents domaines.

1.1.2 Accès à Sage

Le logiciel Sage est libre ; ses auteurs n'imposent aucune restriction à la diffusion, à l'installation et même à la modification de Sage dès l'instant que le code source reste libre. Si Sage était un livre, il pourrait être emprunté et photocopié sans limite dans toutes les bibliothèques ! Cette licence est en harmonie avec les buts d'enseignement et de recherche pour faciliter la diffusion des connaissances.

William Stein, un enseignant-chercheur américain, commence le développement de Sage en 2005 dans le but d'écrire un logiciel d'expérimentation en algèbre et géométrie, *Software for Algebra and Geometry Experimentation*.

Ces adresses pointent sur le site internet de référence de Sage et une présentation en français du logiciel :

`http://www.sagemath.org` `http://www.sagemath.fr`
`http://www.sagemath.org/doc/` pour le tutoriel

La liste de diffusion en langue anglaise est la plus adaptée aux questions des utilisateurs de Sage, et les utilisateurs francophones peuvent aussi dialoguer sur une seconde liste :

```
mail:sage-support@googlegroups.com
mail:sagemath-edu@mail.irem.univ-mrs.fr
```

L'utilisateur peut au choix compiler le code source ou installer directement le programme sur les systèmes d'exploitation Linux, Windows, Mac OS X et Solaris. Certaines distributions de Linux proposent un paquetage de Sage prêt à être utilisé, mais la version correspondante n'est pas toujours la plus récente.

L'interface d'utilisation de Sage peut être un terminal (`xterm` sous Linux) ou une fenêtre d'un navigateur internet comme `Firefox`. Dans le premier cas

il suffit de lancer le programme par la commande `sage`, et de saisir ensuite les calculs ligne par ligne.

Dans le second cas, le navigateur internet se connecte à un serveur Sage, et l'utilisateur ouvre une nouvelle feuille de calcul par le lien `New Worksheet`. Ce serveur peut être le programme Sage installé par l'utilisateur et lancé par la commande `sage -notebook`[1].

Le développement de Sage est relativement rapide car il privilégie l'usage de bibliothèques de calcul déjà publiées. Plus précisément le programme est écrit en langage Python, ainsi les instructions de Sage sont en fait des appels à ces bibliothèques mathématiques liées à Python, et les programmes écrits pour Sage sont des instructions évaluées par Python. Par conséquent les syntaxes de Sage et de Python sont similaires mais Sage modifie légèrement le fonctionnement standard de Python pour le spécialiser dans les mathématiques.

1.2 Premier calcul, aide en ligne et complétion

Selon l'interface, l'utilisateur lance un calcul par la touche de passage à la ligne ⟨Entrée⟩ ou ⟨Maj – Entrée⟩, ou en validant le lien `evaluate` à la souris :

```
sage: 2*3
  6
```

L'accès à l'aide en ligne s'obtient en faisant suivre le nom de la commande d'un point d'interrogation `?` :

```
sage: diff?
```

La page de documentation contient la description de la fonction, sa syntaxe, puis plusieurs exemples. Les touches fléchées du clavier et les commandes de l'éditeur `vi` permettent de naviguer dans la page d'aide :

`j` : descendre `k` : monter `q` : quitter `h` : aide sur l'aide

De même faire suivre une commande par deux points d'interrogation renvoie le code source de cette commande ; celui-ci contient d'ailleurs au début le texte de l'aide en ligne précédente :

```
sage: diff??
```

La touche de tabulation ⟨tab⟩ à la suite d'un début de mot montre quelles sont les commandes commençant par ces lettres : `arc` suivi de la tabulation affiche ainsi le nom de toutes les fonctions trigonométriques réciproques. En particulier la commande `a.`⟨tab⟩ énumère toutes les méthodes qui s'appliquent à l'objet `a`. La fin de ce chapitre abordera l'objet `vector` :

[1] La communication entre Sage et le navigateur se fait généralement par l'intermédiaire de l'addresse `http://localhost:8000/home/admin`

```
sage: a = vector([1, 2, 3])
sage: a.⟨tab⟩
a.Mod                 a.get                 a.order
a.additive_order      a.inner_product       a.pairwise_product
a.apply_map           a.integral            a.parent
a.base_extend         a.integrate           a.plot
a.base_ring           a.is_dense            a.plot_step
--More--
```

L'instruction `tutorial()` de Sage accède à un tutoriel en anglais.

1.3 Syntaxe générale

Les instructions élémentaires de Sage sont en général traitées ligne par ligne. L'interpréteur de commandes considère le caractère dièse « # » comme un début de commentaire et ignore le texte saisi jusqu'à la fin de la ligne.

Le point-virgule « ; » sépare les instructions placées sur une même ligne :

```
sage: 2*3 ; 3*4 ; 4*5            # un commentaire, 3 résultats
  6
  12
  20
```

Une commande du mode terminal peut être saisie sur plusieurs lignes en faisant précéder les retours à la ligne intermédiaires d'une contre-oblique « \ » ; ces retours à la ligne sont considérés comme un simple blanc :

```
sage:   123 \
....: + 345                                  # de résultat 468
```

Un identificateur — c'est-à-dire un nom de variable, de fonction, etc. — est constitué uniquement de lettres, de chiffres ou du caractère de soulignement « _ » sans commencer par un chiffre. Les identificateurs doivent être différents des mots-clefs du langage ; de façon anecdotique la liste des mots-clés s'obtient après le chargement d'une bibliothèque spécialisée de Python dans Sage :

```
sage: import keyword ; keyword.kwlist
```

Ces mots-clefs forment le noyau du langage Python.

- `while`, `for...in` et `if...elif...else` décrivent boucles et tests,
- `continue`, `break`, `try...except...finally` et `raise...`. modifient le déroulement des programmes,
- `def`, `lambda`, `return` et `global` définissent les fonctions,
- `and`, `or`, `not` et `is` sont les opérateurs logiques,
- `del` détruit toutes les propriétés associées à une variable,
- `print` affiche ses arguments, et `assert` aide au débogage,

– `class` et `with` interviennent en programmation objet,
– `from...import...as...` appelle une bibliothèque,
– `exec...in` interprète une commande,
– `pass` est l'instruction vide et `yield` le résultat d'un itérateur.

Par ailleurs ces termes jouent un rôle particulier même s'ils ne sont pas définis dans le noyau de Python :

– `None`, `True` et `False` sont des constantes prédéfinies du système,
– $\texttt{pi} = \pi$, $\texttt{euler\_gamma} = \gamma = \lim_{n\to+\infty} \sum_{k=1}^{n} \frac{1}{k} - \ln n$,
$\texttt{I} = \mathrm{i} \in \mathbb{C}$, $\texttt{golden\_ratio} = \frac{1+\sqrt{5}}{2}$,
$\texttt{e} = \exp(1)$ et $\texttt{catalan} = \lim_{n\to+\infty} \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$
représentent les constantes mathématiques,
– `infinity` et les deux voyelles à la suite `oo` correspondent à l'infini ∞,
– et la commande interactive `quit` quitte la session de Sage en cours.

Les fonctions de Sage comme `cos(..)` ou `integrate(..)` sont codées en Python. Le bon fonctionnement de Sage impose de ne pas utiliser ces identificateurs comme des variables au risque de détériorer en profondeur le système.

Ces caractères jouent un rôle spécial dans Sage :

– `,` et `;` séparent les arguments et les instructions,
– `:` commence un bloc d'instructions, et construit des sous-listes,
– `.` est le séparateur décimal et effectue l'accès aux méthodes des objets,
– `=` effectue l'affectation des variables,
– `+`, `-`, `*` et `/` définissent les quatre opérations, et `^` ou `**`, la puissance,
– `%` et `//` calculent reste et quotient des divisions euclidiennes,
– `+=`, `-=`, `*=`, `/=`, `^=` et `**=` modifient directement une variable,
– `<`, `<=`, `>` et `>=` effectuent les comparaisons,
– `==`, `!=`, `<>` et `is` testent les égalités,
– `&`, `|`, `^^`, `<<` et `>>` opèrent sur les nombres binaires,
– `#` commence des commentaires jusqu'à la fin de la ligne,
– `[..]` définit les listes et accède à leurs éléments,
– `(..)` regroupe des arguments et construit les énumérations immuables,
– `{..:..}` construit les dictionnaires, `\` est le caractère d'échappement,
– `@` est associé aux décorateurs des fonctions,
– `?` appelle l'aide en ligne, `$` n'intervient pas dans le langage, et
– `_` et `__` reprennent les deux derniers résultats.

L'évaluation d'une fonction impose de placer ses éventuels arguments entre parentèses, comme dans `cos(pi)` ou dans la fonction sans argument `reset()`. Cependant ces parenthèses sont superflues pour les arguments d'une commande : les instructions `print(6*7)` et `print 6*7` sont équivalentes. Le nom d'une fonction sans argument ni parenthèse représente la fonction

elle-même et n'effectue aucun calcul.

En programmation objet, une méthode `meth` associée à un objet `obj` est une fonction qui spécialisée sur ce type de données. Python est un langage objet dont la syntaxe d'appel d'une méthode est `obj.meth(...)` ; dans certains cas la commande `meth(obj,...)` est équivalente.

1.4 Variables

1.4.1 Variables et affectations

Sage note par le signe égal « = » la commande d'affectation d'une valeur à une variable. La partie située à droite du caractère « = » est d'abord évaluée puis sa valeur est mémorisée dans la variable dont le nom est à gauche :

```
y=3 ; y=3*y+1 ; y=3*y+1 ; y                    # résultat final 31
```

Les affectations précédentes modifient la valeur de la variable `y` sans afficher de résultat intermédiaire, la dernière de ces quatre commandes affiche la valeur 31 de la variable `y` à la fin de ces calculs.

La commande `del x` supprime l'affectation de la variable `x`, et la fonction sans paramètre `reset()` réinitialise l'ensemble des variables.

L'affectation de plusieurs variables de façon parallèle ou synchronisée est aussi possible, ces commandes sont équivalentes et ne correspondent en rien aux affectations successives `a=b;b=a` :

```
sage: a,b = 10,20   # (a,b)=(10,20) et [a,b]=[10,10] possibles
sage: a,b = b,a                          # résultat a=20 et b=10
```

Cette dernière affectation est équivalente à l'échange des valeurs des variables `a` et `b` en utilisant une variable intermédiaire :

```
sage: temp = a ; a = b ; b = temp   # est équivalent à a,b=b,a
```

L'affectation multiple affecte la même valeur à plusieurs variables, avec une syntaxe de la forme `a=b=c=0` ; les instructions `x+=5` et `n*=2` sont respectivement équivalentes à `x=x+5` et à `n=n*2`.

1.4.2 Variables symboliques

Sage distingue les variables au sens informatique des symboles intervenant dans les expressions mathématiques formelles. Les variables formelles doivent être déclarées par la fonction `var(..)` avant d'être employées :

```
sage: var('a b') ; c = 2  ; u = a*c + b ; u
  2*a + b
```

L'affectation de `u` opère sur une variable qui *prend une valeur* au sens informatique du terme, alors que la variable `a` est une variable symbolique qui *reste* `a`.

Par défaut la seule variable symbolique de Sage est `x`. L'argument de la fonction `var(..)` est une chaîne de caractères placée entre parenthèses et délimitée par des guillemets simples `'...'` ou doubles `"..."`; les noms des variables sont alors séparés par des virgules « `,` » ou par des espaces. Cette fonction remplace l'éventuelle affectation associée à une telle variable par la variable symbolique de même nom, et renvoie comme résultat cette variable symbolique ou la liste des variables symboliques crées. La commande `del...` détruit aussi les variables symboliques. L'instruction `automatic_names(True)` dans le navigateur internet permet d'utiliser n'importe quel nom comme variable symbolique sans déclaration préalable.

L'exemple suivant illustre que la commande `var('u')` crée et affecte en fait la variable symbolique u à la variable informatique `u`; ensuite la variable affectée `u` a pour valeur l'expression symbolique $u+2$ et Sage n'effectue aucune substitution ni réécriture de l'expression finale; cette forme de code est cependant déconseillée car il n'est ensuite plus possible de récupérer simplement la variable symbolique u :

```
var('u') ; u=u+1 ; u=u+1 ; u                    # de résultat u+2
```

Cet autre exemple illustre comment échanger les valeurs symboliques des variables `a` et `b` par des sommes et des différences sans variable intermédiaire :

```
sage: var ('x y') ; a = x ; b = y          # au début a=x et b=y
sage: a = a+b ; b = a-b ; a = a-b          # à la fin a=y et b=x
sage: a, b                                 #      ici a=y et b=x
```

Cette dernière commande est équivalente à `(a,b)` et construit l'énumération ou séquence des deux termes a et b à la manière d'un couple.

Par ailleurs Sage effectue le test de comparaison entre deux expressions par le double signe d'égalité « `==` » :

```
sage: 2+2 == 2^2, 3*3 == 3^3          # de réponse (True, False)
```

1.5 Calcul formel et méthodes numériques

Un système de calcul formel est un logiciel qui a pour but de manipuler, de simplifier et de calculer des formules mathématiques en appliquant uniquement des transformations exactes. Le terme *formel* s'oppose ici à *numérique*; il signifie que les calculs sont effectués sur des formules, de façon algébrique, en manipulant formellement des symboles. Pour cette raison l'expression *calcul symbolique* est parfois employée à la place de *calcul formel*, et le terme anglais fait référence à l'algèbre sous le vocable *computer algebra*.

Les calculatrices manipulent de façon exacte les nombres entiers ayant moins d'une douzaine de chiffres ; les nombres plus grands sont arrondis, ce qui entraîne des erreurs. Ainsi une calculatrice numérique évalue de façon erronée l'expression suivante en obtenant 0 à la place de 1 :

$$(1 + 10^{50}) - 10^{50}.$$

De telles erreurs sont difficilement détectables si elles se produisent lors d'un calcul intermédiaire sans être prévues — ni facilement prévisibles — par une étude théorique. Les systèmes de calcul formel, au contraire, s'appliquent à repousser ces limites et à ne faire aucune approximation sur les nombres entiers pour que les opérations qui en découlent soient exactes : ils répondent 1 au calcul précédent.

Les méthodes d'analyse numérique approchent à une précision donnée (par une méthode des trapèzes, de Simpson, de Gauss, etc.) l'intégrale $\int_0^\pi \cos t \, \mathrm{d}t$ pour obtenir un résultat numérique plus ou moins proche de zéro (à 10^{-10} près par exemple) sans pouvoir affirmer si le résultat est le nombre entier 0 — de façon exacte — ou au contraire est proche de zéro mais est non nul.

Un système formel transforme par une manipulation de symboles mathématiques l'intégrale $\int_0^\pi \cos t \, \mathrm{d}t$ en la formule $\sin \pi - \sin 0$ qui est ensuite évaluée de façon exacte en $0 - 0 = 0$. Cette méthode prouve donc $\int_0^\pi \cos t \, \mathrm{d}t = 0 \in \mathbb{N}$.

Cependant les transformations uniquement algébriques ont aussi des limites. La plupart des expressions manipulées par les systèmes formels sont des fractions rationnelles et l'expression a/a est automatiquement simplifiée en 1. Ce mode de calcul algébrique ne convient pas à la résolution des équations ; dans ce cadre, la solution de l'équation $ax = a$ est $x = a/a$ qui est simplifiée en $x = 1$ sans distinguer suivant la valeur de a par rapport à 0, valeur particulière pour laquelle tout nombre x est solution.

Les algorithmes programmés dans tout système de calcul formel reflètent uniquement les connaissances mathématiques des concepteurs du système. Un tel outil ne peut pas démontrer directement un nouveau théorème ; il permet simplement d'effectuer des calculs dont les méthodes sont bien connues mais qui sont trop longs pour être menés à bien « avec un papier et un crayon ».

1.6 Classes et classes normales

Les sections suivantes présentent la notion de classe d'objets fondamentale pour tout système de calcul formel. On suit ici la présentation qui en est faite dans la deuxième partie du premier chapitre de l'ouvrage Calcul formel : mode d'emploi. Exemples en Maple de Philippe Dumas, Claude Gomez, Bruno Salvy et Paul Zimmermann [DGSZ95].

Chaque fonction Sage s'applique à (et produit) une classe bien définie d'expressions. Reconnaître qu'une expression appartient à telle ou telle classe permet du même coup de savoir quelles fonctions on peut lui appliquer.

Un problème pour lequel cette reconnaissance est essentielle est celui de la simplification d'expressions. C'est autour de ce problème que sont définies les principales classes d'expressions des systèmes de calcul formel. En effet, dès qu'il est possible de déterminer si une expression appartenant à une classe est nulle ou non, il est possible d'effectuer des divisions dans cette classe. Autrement, tous les calculs faisant intervenir une division deviennent hasardeux.

Une classe est dite normale lorsque deux expressions représentant le même objet mathématique sont nécessairement identiques. Les classes élémentaires décrites ci-dessous sont des classes normales. Cependant, la représentation idéale n'est pas toujours la forme normale. Dans le cas des polynômes par exemple, la représentation développée est une forme normale, mais la représentation factorisée permet des calculs de PGCD bien plus rapides. Ce genre d'exemple amène les systèmes de calcul formel à un compromis. Un certain nombre de simplifications basiques, comme la réduction des rationnels ou la multiplication par zéro, sont effectuées automatiquement ; les autres réécritures sont laissées à l'initiative de l'utilisateur auquel des commandes spécialisées sont proposées.

Dans Sage, les principales fonctions permettant de réécrire des expressions sont `expand`, `combine`, `collect` et `simplify`. Pour bien utiliser ces fonctions, il faut savoir quel type de transformations elles effectuent et à quelle classe d'expressions ces transformations s'appliquent. Ainsi, l'usage aveugle de la fonction `simplify` peut conduire à des résultats faux. Des variantes de `simplify` permettent alors de préciser la simplification à effectuer.

1.7 Les classes élémentaires

Les *classes élémentaires* sont formées d'expressions sans variable, c'est-à-dire de constantes : entiers, rationnels, nombres flottants, booléens, résidus modulo p et nombres p-adiques.

- *Entiers*
  Dans un système de calcul formel les opérations sur des nombres entiers ou rationnels sont exactes.

  Exemple. Un calcul typique est celui de la factorielle de 100.

  ```
  sage: factorial(100)
  ```

  ```
  933262154439441526816992388562667004907159682643816\21\
  468592963895217599993229915608941463976156518286253\69\
  792082722375825118521091686400000000000000000000000
  ```

  De nombreuses fonctions s'appliquent aux entiers. Parmi celles qui sont spécifiques aux entiers, citons :
  – le reste modulo k : `n%k`

– la factorielle : $n!$ = `factorial(n)`
– les coefficients binomiaux : $\binom{n}{k}$ = `binomial(n,k)`

Exemple. FERMAT avait conjecturé que tous les nombres de la forme $2^{2^n}+1$ étaient premiers. Voici le premier exemple qui invalide sa conjecture :

```
sage: factor(2^(2^5)+1)
641 * 6700417
```

Du point de vue de la simplification, tous les entiers sont représentés en base dix (ou deux selon les systèmes), ce qui constitue une forme normale. L'égalité d'entiers est donc facile à tester.
Toute opération sur des entiers est immédiatement effectuée ; ainsi 2^2 n'est pas représentable en Sage, il est immédiatement transformé en 4. Cela signifie aussi qu'un nombre factorisé ne peut pas être représenté comme un entier, puisqu'alors il serait immédiatement développé. Dans l'exemple 1.7, le résultat est en réalité un produit de fonctions.
Cette bibliothèque `Integer` de calcul sur les entiers est propre à Sage. Par défaut Python utilise des entiers de type `int`. En général la conversion de l'un vers l'autre est automatique, mais il peut être nécessaire d'indiquer explicitement cette conversion par l'une ou l'autre de ces commandes :

```
sage: int(5) ; Integer(5)
```

- *Rationnels*
  La propriété de forme normale s'étend aux nombres rationnels. Non seulement les additions, multiplications et quotients sont immédiatement exécutés, mais en plus les fractions rationnelles sont toutes réduites.

  Exemple. Dans cet exemple, les factorielles sont d'abord évaluées, puis le rationnel obtenu est simplifié :

  ```
  sage: factorial(99) / factorial(100) - 1 / 50
  -1/100
  ```

- *Flottants*
  Les règles de simplification automatique sont moins systématiques pour les nombres approchés numériquement, appelés aussi nombres en virgule flottante, ou plus simplement flottants. Lorsqu'ils interviennent dans une somme, un produit ou un quotient faisant intervenir par ailleurs des rationnels, ils sont contagieux, c'est-à-dire que toute l'expression devient un nombre flottant.

  Exemple.

```
sage: 72/53-5/3*2.7
-3.14150943396227
```

De même, lorsque l'argument d'une fonction est un flottant, le résultat est alors un flottant :

Exemple.

```
sage: cos(1); cos(1.)
cos(1)
0.540302305868140
```

Pour les autres expressions, la fonction de base pour ces calculs est `numerical_approx` (ou son alias `n`) qui évalue numériquement une expression (tous les nombres sont transformés en flottants). Un argument optionnel permet de préciser le nombre de chiffres significatifs utilisés lors du calcul.

Exemple. Voici π avec 50 chiffres significatifs.

```
sage: pi.n(digits=50)  # N(pi,digits=10^6) aussi possible
3.1415926535897932384626433832795028841971693993751
```

Dans Sage, les flottants sont liés à leur précision : ainsi la valeur précédente est différente syntaxiquement de la valeur de π calculée avec dix chiffres significatifs. Compte tenu de cette restriction, les flottants renvoyés par `numerical_approx` sont sous forme normale.

- *Nombres complexes*
  Les nombres complexes existent dans tous les systèmes de calcul formel. En Sage, on note `I` ou `i` le nombre imaginaire i.
  Les commandes principales sont `Re`, `Im`, `abs` donnant respectivement la partie réelle, la partie imaginaire, le module.

```
sage: z = 3 * exp(I*pi/4)
sage: z.real(), z.imag(), z.abs().simplify_exp()
```

$$\frac{3}{2}\sqrt{2}, \quad \frac{3}{2}\sqrt{2}, \quad 3$$

Pour calculer un argument du complexe $z = x + iy$, on peut utiliser la fonction « `arctan2(y,x)=arctan2(z.im(), z.real())` ». Certes, il existe une fonction « `arg` » ; cependant, elle ne s'applique qu'aux objets du type `CC` :

```
sage: z = CC(1,2); z.arg()
1.10714871779409
```

- *Booléens*
  Les expressions booléennes forment aussi une classe élémentaire. Les deux formes normales sont `True` et `False`.

  Exemple.

  ```
  sage: a, b, c = 0, 2, 3
  sage: a == 1 or (b == 2 and c == 3)
  True
  ```

  Dans les tests et les boucles, les conditions composées à l'aide des opérateurs `or` et `and` sont évaluées de façon paresseuse de la gauche vers la droite. Ceci signifie que l'évaluation d'une condition composée par `or` se termine après le premier prédicat de valeur `True`, sans évaluer les termes placés plus à droite ; de même avec `and` et `False`. Ainsi le test ci-dessous décrit la divisibilité $a|b$ sur les entiers et ne provoque aucune erreur même si $a = 0$:

  ```
  sage: a=0; b=12; (a==0)and(b==0) or (a!=0)and(b%a==0)
  ```

  Par ailleurs l'opérateur `not` est prioritaire sur `and` qui est lui même prioritaire sur `or` ; et les tests d'égalité et de comparaison sont eux même prioritaires sur les opérateurs booléens. Les deux tests suivants sont donc identiques au précédent :

  ```
  ((a==0)and(b==0)) or  ((a!=0) and(b%a==0))
    a==0 and b==0   or not a==0 and b%a==0
  ```

  En outre Sage autorise les tests d'encadrement et d'égalités multiples de la même façon qu'ils sont écrits en mathématiques :

  $x \leq y < z \leq t$ codé par `x <= y < z <= t`
  $x = y = z \neq t$ `x == y == z != t`

  Ces tests sont généralement effectués directement, sauf pour les équations, dans ce cas la commande `bool` est nécessaire :

  ```
  sage: var('x y'); bool((x-y)*(x+y)==x^2-y^2)  # est True
  ```

  Enfin les opérateurs et, ou et ou-exclusif opérant bit à bit sur les entiers sont notés `|`, `&` et `^^`, et les opérateurs `<<` et `>>` correspondent au décalage à gauche ou à droite des bits.

- *Classes issues de l'arithmétique*
  Les autres constantes formant une classe élémentaire munie d'une forme normale sont les résidus modulo p, avec pour fonction de réduction « `%` », et les nombres p-adiques.

La classe d'un nombre peut être explicitement déclarée en même temps que ce nombre, ainsi `ZZ(1)`, `QQ(1)`, `RR(1)` et `CC(1)` définissent respectivement l'entier $1 \in \mathbb{Z}$, le nombre rationnel $1 \in \mathbb{Q}$, et les approximations flottantes $1.0 \in \mathbb{R}$ et $1.0+0.0\mathrm{i} \in \mathbb{C}$. Réciproquement la fonction `parent` renvoie la classe d'un élément, et `isinstance(x,E)` teste si l'objet `x` est dans la classe `E` ; les résultats de ces tests sont `True` :

```
sage: isinstance(ZZ(5), Integer) ; isinstance(QQ(5), Rational)
```

De même la déclaration d'un entier modulaire m de l'anneau $\mathbb{Z}/n\mathbb{Z}$ ou du corps $\mathbb{Z}/p\mathbb{Z}$ lorsque l'entier p est premier s'obtient respectivement par `IntegerModRing(n)(m)` et `GF(p)(m)`.

On dispose des fonctions suivantes sur les objets des quatre premières classes (entiers, rationnels, flottants, complexes) :

Fonction	`Syntaxe`
les quatre opérations arithmétiques de base	`a+b, a-b, a/b, a*b`
la fonction puissance	`a^b`
la division entière	`a//b`
la partie entière	`floor(a)`
la valeur absolue, le module	`abs(a)`
la racine carrée	`sqrt(a)`
la racine n-ième	`a^(1/n)`
les fonctions usuelles	`sin cos tan cot arcsin`
	`sinh tanh log exp ...`

1.8 Autres classes à forme normale

À partir de constantes bien définies, des classes d'objets symboliques faisant intervenir des variables et admettant une forme normale peuvent être construites. Les plus importantes sont les matrices, les polynômes et fractions rationnelles, les développements limités et les nombres algébriques. Pour chacune de ces classes, nous indiquons les principales fonctions de réécriture.

- *Matrices*
  La forme normale d'une matrice est obtenue lorsque tous ses coefficients sont eux-mêmes sous forme normale. Sage effectue automatiquement cette mise sous forme normale des coefficients, comme le montre l'exemple suivant :

  ```
  sage: a = matrix(QQ, [[1,2,3],[2,4,8],[3,9,27]])
  ```

```
sage: I = identity_matrix(QQ, 3)
sage: (a^2 + I) * a^(-1)
[  -5 13/2  7/3]
[   7    1 25/3]
[   2 19/2   27]
```

Dans cet exemple, on précise le corps dans lequel il faut effectuer les calculs, à savoir le corps `QQ`$= \mathbb{Q}$.

- *Polynômes et fractions rationnelles*
  Les calculs sur les polynômes et les fractions rationnelles à une ou plusieurs indéterminées sont des opérations de base d'un système de calcul formel. Contrairement aux classes présentées jusqu'ici, il n'y a pas une bonne représentation des polynômes.
  Les fonctions qui permettent de réécrire un polynôme sous diverses formes sont résumées dans le tableau ci-dessous :

Polynôme `p`	$zx^2+x^2-(x^2+y^2)(ax-2by)+zy^2+y^2$
`p.expand().collect(x)`	$-ax^3-axy^2+2\,by^3+(2\,by+z+1)x^2+y^2z+y^2$
`p.collect(x).collect(y)`	$2\,bx^2y+2\,by^3-(ax-z-1)x^2-(ax-z-1)y^2$
`p.expand()`	$-ax^3-axy^2+2\,bx^2y+2\,by^3+x^2z+y^2z+x^2+y^2$
`p.factor()`	$-\left(x^2+y^2\right)(ax-2\,by-z-1)$
`p.factor_list()`	$\left[\left(x^2+y^2,1\right),(ax-2\,by-z-1,1),(-1,1)\right]$

Pour transformer des fractions rationnelles, on dispose des fonctions résumées dans le tableau suivant. La fonction `combine` permet de regrouper les termes qui ont même dénominateur. Quant à la fonction `partial_fraction`, elle effectue la décomposition en éléments simples dans $\mathbb{Q}$. Pour préciser le corps dans lequel il faut effectuer la

décomposition en éléments simples, on se reportera au chapitre 9.1.7.

Fraction rationnelle `r`	$\frac{x^3+x^2y+3\,x^2+3\,xy+2\,x+2\,y}{x^3+2\,x^2+xy+2\,y}$
`r.simplify_rational()`	$\frac{(x+1)y+x^2+x}{x^2+y}$
`r.factor()`	$\frac{(x+1)(x+y)}{x^2+y}$
`r.factor().expand()`	$\frac{x^2}{x^2+y}+\frac{xy}{x^2+y}+\frac{x}{x^2+y}+\frac{y}{x^2+y}$
Fraction rationnelle `r`	$\frac{(x-1)x}{x^2-7}+\frac{y^2}{x^2-7}+\frac{b}{a}+\frac{c}{a}+\frac{1}{x+1}$
`r.combine()`	$\frac{(x-1)x+y^2}{x^2-7}+\frac{b+c}{a}+\frac{1}{x+1}$
Fraction rationnelle `r`	$\frac{1}{(x^3+1)y^2}$
`r.partial_fraction(x)`	$\frac{-(x-2)}{3\,(x^2-x+1)y^2}+\frac{1}{3\,(x+1)y^2}$

- *Développements limités*
  Comme les matrices et les flottants, les développements limités ont une forme normale, mais celle-ci n'est pas produite automatiquement. La commande de réduction est `series`.

```
sage: f = cos(x).series(x == 0, 6); 1 / f
```

$$\frac{1}{1+(-\frac{1}{2})x^2+\frac{1}{24}x^4+\mathcal{O}(x^6)}$$

```
sage: (1 / f).series(x == 0, 6)
```

$$1+\tfrac{1}{2}x^2+\tfrac{5}{24}x^4+\mathcal{O}\left(x^6\right)$$

- *Nombres algébriques*
  Un nombre algébrique est défini comme racine d'un polynôme. Lorsque le degré du polynôme est plus grand que 4, il n'est pas possible de le résoudre explicitement en général. Cependant, de nombreux calculs sur ses racines peuvent être menés à bien sans autre information que le polynôme lui-même. Ces calculs sont détaillés au chapitre 8.

```
sage: k.<a> = NumberField(x^3 + x + 1); a^3; a^4+3*a
-a - 1
-a^2 + 2*a
```

2

Analyse et algèbre avec Sage

Ce chapitre présente les fonctions de base utiles en analyse et en algèbre à travers des exemples simples. Les lycéens et étudiants trouveront matière à remplacer *le crayon et le papier* par *le clavier et l'écran* tout en relevant le même défi intellectuel de compréhension des mathématiques.

Cet exposé des principales commandes de calcul avec Sage se veut accessible aux élèves de lycée ; figurent également des compléments signalés par un astérisque (*) pour les étudiants de première année de licence. On renvoie aux autres chapitres pour plus de détails.

2.1 Simplification d'expressions symboliques

2.1.1 Expressions symboliques et fonctions symboliques

Un système de calcul formel effectue les calculs d'analyse à partir d'expressions symboliques combinant des nombres, des variables symboliques, les quatre opérations, et des fonctions usuelles comme `sqrt`, `exp`, `log`, `sin`, `cos`, etc. L'opération la plus commune consiste à évaluer une expression pour une valeur d'un ou plusieurs paramètres. Avec Sage, la méthode `subs` — qui peut être sous-entendue — effectue ces manipulations :

```
sage: var ('a,x')  ; y = cos(x+a)*(x+1)    ; y
sage: y.subs(a=-x) ; y.subs(x=pi/2,a=pi/3) ; y.subs(x=0.5,a=2.3)
sage: y(a=-x)      ; y(x=pi/2,a=pi/3)      ;      y(x=0.5,a=2.3)
```

Par rapport à la notation mathématique usuelle $x \mapsto f(x)$, le nom de la variable substituée doit être indiqué. La substitution avec plusieurs paramètres

est faite de façon parallèle, alors que plusieurs substitutions effectuent des réécritures séquentielles, comme le montrent les deux tests ci-dessous :

```
sage: var('x,y,z') ; q = x*y+y*z+z*x
sage: bool (q(x=y,y=z,z=x)==q), bool(q(z=y)(y=x) == 3*x^2)
```

Notons que pour substituer une expression à une autre, on dispose de la fonction `subs_expr` :

```
sage: var('y z'); f = x^3+y^2+z; f.subs_expr(x^3==y^2, z==1)
2*y^2 + 1
```

Par ailleurs Sage permet de définir des fonctions symboliques pour la manipulation d'expressions :

```
sage: f(x)=(2*x+1)^3 ; f(-3)
-125
sage: f(x).expand()
8*x^3 + 12*x^2 + 6*x + 1
```

À noter que ces fonctions symboliques n'ont pas le même rôle que les fonctions Python qui sont décrites dans le chapitre 3. Une fonction symbolique possède les mêmes attributs qu'une expression symbolique, ce qui n'est pas le cas d'une fonction Python ; par exemple, la méthode `expand` ne s'applique pas à ces dernières :

```
sage: def f(x):
....:     return (2*x+1)^3
sage: f.expand()
AttributeError: 'function' object has no attribute 'expand'
```

Ainsi une fonction symbolique n'est autre qu'une expression que l'on peut appeler et où l'ordre des variables est fixé. Pour convertir une expression symbolique en fonction symbolique, on utilise soit la méthode `function`, soit la syntaxe déjà mentionnée :

```
sage: var('y'); u = sin(x) + x*cos(y)
sage: v = u.function(x, y); v
(x, y) |--> x*cos(y) + sin(x)
sage: w(x, y) = u; w
(x, y) |--> x*cos(y) + sin(x)
```

2.1.2 Expressions complexes et simplification

Les classes d'expressions présentées aux paragraphes 1.7 et 1.8 partagent la propriété d'avoir une procédure de décision pour la nullité. C'est-à-dire que pour toutes ces classes un programme peut déterminer si une expression

donnée est nulle ou non. Dans de nombreux cas, cette décision se fait par réduction à la forme normale : l'expression est nulle si et seulement si sa forme normale est le symbole 0.

Malheureusement, toutes les classes d'expressions n'admettent pas une forme normale. Pire encore, pour certaines classes il est impossible de prouver la nullité d'une expression en temps fini. Un exemple d'une telle classe est fourni par les expressions composées à partir des rationnels, des nombres π et $\log 2$ et d'une variable, par utilisation répétée de l'addition, de la soustraction, du produit, de l'exponentielle et du sinus. Bien sûr, une utilisation répétée de `numerical_approx` en augmentant la précision permet souvent de conjecturer si une expression particulière est nulle ou non ; mais Richardson a montré qu'il est impossible d'écrire un programme prenant en argument une expression de cette classe et donnant au bout d'un temps fini le résultat vrai si celle-ci est nulle, et faux sinon.

C'est dans ces classes que se pose avec le plus d'acuité le problème de la simplification. Sans forme normale, les systèmes ne peuvent que donner un certain nombre de fonctions de réécriture avec lesquelles l'utilisateur doit jongler pour parvenir à un résultat. Pour y voir plus clair dans cette jungle, il faut là encore distinguer plusieurs sous-classes, savoir quelles fonctions s'appliquent et quelles transformations sont effectuées.

Constantes

Comme dit précédemment, les calculs se font avec des nombres entiers ou rationnels exacts et avec des constantes symboliques (qui ne sont pas des approximations flottantes).

Les constantes les plus simples sont les rationnels, le nombre π noté `pi`, la base e des logarithmes népériens notée `e`, le nombre imaginaire i noté `I` ou `i`, la constante d'Euler γ notée `euler_gamma` et le nombre d'or φ noté `golden_ratio`.

Ces constantes sont relativement bien connues du système. Pour la classe simple des polynômes en π et e, aucun algorithme de forme normale n'est connu : à ce jour on ignore s'il existe un tel polynôme non trivial qui vaille zéro.

En utilisation interactive, une bonne façon de traiter ces constantes dans des simplifications compliquées est de les remplacer toutes sauf i par des variables et d'utiliser les procédures de forme normale des fractions rationnelles.

Ceci revient à faire l'hypothèse que toutes ces constantes sont algébriquement indépendantes. Cette remarque se généralise à des constantes plus complexes comme $\log 2$, $\exp(\pi + \log 3)$,... mais il faut alors être sûr que celles-ci ne sont pas trivialement dépendantes.

Fonctions mathématiques usuelles

La plupart des fonctions mathématiques se retrouvent en Sage, en particulier les fonctions trigonométriques, le logarithme et l'exponentielle.

Fonction	Syntaxe
Fontions exponentielle et logarithme	`exp, log`
Fontion logarithme de base a	`log(x, a)`
Fonctions trigonométriques	`sin, cos, tan`
Fonctions trigonométriques réciproques	`arcsin, arccos, arctan`
Fonctions hyperboliques	`sinh, cosh, tanh`
Fonctions hyperboliques réciproques	`arcsinh, arccosh, arctanh`
Partie entière, ...	`floor, ceil, trunc, round`
Racine carrée et n-ième	`sqrt, nth_root`

La simplification de telles fonctions est cruciale. Pour simplifier une expression ou une fonction symbolique, on dispose de la commande `simplify` :

```
sage: (x^x/x).simplify()
x^(x - 1)
```

Cependant, pour des simplifications plus subtiles, on doit préciser le type de simplification attendue :

```
sage: f = (e^x-1)/(1+e^(x/2)); f.simplify_exp()
e^(1/2*x) - 1
```

Pour simplifier des expressions trigonométriques, on utilise la commande `simplify_trig` :

```
sage: f = cos(x)^6 + sin(x)^6 + 3 * sin(x)^2 * cos(x)^2
sage: f.simplify_trig()
1
```

Pour linéariser (resp. anti-linéariser) une expression trigonométrique, on utilise `reduce_trig` (resp. `expand_trig`) :

```
sage: f = cos(x)^6; f.reduce_trig()
```

$$\frac{15}{32}\cos(2\,x) + \frac{3}{16}\cos(4\,x) + \frac{1}{32}\cos(6\,x) + \frac{5}{16}$$

```
sage: f = sin(5 * x); f.expand_trig()
```

$$\sin(x)^5 - 10\sin(x)^3\cos(x)^2 + 5\sin(x)\cos(x)^4$$

Le tableau suivant résume les fonctions utiles pour simplifier des expressions trigonométriques :

Type de simplification	Syntaxe
Simplification	`simplify_trig`
Linéarisation	`reduce_trig`
Anti-linéarisation	`expand_trig`

On peut également simplifier des expressions faisant intervenir des factorielles :

```
sage: n = var('n'); f = factorial(n+1)/factorial(n)
sage: f.simplify_factorial()
n + 1
```

La fonction `simplify_rational`, quant à elle, cherche à simplifier une fraction rationnelle en développant ses membres. Pour simplifier des racines carrées, des logarithmes ou des exponentielles, on utilise `simplify_radical` :

```
sage: f = sqrt(x^2); f.simplify_radical()
abs(x)
sage: f = log(x*y); f.simplify_radical()
log(x) + log(y)
```

La commande `simplify_full` applique (dans l'ordre) les fonctions suivantes : `simplify_factorial`, `simplify_trig`, `simplify_rational` et `simplify_radical`.

Tout ce qui concerne le classique tableau de variation de la fonction (calcul des dérivées, des asymptotes, des extrema, recherche des zéros et tracé de la courbe) peut être facilement réalisé à l'aide d'un système de calcul formel. Les principales opérations Sage qui s'appliquent à une fonction sont présentées dans la section 2.3.

2.1.3 Hypothèses sur une variable symbolique

Les variables non affectées posent problème lors des calculs. Un cas typique est la simplification de l'expression $\sqrt{x^2}$. Pour simplifier de telles expressions, la bonne solution consiste à utiliser la fonction `assume` qui permet de préciser les propriétés d'une variable. Si on souhaite annuler cette hypothèse, on emploie l'instruction `forget` :

```
sage: assume(x > 0); bool(sqrt(x^2) == x)
True
```

```
sage: forget(x > 0); bool(sqrt(x^2) == x)
False
sage: var('n'); assume(n, 'integer'); sin(n*pi).simplify()
0
```

2.2 Équations

Un leitmotiv du calcul formel est la manipulation d'objets définis par des équations, sans passer par la résolution de celles-ci.

Ainsi, une fonction définie par une équation différentielle linéaire et des conditions initiales est parfaitement précisée. L'ensemble des solutions d'équations différentielles linéaires est clos par addition et produit (entre autres) et forme ainsi une importante classe où l'on peut décider de la nullité. En revanche, si l'on résout une telle équation, la solution, privée de son équation de définition, tombe dans une classe plus grande où bien peu est décidable. Le chapitre 6 reviendra plus en détail sur ces considérations. Cependant, dans certains cas, surtout en utilisation interactive, il est utile de chercher une solution explicite, par exemple pour passer à une application numérique. Les principales fonctions de résolution sont résumées ci-dessous.

Type de résolution	Syntaxe
Résolution symbolique	`solve`
Résolution (avec multiplicités)	`roots`
Résolution numérique	`find_root`
Résolution d'équations différentielles	`desolve`
Résolution de récurrences	`rsolve`
Résolution d'équations linéaires	`right_solve, left_solve`

Pour extraire le membre de gauche (resp. de droite) d'une équation, on dispose de la fonction `lhs` (resp. `rhs`). Donnons quelques exemples d'utilisation de la fonction `solve`. Soit à résoudre l'équation d'inconnue z et de paramètre φ suivante :

$$z^2 - \frac{2}{\cos\varphi} z + \frac{5}{\cos^2\varphi} - 4 = 0, \qquad \text{avec } \varphi \in]-\tfrac{\pi}{2}, \tfrac{\pi}{2}[.$$

```
sage: z, phi = var('z, phi')
sage: solve(z**2 -2 / cos(phi) * z + 5 / cos(phi) ** 2 - 4, z)
```

$$\left[z = -\frac{2\sqrt{\cos(\varphi)^2 - 1} - 1}{\cos(\varphi)}, z = \frac{2\sqrt{\cos(\varphi)^2 - 1} + 1}{\cos(\varphi)}\right]$$

Soit à résoudre l'équation $y^6 = y$.

```
sage: var('y'); solve(y^6==y, y)
[y == e^(2/5*I*pi), y == e^(4/5*I*pi), y == e^(-4/5*I*pi),
 y == e^(-2/5*I*pi), y == 1, y == 0]
```

Les solutions peuvent être renvoyées sous la forme d'un objet du type dictionnaire (cf. § 3.2.9) :

```
sage: solve(x^2-1, x, solution_dict=True)
[{x: -1}, {x: 1}]
```

La commande `solve` permet également de résoudre des systèmes :

```
sage: solve([x+y == 3, 2*x+2*y == 6],x,y)
[[x == -r1 + 3, y == r1]]
```

Ce système linéaire étant indéterminé, l'inconnue secondaire qui permet de paramétrer l'ensemble des solutions est un réel désigné par `r1`, `r2`, etc. Si le paramètre est implicitement un entier, il est désigné sous la forme `z1`, `z2`, etc. :

```
sage: solve([cos(x)*sin(x) == 1/2, x+y == 0], x, y)
[[x == 1/4*pi + pi*z30, y == -1/4*pi - pi*z30]]
```

Enfin, la fonction `solve` peut être utilisée pour résoudre des inéquations :

```
sage: solve(x^2+x-1>0,x)
[[x < -1/2*sqrt(5) - 1/2], [x > 1/2*sqrt(5) - 1/2]]
```

Il arrive que les solutions d'un système renvoyées par la fonction `solve` soient sous formes de flottants. Soit, par exemple, à résoudre dans $\mathbb{C}^3$ le système suivant :

$$\begin{cases} x^2yz &= 18, \\ xy^3z &= 24, \\ xyz^4 &= 6. \end{cases}$$

```
sage: x, y, z = var('x, y, z')
sage: solve([x^2 * y * z == 18, x * y^3 * z == 24,\
             x * y * z^4 == 6], x, y, z)

[[x == 3, y == 2, z == 1],
[x == (1.33721506733-2.68548987407*I)],
        ...]
```

Sage renvoie ici 17 solutions, dont un triplet d'entiers et 16 triplets de solutions complexes approchées. Pour obtenir une solution symbolique, on se reportera au chapitre 9.

Pour effectuer des résolutions numériques d'équations, on utilise la fonction `find_root` qui prend en argument une fonction d'une variable ou une égalité symbolique, et les bornes de l'intervalle dans lequel il faut chercher une solution.

```
sage: expr = sin(x) + sin(2 * x) + sin(3 * x)
sage: solve(expr, x)
[sin(3*x) == -sin(2*x) - sin(x)]
```

Sage ne trouve pas de solution symbolique à cette équation. Deux choix sont alors possibles. Soit on passe à une résolution numérique :

```
sage: find_root(expr, 0.1, pi)
2.0943951023931957
```

Soit on transforme au préalable l'expression :

```
sage: f = expr.simplify_trig(); f
2*(2*cos(x)^2 + cos(x))*sin(x)
sage: solve(f, x)
[x == 0, x == 2/3*pi, x == 1/2*pi]
```

Enfin la fonction `roots` permet d'obtenir les solutions exactes d'une équation, avec leur multiplicité. On peut préciser en outre l'anneau dans lequel on souhaite effectuer la résolution ; si on choisit `RR` $\approx \mathbb{R}$ ou `CC` $\approx \mathbb{C}$ on obtient les résultats sous forme de nombres à virgule flottante : la méthode de résolution sous-jacente n'est pas pour autant une méthode numérique, comme c'est le cas lorsqu'on utilise `find_roots`.

Considérons l'équation du troisième degré $x^3+2\,x+1=0$. Cette équation est de discriminant négatif, donc elle possède une racine réelle et deux racines complexes, que l'on peut obtenir grâce à la fonction `roots` :

```
sage: (x^3+2*x+1).roots(x)
```

$$\left[\left(-\tfrac{1}{2}\left(I\sqrt{3}+1\right)\left(\tfrac{1}{18}\sqrt{3}\sqrt{59}-\tfrac{1}{2}\right)^{\left(\frac{1}{3}\right)}+\frac{-\left(I\sqrt{3}-1\right)}{3\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}},1\right),\right.$$
$$\left(-\tfrac{1}{2}\left(-I\sqrt{3}+1\right)\left(\tfrac{1}{18}\sqrt{3}\sqrt{59}-\tfrac{1}{2}\right)^{\left(\frac{1}{3}\right)}+\frac{-\left(-I\sqrt{3}-1\right)}{3\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}},1\right),$$
$$\left.\left(\left(\tfrac{1}{18}\sqrt{3}\sqrt{59}-\tfrac{1}{2}\right)^{\left(\frac{1}{3}\right)}+\frac{-2}{3\left(\frac{1}{18}\sqrt{3}\sqrt{59}-\frac{1}{2}\right)^{\left(\frac{1}{3}\right)}},1\right)\right]$$

```
sage: (x^3+2*x+1).roots(x, ring=RR)
```

$$[(-0.453397651516404, 1)]$$

```
sage: (x^3+2*x+1).roots(x, ring=CC)
```

$$[(-0.453397651516404, 1),\ (0.226698825758202 - 1.46771150871022*I, 1),$$
$$(0.226698825758202 + 1.46771150871022*I, 1)]$$

2.3 Analyse

Dans cette section, nous présentons succinctement les fonctions couramment utiles en analyse réelle. Pour une utilisation avancée ou des compléments, on renvoie aux chapitres suivants notamment ceux qui traitent de l'intégration numérique (ch. 6), de la résolution des équations non linéaires (ch. 7), et des équations différentielles (ch. 11).

2.3.1 Sommes et produits

Pour calculer des sommes symboliques on utilise la fonction `sum`. Calculons par exemple la somme des n premiers entiers non nuls :

```
sage: var('k n'); sum(k, k, 1, n).factor()
```

$$\frac{1}{2}\,(n+1)n$$

La fonction `sum` permet d'effectuer des simplifications à partir du binôme de Newton :

```
sage: var('n y'); sum(binomial(n,k) * x^k * y^(n-k), k, 0, n)
```

$$(x+y)^n$$

Voici d'autres exemples, dont la somme des cardinaux des parties d'un ensemble à n éléments :

```
sage: var('k n'); sum(binomial(n,k), k, 0, n),\
                  sum(k * binomial(n, k), k, 0, n),\
                  sum((-1)^k*binomial(n,k), k, 0, n)
```

$$\left(2^n, n2^{(n-1)}, 0\right)$$

Enfin, quelques exemples de sommes géométriques :

```
sage: var('a q'); sum(a*q^k, k, 0, n)
```

$$\frac{aq^{(n+1)} - a}{q-1}$$

Pour calculer la série correspondante, il faut préciser que le module de la raison est inférieur à 1 :

```
sage: assume(abs(q) < 1)
sage: sum(a*q^k, k, 0, infinity)
```

$$-\frac{a}{(q-1)}$$

```
sage: forget(); assume(q > 1); sum(a*q^k, k, 0, infinity)
ValueError: Sum is divergent.
```

Exercice 1. (*Un calcul de somme par récurrence*) Calculer sans utiliser l'algorithme de Sage la somme des puissances p-ièmes des n premiers entiers :

$$S_n(p) = \sum_{k=0}^{n} k^p.$$

Pour calculer cette somme, on peut utiliser la formule de récurrence suivante :

$$S_n(p) = \frac{1}{p+1}\left((n+1)^{p+1} - \sum_{k=0}^{p-1}\binom{p+1}{k} S_n(k)\right).$$

Cette relation de récurrence s'établit facilement en calculant de deux manières la somme télescopique $\sum_{0\leqslant k\leqslant n} (k+1)^p - k^p$.

2.3.2 Limites

Pour calculer une limite, on utilise la commande `limit` ou son alias `lim`. Soit à calculer les limites suivantes :

$$a)\quad \lim_{x\to 8} \frac{\sqrt[3]{x}-2}{\sqrt[3]{x+19}-3}\ ; \qquad b)\quad \lim_{x\to \frac{\pi}{4}} \frac{\cos\left(\frac{\pi}{4}-x\right)-\tan x}{1-\sin\left(\frac{\pi}{4}+x\right)}.$$

```
sage: limit((x**(1/3) -2) / ((x + 19)**(1/3) - 3), x = 8)
9/4
sage: f(x) = (cos(pi/4-x)-tan(x))/(1-sin(pi/4 + x))
sage: limit(f(x), x = pi/4)
Infinity
```

La dernière réponse indique que l'une des limites à gauche ou à droite est infinie. Pour préciser le résultat, on étudie les limites à gauche et à droite, en utilisant l'option `dir` :

```
sage: limit(f(x), x = pi/4, dir='minus')
+Infinity
sage: limit(f(x), x = pi/4, dir='plus')
-Infinity
```

2.3.3 Suites

Les fonctions précédentes permettent d'effectuer des études de suites. On donne un exemple d'étude de croissance comparée entre une suite exponentielle et une suite géométrique.

Exemple. (*Une étude de suite*) On considère la suite $u_n = \frac{n^{100}}{100^n}$. Calculer les 10 premiers termes de la suite. Quelle est la monotonie de la suite ? Quelle est la limite de la suite ? À partir de quel rang a-t-on $u_n \in \left]0, 10^{-8}\right[$?

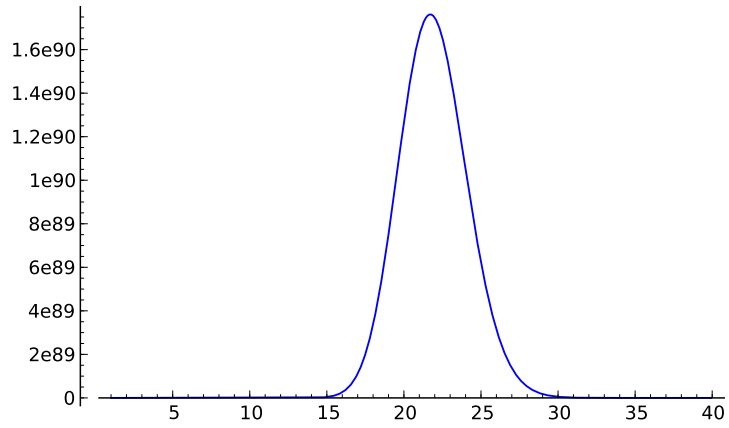


FIG. 2.1 – Graphe de $x \mapsto x^{100}/100^x$.

1. Pour définir le terme de la suite u_n, on utilise une fonction symbolique. On effectue le calcul des 10 premiers termes « à la main » (en attendant d'avoir vu les boucles au chapitre 3) :

```
sage: u(n) = n^100 / 100.^n
sage: u(1);u(2);u(3);u(4);u(5);u(6);u(7);u(8);u(9);u(10)
0.0100000000000000
1.26765060022823e26
5.15377520732011e41
1.60693804425899e52
7.88860905221012e59
6.53318623500071e65
3.23447650962476e70
2.03703597633449e74
2.65613988875875e77
1.00000000000000e80
```

   On pourrait en conclure hâtivement que u_n tend vers l'infini...

2. Pour avoir une idée de la monotonie, on peut tracer la fonction à l'aide de laquelle on a défini la suite u_n (cf. Fig. 2.1).

```
sage: plot(u(x), x, 1, 40)
```

   On conjecture que la suite va décroître à partir du rang 22.

```
sage: v(x) = diff(u(x), x); sol = solve(v(x) == 0, x); sol
[x == 0, x == (268850/12381)]
sage: floor(sol[1].rhs())
21
```

   La suite est donc croissante jusqu'au rang 21, puis décroissante à partir du rang 22.

3. On effectue ensuite le calcul de la limite :

```
sage: limit(u(n), n=infinity)
0
sage: n0 = find_root(u(n) - 1e-8 == 0, 22, 1000); n0
105.07496210187252
```

La suite étant décroissante à partir du rang 22, on en déduit qu'à partir du rang 106, la suite reste confinée à l'intervalle $]0, 10^{-8}[$.

2.3.4 Développements limités (*)

Pour calculer un développement limité d'ordre n au sens fort[1] en x_0, on dispose de la fonction `f(x).series(x==x0, n)`. Si l'on ne s'intéresse qu'à la partie régulière du développement limité à l'ordre n (au sens faible), on peut aussi utiliser la fonction `taylor(f(x), x, x0, n)`.

Déterminons le développement limité des fonctions suivantes :

a) $(1+\arctan x)^{\frac{1}{x}}$ à l'ordre 3, en $x_0 = 0$;

b) $\ln(2\sin x)$ à l'ordre 3, en $x_0 = \frac{\pi}{6}$.

```
sage: taylor((1+arctan(x))**(1/x), x, 0, 3)
```

$$\tfrac{1}{16}\,x^3 e + \tfrac{1}{8}\,x^2 e - \tfrac{1}{2}\,x e + e$$

```
sage: (ln(2* sin(x))).series(x==pi/6, 3)
```

$$(\sqrt{3})(-\tfrac{1}{6}\,\pi + x) + (-2)(-\tfrac{1}{6}\,\pi + x)^2 + \mathcal{O}\left(-\tfrac{1}{216}\,(\pi - 6\,x)^3\right)$$

Pour extraire la partie régulière d'un développement limité obtenu à l'aide de `series`, on utilise la fonction `truncate` :

```
sage: (ln(2* sin(x))).series(x==pi/6, 3).truncate()
```

$$-\tfrac{1}{18}\,(\pi - 6\,x)^2 - \tfrac{1}{6}\,(\pi - 6\,x)\sqrt{3}$$

La commande `taylor` permet également d'obtenir des développements asymptotiques. Par exemple, pour obtenir un équivalent au voisinage de $+\infty$ de la fonction $(x^3+x)^{\frac{1}{3}} - (x^3-x)^{\frac{1}{3}}$:

```
sage: taylor((x**3+x)**(1/3) - (x**3-x)**(1/3), x, infinity, 2)
2/3/x
```

[1] On appelle développement limité en 0 *au sens fort* une égalité de la forme $f(x) = P(x) + \mathcal{O}(x^{n+1})$ avec P polynôme de degré au plus n, par opposition à la notion de développement limité *au sens faible* qui désigne une égalité de la forme $f(x) = P(x) + o(x^n)$.

Exercice 2. (*Un calcul symbolique de limite*) Soit f de classe $\mathcal{C}^3$ au voisinage de $a \in \mathbb{R}$. Calculer

$$\lim_{h\to 0} \frac{1}{h^3}\left(f(a+3h) - 3f(a+2h) + 3f(a+h) - f(a)\right)$$

Généralisation ?

Exemple (*)**.** (*La formule de Machin*) Montrer la formule suivante :

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}.$$

C'est à l'aide de cette formule et du développement d'arctan en série entière, que l'astronome John Machin *(1680-1752)* calcula cent décimales de π en 1706. Déduire de cette formule une valeur approchée de π en utilisant sa méthode.

On commence par remarquer que $4\arctan\frac{1}{5}$ et $\frac{\pi}{4} + \arctan\frac{1}{239}$ ont même tangente :

```
sage: tan(4*arctan(1/5)).simplify_trig()
120/119
sage: tan(pi/4+arctan(1/239)).simplify_trig()
120/119
```

Or les réels $4\arctan\frac{1}{5}$ et $\frac{\pi}{4} + \arctan\frac{1}{239}$ appartiennent tous les deux à l'intervalle ouvert $]0, \pi[$, donc ils sont égaux. Pour obtenir une valeur approchée de π, on peut procéder de la manière suivante :

```
sage: f = arctan(x).series(x, 10); f
1*x + (-1/3)*x^3 + 1/5*x^5 + (-1/7)*x^7 + 1/9*x^9 + Order(x^10)
sage: (16*f.subs(x==1/5) - 4*f.subs(x==1/239)).n(); pi.n()
3.14159268240440
3.14159265358979
```

Exercice 3 (*)**.** (*Une formule due à Gauss*) La formule suivante nécessite le calcul de 20 pages de tables de factorisations dans l'édition des œuvres de Gauss (cf. *Werke*, ed. Köngl. Ges. de Wiss. Göttingen, vol. 2, p. 477-502) :

$$\frac{\pi}{4} = 12\arctan\frac{1}{38} + 20\arctan\frac{1}{57} + 7\arctan\frac{1}{239} + 24\arctan\frac{1}{268}.$$

1. On pose $\theta = 12\arctan\frac{1}{38} + 20\arctan\frac{1}{57} + 7\arctan\frac{1}{239} + 24\arctan\frac{1}{268}$. Vérifier à l'aide de Sage, que $\tan\theta = 1$.
2. Prouver l'inégalité : $\forall x \in \left[0, \frac{\pi}{4}\right], \quad x \leqslant \tan x \leqslant \frac{4}{\pi}x$. En déduire la formule de Gauss.
3. En approchant la fonction tan par son polynôme de Taylor d'ordre 21 en 0, donner une nouvelle approximation de π.

2.3.5 Séries (*)

On peut utiliser les commandes précédentes pour effectuer des calculs sur les séries. Donnons quelques exemples.

Exemple. (*Calcul de la somme de séries de Riemann*)

```
sage: var('k'); sum(1/k^2, k, 1, infinity),\
                sum(1/k^4, k, 1, infinity),\
                sum(1/k^5, k, 1, infinity)
```

$$\left(\frac{1}{6}\,\pi^2, \frac{1}{90}\,\pi^4, \zeta\left(5\right)\right)$$

Exemple. (*Une formule due à Ramanujan*) En utilisant la somme partielle des 12 premiers termes de la série suivante, donnons une approximation de π et comparons-la avec la valeur donnée par Sage.

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{+\infty} \frac{(4k)! \cdot (1103 + 26390\,k)}{(k!)^4 \cdot 396^{4k}}.$$

```
sage: s = 2*sqrt(2)/9801*(sum((factorial(4*k))*(1103+26390*k)\
     /((factorial(k)) ^ 4 * 396 ^ (4 * k)) for k in (0..11)))
sage: (1 / s).n(digits=100); pi.n(digits=100)
3.1415926535897932384626433832795028841971693993751058209749445
   9230781640628620899862803482534212143 2
3.1415926535897932384626433832795028841971693993751058209749445
   9230781640628620899862803482534211706 8
sage: print "%e" % (pi - 1 / s).n(digits=100)
-4.364154e-96
```

On remarque que la somme partielle des 12 premiers termes donne déjà 95 décimales significatives de π !

Exemple. (*Convergence d'une série*) Soit à étudier la nature de la série

$$\sum_{n\geqslant 0} \sin\left(\pi\sqrt{4\,n^2+1}\right).$$

Pour effectuer un développement asymptotique du terme général, on utilise la 2π-périodicité de la fonction sinus pour que l'argument du sinus tende vers 0 :

$$u_n = \sin\left(\pi\sqrt{4\,n^2+1}\right) = \sin\left[\pi\left(\sqrt{4\,n^2+1} - 2n\right)\right].$$

On peut alors appliquer la fonction `taylor` à cette nouvelle expression du terme général :

```
sage: var('n'); u = sin(pi*(sqrt(4*n^2+1)-2*n))
sage: taylor(u, n, infinity, 3)
```

$$\frac{\pi}{4\,n} + \frac{-(6\,\pi + \pi^3)}{384\,n^3}$$

On en déduit $u_n \sim \frac{\pi}{4\,n}$. Donc, d'après les règles de comparaison aux séries de Riemann, la série $\sum_{n \geqslant 0} u_n$ diverge.

Exercice 4. (*Développement asymptotique d'une suite*) Il est aisé de montrer (en utilisant par exemple le théorème de la bijection monotone) que pour tout $n \in \mathbb{N}$, l'équation $\tan x = x$ admet une et une seule solution, notée x_n, dans l'intervalle $[n\pi, n\pi + \frac{\pi}{2}[$.
Donner un développement asymptotique de x_n à l'ordre 6 en $+\infty$.

2.3.6 Dérivation

La fonction `derivative` (qui a pour alias `diff`) permet de dériver une expression symbolique ou une fonction symbolique.

```
sage: diff(sin(x^2), x)
2*x*cos(x^2)
sage: function('f',x); function('g',x); diff(f(g(x)), x)
D[0](f)(g(x))*D[0](g)(x)
sage: diff(ln(f(x)), x)
D[0](f)(x)/f(x)
```

2.3.7 Dérivées partielles (*)

La commande `diff` permet également de calculer des dérivées n-ièmes ou des dérivées partielles.

```
sage: f(x,y) = x*y + sin(x^2) + e^(-x); derivative(f, x)
(x, y) |--> 2*x*cos(x^2) + y - e^(-x)
sage: derivative(f, y)
(x, y) |--> x
```

Exemple. Soit à vérifier que la fonction suivante est harmonique[2] :
$f(x_1, x_2) = \frac{1}{2}\ln(x_1^2 + x_2^2)$ pour tout $(x_1, x_2) \neq (0, 0)$.

```
sage: var('x y'); f = ln(x**2+y**2) / 2
sage: delta = diff(f,x,2)+diff(f,y,2)
sage: delta.simplify_full()
0
```

Exercice 5. (*Expression du Laplacien en coordonnées polaires*) Soit l'ouvert $U = \mathbb{R}^2 \setminus \{(0,0)\}$, et f une fonction de classe $\mathcal{C}^2$ définie U. On considère la fonction $F(\rho, \theta) = f(\rho\cos\theta, \rho\sin\theta)$. Calculer $\dfrac{\partial^2 F}{\partial \rho^2} + \dfrac{1}{\rho^2}\dfrac{\partial^2 F}{\partial \theta^2} + \dfrac{1}{\rho}\dfrac{\partial F}{\partial \rho}$. En déduire l'expression du Laplacien $\Delta f = \partial_1^2 f + \partial_2^2 f$ en coordonnées polaires.

[2] Une fonction f est dite *harmonique* lorsque son Laplacien $\Delta f = \partial_1^2 f + \partial_2^2 f$ est nul.

Exercice 6. (*Un contre-exemple dû à Peano au théorème de Schwarz*) Soit f l'application de $\mathbb{R}^2$ dans $\mathbb{R}$ définie par :

$$f(x,y) = \begin{cases} xy\frac{x^2-y^2}{x^2+y^2} & \text{si } (x,y) \neq (0,0), \\ 0 & \text{si } (x,y) = (0,0). \end{cases}$$

A-t-on $\partial_1\partial_2 f(0,0) = \partial_2\partial_1 f(0,0)$?

2.3.8 Intégration

Pour calculer une primitive ou une intégrale, on utilise la fonction `integrate` (ou son alias `integral`) :

```
sage: sin(x).integral(x, 0, pi/2)
1
sage: integrate(1/(1+x^2), x)
arctan(x)
sage: integrate(1/(1+x^2), x, -infinity, infinity)
pi
sage: integrate(exp(-x**2), x, 0, infinity)
1/2*sqrt(pi)
sage: integrate(exp(-x), x, -infinity, infinity)
ValueError: Integral is divergent.
```

Pour effectuer une intégration numérique sur un intervalle, on dispose de la fonction `integral_numerical` qui renvoie un *tuple* à deux éléments, dont la première composante donne une valeur approchée de l'intégrale, et la deuxième une estimation de l'erreur effectuée.

```
sage: integral_numerical(sin(x)/x, 0, 1)
(0.94608307036718298, 1.0503632079297087e-14)
sage: g = integrate(exp(-x**2), x, 0, infinity); g, g.n()
(1/2*sqrt(pi), 0.886226925452758)
sage: approx = integral_numerical(exp(-x**2), 0, infinity)
sage: approx; approx[0]-g.n()
(0.88622692545275683, 1.7147744360127691e-08)
-1.11022302462516e-15
```

Exercice 7 (*)**.** (*La formule BBP*) On cherche à établir par un calcul symbolique la formule BBP (ou Bailey-Borwein-Plouffe) ; cette permet de calculer le n-ième chiffre après la virgule de π en base 2 (ou 16) sans avoir à en calculer les précédents, et en utilisant très peu de mémoire et de temps. Pour $N \in \mathbb{N}$, on pose $S_N = \sum_{n=0}^{N} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right)\left(\frac{1}{16}\right)^n$.

1. Soit la fonction $f : t \longmapsto 4\sqrt{2} - 8t^3 - 4\sqrt{2}t^4 - 8t^5$. Pour $N \in \mathbb{N}$, exprimer en fonction de S_N l'intégrale suivante :
$$I_N = \int_0^{1/\sqrt{2}} f(t) \left(\sum_{n=0}^{N} t^{8n} \right) \mathrm{d}t.$$
2. Pour $N \in \mathbb{N}$, on pose $J = \displaystyle\int_0^{1/\sqrt{2}} \frac{f(t)}{1-t^8}\,\mathrm{d}t$. Montrer $\lim\limits_{N\to+\infty} S_N = J$.
3. Montrer la formule BBP :
$$\sum_{n=0}^{+\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n = \pi.$$
Cette formule remarquable a été obtenue le 19 septembre 1995 par Simon Plouffe en collaboration avec David Bailey et Peter Borwein. Grâce à une formule dérivée de la formule BBP, le 4 000 000 000 000 000$^{\text{e}}$ chiffre de π en base 2 a été obtenu en 2001.

2.3.9 Récapitulatif des fonctions utiles en analyse

On résume les fonctions décrites dans les sections précédentes :

Fonction	Syntaxe
Dérivation	`diff(f(x), x)`
Dérivée n-ième	`diff(f(x), x, n)`
Intégration	`integrate(f(x), x)`
Intégration numérique	`integral_numerical(f(x), a, b)`
Somme symbolique	`sum(f(i), i, imin, imax)`
Limite	`limit(f(x), x=a)`
Polynôme de Taylor	`taylor(f(x), x, a, n)`
Développement limité	`f.series(x==a, n)`
Tracé d'une courbe	`plot(f(x), x, a, b)`

2.4 Calcul matriciel (*)

Dans cette section, on décrit les fonctions de base utiles en algèbre linéaire : opérations sur les vecteurs, puis sur les matrices. Pour plus de détails, on renvoie au chapitre 10 pour le calcul matriciel symbolique et au chapitre 5 pour le calcul matriciel numérique.

2.4.1 Résolution de systèmes linéaires

Pour résoudre un système linéaire, on peut utiliser la fonction `solve` déjà rencontrée.

Exercice 8 (*)**.** (*Approximation polynomiale du sinus*) Déterminer le polynôme de degré au plus 5 qui réalise la meilleure approximation, au sens des moindres carrés, de la fonction sinus sur l'intervalle $[-\pi, \pi]$:

$$\alpha_5 = \min \left\{ \int_{-\pi}^{\pi} |\sin x - P(x)|^2 \, \mathrm{d}x \;\middle|\; P \in \mathbb{R}_5[X] \right\}.$$

2.4.2 Calcul vectoriel

Les fonctions de base utiles pour la manipulation des vecteurs sont résumées dans le tableau suivant :

Fonction	Syntaxe
Déclaration d'un vecteur	`vector`
Produit vectoriel	`cross_product`
Produit scalaire	`dot_product`
Norme d'un vecteur	`norm`

On peut se servir des fonctions précédentes pour traiter l'exercice suivant.

Exercice 9. (*Le problème de Gauss*) On considère un satellite en orbite autour de la Terre et on suppose que l'on connaît trois points de son orbite : A_1, A_2 et A_3. On souhaite déterminer à partir de ces trois points les paramètres de l'orbite de ce satellite.

On note O le centre de la Terre. Les points O, A_1, A_2 et A_3 sont évidemment situés dans un même plan, à savoir le plan de l'orbite du satellite. L'orbite du satellite est une ellipse dont O est un foyer. On peut choisir un repère $(O; \overrightarrow{\imath}, \overrightarrow{\jmath})$ de telle sorte que l'équation polaire de l'ellipse en coordonnées polaires soit dans ce repère $r = \frac{p}{1 - e\cos\theta}$ où e désigne l'excentricité de l'ellipse et p son paramètre. On notera $\overrightarrow{r_i} = \overrightarrow{OA_i}$ et $r_i = \|\overrightarrow{r_i}\|$ pour $i \in \{1; 2; 3\}$. On considère alors les trois vecteurs suivants qui se déduisent de la connaissance de A_1, A_2 et A_3 :

$$\begin{aligned}
\overrightarrow{D} &= \overrightarrow{r_1} \wedge \overrightarrow{r_2} + \overrightarrow{r_2} \wedge \overrightarrow{r_3} + \overrightarrow{r_3} \wedge \overrightarrow{r_1}, \\
\overrightarrow{S} &= (r_1 - r_3) \cdot \overrightarrow{r_2} + (r_3 - r_2) \cdot \overrightarrow{r_1} + (r_2 - r_1) \cdot \overrightarrow{r_3}, \\
\overrightarrow{N} &= r_3 \cdot (\overrightarrow{r_1} \wedge \overrightarrow{r_2}) + r_1 \cdot (\overrightarrow{r_2} \wedge \overrightarrow{r_3}) + r_2 \cdot (\overrightarrow{r_3} \wedge \overrightarrow{r_1}).
\end{aligned}$$

1. Montrer que $\overrightarrow{\imath} \wedge \overrightarrow{D} = -\frac{1}{e}\overrightarrow{S}$ et en déduire l'excentricité e de l'ellipse.

2. Montrer que $\overrightarrow{\imath}$ est colinéaire au vecteur $\overrightarrow{S} \wedge \overrightarrow{D}$.
3. Montrer que $\overrightarrow{\imath} \wedge \overrightarrow{N} = \frac{p}{e}\overrightarrow{S}$ et en déduire le paramètre p de l'ellipse.
4. Exprimer le demi-grand axe a de l'ellipse en fonction du paramètre p et de l'excentricité e.
5. *Application numérique :* dans le plan rapporté à un repère orthonormé direct, on considère les points suivants :

$$A_1\left(\begin{smallmatrix}0\\1\end{smallmatrix}\right), \quad A_2\left(\begin{smallmatrix}2\\2\end{smallmatrix}\right), \quad A_3\left(\begin{smallmatrix}3{,}5\\0\end{smallmatrix}\right), \quad O\left(\begin{smallmatrix}0\\0\end{smallmatrix}\right).$$

Déterminer numériquement les caractéristiques de l'unique ellipse dont O est un foyer et qui passe par les trois points A_1, A_2 et A_3.

2.4.3 Calcul matriciel

Pour définir une matrice, on utilise l'instruction `matrix` en précisant éventuellement l'anneau (ou le corps) de base :

```
A = matrix(QQ,[[1,2],[3,4]]); A
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Pour trouver une solution particulière à l'équation matricielle $Ax = b$ (resp. $xA = b$), on utilise la fonction la fonction `solve_right` (resp. `solve_left`). Pour trouver *toutes* les solutions d'une équation matricielle, il faut ajouter à une solution particulière la forme générale de l'équation homogène associée. Pour résoudre une équation homogène de la forme $Ax = 0$ (resp. $xA = 0$), on utilisera la fonction `right_kernel` (resp. `left_kernel`), comme dans l'exercice qui suit ce tableau récapitulatif :

Fonction	Syntaxe
Déclaration d'une matrice	`matrix`
Résolution d'une équation matricielle	`solve_right, solve_left`
Noyau à droite, à gauche	`right_kernel, left_kernel`
Réduction sous forme échelonnée en ligne	`echelon_form`
Sous-espace engendré par les colonnes	`column_space`
Sous-espace engendré par les lignes	`row_space`
Concaténation de matrices	`matrix_block`

Exercice 10. (*Bases de sous-espaces vectoriels*)

1. Déterminer une base de l'espace des solutions du système linéaire homogène associé à la matrice :

$$A = \begin{pmatrix} 2 & -3 & 2 & -12 & 33 \\ 6 & 1 & 26 & -16 & 69 \\ 10 & -29 & -18 & -53 & 32 \\ 2 & 0 & 8 & -18 & 84 \end{pmatrix}.$$

2. Déterminer une base de l'espace F engendré par les colonnes de A.
3. Caractériser F par une ou plusieurs équations.

Exercice 11. (*Une équation matricielle*) On rappelle le lemme de factorisation des applications linéaires. Soient E, F, G des $\mathbb{K}$-espaces vectoriels de dimension finie. Soient $u \in \mathcal{L}(E, F)$ et $v \in \mathcal{L}(E, G)$. Alors les assertions suivantes sont équivalentes :

i) il existe $w \in \mathcal{L}(F, G)$ tel que $v = w \circ u$,
ii) $\mathrm{Ker}(u) \subset \mathrm{Ker}(v)$.

On cherche toutes les solutions à ce problème dans un cas concret. Soient

$$A = \begin{pmatrix} -2 & 1 & 1 \\ 8 & 1 & -5 \\ 4 & 3 & -3 \end{pmatrix} \quad \text{et} \quad C = \begin{pmatrix} 1 & 2 & -1 \\ 2 & -1 & -1 \\ -5 & 0 & 3 \end{pmatrix}.$$

Déterminer toutes les solutions $B \in \mathcal{M}_3(\mathbb{R})$ de l'équation $A = BC$.

3

Programmation et structures de données

L'utilisation de Sage décrite dans les chapitres précédents effectue en une seule commande un calcul mathématique, mais le système autorise aussi la programmation d'une suite d'instructions.

Le système de calcul formel Sage est en fait une extension du langage informatique Python et permet, à quelques changements de syntaxe près, d'exploiter les méthodes de programmation de ce langage.

Les commandes décrites dans les chapitres précédents prouvent qu'il n'est pas nécessaire de connaître le langage Python pour utiliser Sage ; ce chapitre montre au contraire comment employer dans Sage les structures élémentaires de programmation de Python. Il se limite aux bases de la programmation et peut être survolé par les personnes connaissant Python ; les exemples sont choisis parmi les *plus classiques* rencontrés en mathématiques pour permettre au lecteur d'assimiler rapidement la syntaxe de Python par analogie avec les langages de programmation qu'il connaît éventuellement.

La première section de ce chapitre présente la méthode algorithmique de programmation structurée avec les instructions de boucles et de tests, et la seconde section expose les fonctions opérant sur les listes et les autres structures composées de données.

3.1 Algorithmique

La programmation structurée consiste à décrire un programme informatique comme une suite finie d'instructions effectuées les unes à la suite des autres. Ces instructions peuvent être élémentaires ou composées :

– une instruction élémentaire correspond par exemple à l'affectation d'une valeur à une variable (cf. § 1.4.1), ou à l'affichage d'un résultat ;
– une instruction composée, utilisée dans les boucles et dans les tests, est construite à partir de plusieurs instructions qui peuvent être elles-mêmes simples ou composées.

3.1.1 Les boucles

Les boucles d'énumération

Une boucle d'énumération effectue les mêmes calculs pour toutes les valeurs entières d'un indice $k \in \{a \cdots b\}$; l'exemple suivant affiche le début 7, 14, 21, 28 et 35 de la table de multiplication par 7 :

```
sage: for k in [1..5] :
....:     print 7*k  # bloc qui contient une seule instruction[1]
```

Les deux-points « `:` » à la fin de la première ligne introduisent le bloc d'instructions qui est évalué pour chaque valeur successive 1, 2, 3, 4 et 5 de la variable `k`. À chaque itération Sage affiche le produit $7 \times k$ par l'intermédiaire de la commande `print`.

Sur cet exemple le bloc des instructions répétées contient une seule instruction (`print`) qui est saisie de façon décalée par rapport au mot-clef `for`. Un bloc composé de plusieurs instructions est caractérisé par des instructions saisies les unes sous les autres avec la même indentation.

Cette boucle sert entre autre à calculer un terme donné d'une suite récurrente et est illustrée dans les exemples placés à la fin de cette section.

Boucles *tant que*

L'autre famille de boucles est constituée des boucles *tant que*. Comme les boucles d'énumération `for`, celles-ci évaluent un certain nombre de fois les mêmes instructions ; en revanche le nombre de répétitions n'est pas fixé au début de la boucle mais dépend de la réalisation d'une condition.

La boucle *tant que*, comme son nom l'indique, exécute des instructions tant qu'une condition est réalisée, l'exemple suivant calcule la somme des carrés des entiers naturels dont l'exponentielle est inférieure ou égale à 10^6 :

```
sage: S = 0 ; k = 0       #            La somme S commence à 0
sage: while e^k <= 10^6 : #                e^13 <= 10^6 < e^14
....:     S = S+k^2       #               ajout des carrés k^2
....:     k = k+1 #une ligne vide peut être nécessaire ensuite
....: S                   #    de résultat 819=0+1+4+9+..+13^2
```

[1] La saisie d'un tel bloc doit se terminer par une ligne vide supplémentaire dans l'interpréteur de commandes interactif d'un terminal `xterm`.

La dernière instruction renvoie la valeur de la variable `S` et affiche le résultat :

$$S = \sum_{\substack{k\in\mathbb{N}\\ e^k\leq 10^6}} k^2 = \sum_{k=0}^{13} k^2 = 819, \qquad e^{13} \approx 442413 \leq 10^6 < e^{14} \approx 1202604.$$

Le bloc d'instructions ci-dessous comporte deux instructions d'affectations, la première additionne le nouveau terme, et la seconde passe à l'indice suivant. Ces deux instructions sont bien placées les unes sous les autres et indentées de la même façon à l'intérieur de la structure `while`.[2]

L'exemple suivant est un autre exemple typique de la boucle *tant que*. Il consiste à rechercher, pour un nombre $x \geq 1$, l'unique valeur $n \in \mathbb{N}$ vérifiant $2^{n-1} \leq x < 2^n$ qui est le plus petit entier vérifiant $x < 2^n$. Le programme ci-dessous compare x à 2^n dont la valeur est successivement 1, 2, 4, 8, etc. ; il effectue ce calcul pour $x = 10^4$:

```
sage: x = 10^4 ; u = 1 ; n = 0
sage: while u <= x : n=n+1; u=2*u  # n+=1; u*=2 aussi possible
sage: n                                          # de résultat 14
```

Tant que la condition $2^n \leq x$ est vérifiée, ce programme calcule les nouvelles valeurs $n + 1$ et $2^{n+1} = 2 \cdot 2^n$ des deux variables `n` et `u` et les mémorise à la place de n et 2^n. Cette boucle se termine lorsque la condition n'est plus vérifiée, c'est-à-dire pour $x < 2^n$:

$$x = 10^4, \qquad \min\{n \in \mathbb{N} \mid x \leq 2^n\} = 14, \qquad 2^{13} = 8192, \qquad 2^{14} = 16384.$$

Le corps d'une boucle *tant que* n'est pas effectué lorsque l'évaluation du test est faux dès la première évaluation.

Les blocs de commandes simples peuvent aussi être saisis sur une ligne à la suite des deux-points « `:` » sans définir un nouveau bloc indenté à partir de la ligne suivante.

Exemples d'application aux suites et aux séries

La boucle `for` permet de calculer facilement un terme donné d'une suite récurrente, soit par exemple cette suite récurrente :

$$u_0 = 1, \qquad \forall n \in \mathbb{N} \quad u_{n+1} = \frac{1}{1 + u_n^2}.$$

Le programme ci-dessous détermine une approximation numérique du terme u_n pour $n = 10$; la variable `U` est modifiée à chaque itération de la boucle

[2] Lors de la saisie dans un terminal une ligne vide est nécessaire pour clore la définition du bloc d'instructions de la boucle, avant de demander la valeur de la variable `S`. Cette condition n'est pas nécessaire en utilisant Sage par l'intermédiaire d'un navigateur internet et sera sous-entendue dans la suite.

pour passer de la valeur u_{n-1} à u_n en suivant la formule de récurrence. La première itération calcule u_1 à partir de u_0 pour $n = 1$, la deuxième fait de même de u_1 à u_2 quand $n = 2$, et la dernière des n itérations modifie la variable `U` pour passer de u_{n-1} à u_n :

```
sage: U=1.0
sage: for n in [1..20] :
....:   U = 1/(1+U^2)
sage: U                              # de résultat 0.68236043...
```

Le même programme avec l'entier `U=1` à la place de l'approximation numérique `U=1.0` sur la première ligne fait des calculs exacts sur les fractions rationnelles ; le résultat exact u_{10} est une fraction avec plus d'une centaine de chiffres, et u_{20} en comporte plusieurs centaines de milliers. Les calculs exacts sont intéressants lorsque les erreurs d'arrondis se cumulent dans les approximations numériques. Sinon, *à la main* comme *à la machine* les opérations sur les approximations numériques d'une dizaine de décimales sont plus rapides que celles sur des entiers ou des rationnels faisant intervenir 500, 1000 chiffres, ou plus.

Les sommes ou les produits peuvent être mis sous forme de suites récurrentes et se calculent de la même manière :

$$S_n = \sum_{k=1}^{n} (2k)(2k+1) = 2 \cdot 3 + 4 \cdot 5 + \cdots + (2n)(2n+1),$$
$$S_0 = 0, \qquad S_n = S_{n-1} + (2n)(2n+1) \quad \text{pour } n \in \mathbb{N}^*.$$

La méthode de programmation de cette série est celle des suites récurrentes ; le programme effectue des sommes successives à partir de 0 en additionnant les termes pour $k = 0$, $k = 1$, jusqu'à $k = n$:

```
sage: S=0 ; n=10
sage: for k in [1..n] : S = S+(2*k)*(2*k+1)
sage: S                          # La valeur de la somme est 1650
```

Cet exemple illustre une méthode générale de programmation d'une somme, mais, dans ce cas simple, le calcul formel aboutit au résultat en toute généralité :

```
sage: var('n k') ; res = sum(2*k*(2*k+1),k,1,n)
sage: res , factor(res)    # résultat développé puis factorisé
  4/3*n^3 + 3*n^2 + 5/3*n , 1/3*(n+1)*(4*n+5)*n
```

Ces résultats peuvent aussi être obtenus avec *un papier et un crayon* à partir de sommes bien connues :

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}, \qquad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$
$$\sum_{k=1}^{n} 2k\,(2k+1) = 4\sum_{k=1}^{n} k^2 + 2\sum_{k=0}^{n} k = \frac{2\,n\,(n+1)(2n+1)}{3} + n(n+1)$$
$$= \frac{n(n+1)((4n+2)+3)}{3} = \frac{n(n+1)(4n+5)}{3}.$$

Exemples d'approximation de limites de suites

La boucle d'énumération permet de calculer un terme donné d'une suite ou d'une série, la boucle *tant que* est au contraire bien adaptée pour approcher numériquement une limite de suite.

Si une suite $(a_n)_{n\in\mathbb{N}}$ converge vers $\ell \in \mathbb{R}$, les termes a_n sont *voisins* de ℓ pour n *assez grand*. Il est donc possible d'approcher ℓ par un certain terme a_n, et le problème mathématique consiste alors à majorer plus ou moins facilement l'erreur $|\ell - a_n|$. Cette majoration est immédiate pour des suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ adjacentes :

$$\begin{cases} (u_n)_{n\in\mathbb{N}} \text{ est croissante,} \\ (v_n)_{n\in\mathbb{N}} \text{ est décroissante,} \\ \lim\limits_{n\to+\infty} v_n - u_n = 0\ ; \end{cases}$$
$$\text{donc} \quad \begin{cases} \text{les deux suites convergent vers la même limite } \ell, \\ \forall p \in \mathbb{N} \quad u_p \le \lim\limits_{n\to+\infty} u_n = \ell = \lim\limits_{n\to+\infty} v_n \le v_p, \\ \qquad\quad \left|\ell - \frac{u_p+v_p}{2}\right| \le \frac{v_p-u_p}{2}. \end{cases}$$

Une étude mathématique prouve que les deux suites ci-dessous sont adjacentes et convergent vers $\sqrt{ab}$ lorsque $0 < a < b$:

$$u_0 = a, \qquad v_0 = b > a, \qquad u_{n+1} = \frac{2\,u_n v_n}{u_n + v_n}, \qquad v_{n+1} = \frac{u_n + v_n}{2}.$$

La limite commune de ces deux suites porte le nom de moyenne arithmético-harmoniques du fait que la moyenne arithmétique de deux nombres a et b est la moyenne au sens usuel $(a+b)/2$ et la moyenne harmonique h est l'inverse de la moyenne des inverses : $1/h = (1/a + 1/b)/2 = (a+b)/2ab$. Le programme suivant effectue la vérification sur des valeurs numériques particulières :

```
sage: U=2.0 ; V = 50.0 ;
sage: while V-U >= 1.0e-6 :      # remplace le rationnel 10^-6
....:   temp = U
....:   U = 2*U*V/(U+V)
....:   V = (temp+V)/2
sage: U, V                    # U=9.999999... et V=10.000000...
```

Les valeurs de u_{n+1} et de v_{n+1} dépendent de u_n et v_n ; pour cette raison la boucle principale de ce programme fait intervenir une variable intermédiaire appelée ici `temp` de façon à ce que les nouvelles valeurs u_{n+1}, v_{n+1} de `U`, `V` dépendent des deux valeurs précédentes u_n, v_n. Les deux premiers blocs ci-dessous définissent les deux mêmes suites alors que le dernier construit deux autres suites $(u'_n)_n$ et $(v'_n)_n$; l'affectation parallèle évite d'utiliser une variable intermédiaire :

`temp = 2*U*V/(U+V)`	`U,V = 2*U*V/(U+V),(U+V)/2`	`U = 2*U*V/(U+V)`
`V = (U+V)/2`		`V = (U+V)/2`
`U = temp`	(affectation parallèle)	$u'_{n+1} = \frac{2u'_n v'_n}{u'_n+v'_n}$
		$v'_{n+1} = \frac{u'_{n+1}+v'_n}{2}$

La série $S_n = \sum_{k=0}^{n}(-1)^k a_k$ est alternée dès que la suite $(a_n)_{n\in\mathbb{N}}$ est décroissante et de limite nulle, ceci signifie que les deux suites extraites $(S_{2n})_{n\in\mathbb{N}}$ et $(S_{2n+1})_{n\in\mathbb{N}}$ sont adjacentes de même limite notée ℓ. La suite $(S_n)_{n\in\mathbb{N}}$ converge donc aussi vers cette limite et $S_{2p+1} \leq \ell = \lim_{n\to+\infty} S_n \leq S_{2p}$.

Le programme suivant illustre ce résultat pour la suite $a_k = 1/k^3$ à partir de $k = 1$, en mémorisant dans deux variables `U` et `V` les deux termes successifs S_{2n} et S_{2n+1} de la série qui encadrent la limite :

```
sage: U = 0.0  # la somme S0 est vide, de valeur nulle
sage: V =-1.0  # S1 = -1/1^3
sage: n=0      # U et V contiennent S(2n) et S(2n+1)
sage: while U-V >= 1.0e-6 :
....:   n = n+1             # n+=1 est équivalent
....:   U = V + 1/(2*n)^3   # passage de S(2n-1) à S(2n)
....:   V = U - 1/(2*n+1)^3 # passage de S(2n) à S(2n+1)
sage: V,U
```

La boucle principale du programme modifie la valeur de n pour passer à l'indice suivant tant que les deux valeurs S_{2n} et S_{2n+1} ne sont pas assez proches l'une de l'autre. Il est donc nécessaire d'employer deux variables `U` et `V` pour mémoriser ces deux termes successifs ; le corps de la boucle détermine à partir de S_{2n-1} successivement S_{2n} et S_{2n+1}, d'où les affectations croisées à `U` et à `V`.

Le programme se termine lorsque deux termes consécutifs S_{2n+1} et S_{2n} qui encadrent la limite sont suffisamment proches l'un de l'autre, l'erreur d'approximation est $0 \leq a_{2n+1} = S_{2n} - S_{2n+1} \leq 10^{-6}$.

Les programmes sont similaires pour les séries alternées associées à ces suites :

$$\frac{1}{\log k} \text{ à partir du rang 2,} \qquad \frac{1}{k}, \qquad \frac{1}{k^2}, \qquad \frac{1}{k^4}, \qquad e^{-k\ln k} = 1/k^k.$$

Ces suites tendent plus ou moins vite vers 0 et l'approximation de la limite de ces séries demande selon les cas plus ou moins de calculs.

La recherche d'une précision de 3, 10, 20 ou 100 décimales sur les limites de ces séries consiste à résoudre les inéquations suivantes :

$$1/\log n \le 10^{-3} \iff n \ge e^{(10^3)} \approx 1.97\,10^{434}$$

$1/n \le 10^{-3}$	$\iff n \ge 10^3$	$1/n \le 10^{-10}$	$\iff n \ge 10^{10}$
$1/n^2 \le 10^{-3}$	$\iff n \ge \sqrt{10^3} \approx 32$	$1/n^2 \le 10^{-10}$	$\iff n \ge 10^5$
$1/n^4 \le 10^{-3}$	$\iff n \ge (10^3)^{1/4} \approx 6$	$1/n^4 \le 10^{-10}$	$\iff n \ge 317$
$e^{-n\log n} \le 10^{-3}$	$\iff n \ge 5$	$e^{-n\log n} \le 10^{-10}$	$\iff n \ge 10$
$1/n^2 \le 10^{-20}$	$\iff n \ge 10^{10}$	$1/n^2 \le 10^{-100}$	$\iff n \ge 10^{50}$
$1/n^4 \le 10^{-20}$	$\iff n \ge 10^5$	$1/n^4 \le 10^{-100}$	$\iff n \ge 10^{25}$
$e^{-n\log n} \le 10^{-20}$	$\iff n \ge 17$	$e^{-n\log n} \le 10^{-100}$	$\iff n \ge 57$

Dans les cas les plus simples la résolution de ces inéquations détermine donc un indice n à partir duquel la valeur S_n permet approcher la limite ℓ de la série, ainsi une boucle d'énumération `for` est aussi possible. Au contraire une boucle `while` est nécessaire dès que la résolution algébrique en n de l'inéquation $a_n \le 10^{-p}$ s'avère impossible.

Un programme Sage effectue de 10^5 à 10^7 opérations élémentaires par seconde. Un programme optimisé pour un processeur standard peut effectuer de l'ordre de 10^9 opérations par seconde.

Certaines approximations des limites précédentes demandent trop de calculs pour être obtenues directement, notamment dès que l'indice n dépasse un ordre de grandeur de 10^{10} ou 10^{12}. Une étude mathématique plus approfondie peut parfois permettre de déterminer la limite ou de l'approcher par d'autres méthodes, ainsi en est-il des séries de Riemann :

$$\lim_{n\to+\infty}\sum_{k=1}^{n}\frac{(-1)^k}{k^3} = -\frac{3}{4}\,\zeta(3), \qquad \text{avec } \zeta(p) = \lim_{n\to+\infty}\sum_{k=1}^{n}\frac{1}{k^p},$$
$$\lim_{n\to+\infty}\sum_{k=1}^{n}\frac{(-1)^k}{k} = -\log 2, \qquad \lim_{n\to+\infty}\sum_{k=1}^{n}\frac{(-1)^k}{k^2} = -\frac{\pi^2}{12},$$
$$\lim_{n\to+\infty}\sum_{k=1}^{n}\frac{(-1)^k}{k^4} = -\frac{7\pi^4}{6!}.$$

Par ailleurs Sage peut calculer formellement certaines de ces séries et détermine une approximation de $\zeta(3)$ avec 1200 décimales en quelques secondes en effectuant bien moins d'opérations que les 10^{400} nécessaires par l'application directe de la définition :

```
sage: var('k') ;                    sum((-1)^k/k, k, 1, +oo)
sage: sum((-1)^k/k^2, k, 1, +oo) ; sum((-1)^k/k^3, k, 1, +oo)
sage: -3/4 * zeta (N(3,digits=1200))
```

3.1.2 Les tests

L'autre instruction composée concerne les tests : les instructions effectuées dépendent de la valeur booléenne d'une condition. La structure et deux

syntaxes possibles de cette instruction sont les suivantes :

`if` une condition `:`
  une suite d'instructions
	`if` une condition `:`
	  une suite d'instructions
	`else :`
	  sinon d'autres instructions

La suite de Syracuse est définie selon une condition de parité :

$$u_0 \in \mathbb{N}^* \qquad u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{si } u_n \text{ est impair.} \end{cases}$$

La « conjecture tchèque » énonce — sans preuve connue en 2010 — que pour toutes les valeurs initiales $u_0 \in \mathbb{N}^*$ il existe un rang n pour lequel $u_n = 1$. Les termes suivants sont alors 4, 2, 1, 4, 2, etc. Calculer chaque terme de cette suite se fait par l'intermédiaire d'un test. Ce test est placé à l'intérieur d'une boucle *tant que* qui détermine la plus petite valeur de $n \in \mathbb{N}$ vérifiant $u_n = 1$:

```
sage: u = 6 ; n = 0
sage: while u != 1 :   # test "différent de" <> aussi possible
....:   if u % 2 == 0 :#  l'opérateur % est le reste euclidien
....:     u = u/2
....:   else :
....:     u = 3*u+1
....:   n = n+1
sage: n                                      # Le résultat est n=8
```

Tester si u_n est pair se fait en comparant à 0 le reste de la division par 2 de u_n. Le nombre n d'itérations effectuées s'obtient en ajoutant 1 à la variable `n` lors de chaque itération, à partir de $n = 0$. Cette boucle se termine dès que la valeur calculée de u_n est 1 ; par exemple si $u_0 = 6$ alors $u_8 = 1$ et $8 = \min\{p \in \mathbb{N}^* | u_p = 1\}$:

$p =$	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
$u_p =$	6	3	10	5	16	8	4	2	1	4	2	$\cdots$

La vérification pas-à-pas du bon fonctionnement de ces lignes peut se faire par un *print-espion* de la forme `print u,n` placé dans le corps de la boucle.

L'instruction `if` permet en outre des tests imbriqués dans la branche `else` à l'aide du mot-clef `elif`. Ces deux structures sont équivalentes :

```
if une condition cond1 :
    une suite d'instructions inst1
else :
    if une condition cond2 :
        une suite d'instructions inst2
    else :
        if une condition cond3 :
            une suite d'instructions inst3
        else :
            dans les autres cas instn
```

```
if cond1 :
    inst1
elif cond2 :
    inst2
elif cond3 :
    inst3
else :
    instn
```

Comme pour les boucles, les instructions simples associées aux tests peuvent être placées à la suite des deux-points et non dans un bloc en dessous.

3.1.3 Les procédures et les fonctions

Syntaxe générale

Comme bien d'autres langages informatiques, Sage permet à l'utilisateur de définir des fonctions ou des instructions *sur mesure*. La commande `def` dont la syntaxe est détaillée ci-dessous autorise la définition de *procédures* — qui ne renvoient pas de résultat — ou de *fonctions* — qui renvoient un résultat — avec un ou plusieurs arguments. Ce premier exemple définit la fonction $(x, y) \mapsto x^2 + y^2$:

```
sage: def fct2 (x,y) :
....:   return x^2 + y^2
sage: var('a') ; fct2 (a, 2*a)              # de résultat 5 a^2
```

L'évaluation de la fonction se termine par la commande `return` dont l'argument, ici $x^2 + y^2$, est le résultat de la fonction.

Une procédure est définie de la même façon sans renvoyer de résultat, et en l'absence de l'instruction `return` le bloc d'instructions définissant le sous-programme est évalué jusqu'au bout.

Par défaut Sage considère que toutes les variables intervenant dans une fonction sont locales. Ces variables sont créées à chaque appel de la fonction, détruites à la fin, et sont indépendantes d'autres variables de même nom pouvant déjà exister, comme dans les autres langages de programmation ; les variables globales ne sont pas modifiées par l'évaluation d'une fonction ayant des variables locales du même nom, et réciproquement Sage empêche par défaut la modification des variables globales à l'intérieur d'une fonction :

```
sage: def essai (u) :
....:   t = u^2
....:   return t*(t+1)
sage: t=1 ; u=2 ; essai(3) ; t ; u        # affiche 90, 1 et 2
```

L'exemple suivant reprend le calcul de la moyenne arithmético-harmonique de deux nombres supposés strictement positifs :

```
sage: def MoyAH (u, v) :
....:    u,v = min(u,v),max(u,v) ;
....:    while v-u > 2.0e-8 :
....:       u,v = 2*u*v/(u+v),(u+v)/2
....:    return (u+v) / 2
sage: MoyAH (1.,2.)                # est proche de 1.41421...
sage: MoyAH                        # correspond à une fonction
  <function MoyAH at 0xc584df4>
```

La fonction `MoyAH` comporte deux paramètres notés `u` et `v` qui sont des variables locales dont les valeurs initiales sont fixées lors de l'appel de cette fonction ; par exemple `moyAH(1.,2.)` débute l'exécution de cette fonction avec ces valeurs 1. et 2. des variables `u` et `v`.

La programmation structurée conseille de définir une fonction de façon à ce que la commande `return` soit la dernière instruction du bloc de la procédure. Placée au milieu du bloc d'instructions d'une fonction, cette commande `return` termine l'exécution de la fonction en interrompant avant la fin l'évaluation complète de ce bloc ; en outre il peut y en avoir plusieurs dans différentes branches des tests.

La traduction informatique de l'état d'esprit des mathématiques suggère de programmer des fonctions renvoyant chacune un résultat à partir de leurs arguments, plutôt que des procédures affichant ces résultats par une commande `print`. Le système de calcul formel Sage repose d'ailleurs sur de très nombreuses fonctions, par exemple `exp` ou `solve` qui renvoient toute un résultat, par exemple un nombre, une expression, une liste de solutions, etc.

Méthode itérative et méthode récursive

Une fonction définie par l'utilisateur est construite comme une suite d'instructions. Une fonction est dite récursive lorsque son évaluation nécessite dans certains cas d'exécuter cette même fonction avec d'autres paramètres. La suite factorielle $(n!)_{n\in\mathbb{N}}$ en est un exemple simple :

$$0! = 1, \qquad (n+1)! = (n+1)\,n! \quad \text{pour tout } n \in \mathbb{N}.$$

Les deux fonctions suivantes aboutissent au même résultat à partir d'un argument entier naturel n ; la première fonction utilise la méthode itérative avec une boucle `for`, et la seconde la méthode récursive traduisant *mot pour mot* la définition récurrente précédente :

```
sage: def fact1 (n) :
....:     res = 1
....:     for k in [1..n] : res = res*k
....:     return res

sage: def fact2 (n) :
....:     if n==0 : return 1
....:     else : return n*fact(n-1)
```

La suite de Fibonacci est une suite récurrente d'ordre 2 car la valeur de u_{n+2} dépend uniquement de celles de u_n et de u_{n+1} :

$$u_0 = 1, \qquad u_1 = 1, \qquad u_{n+2} = u_{n+1} + u_n \quad \text{pour tout } n \in \mathbb{N}.$$

La fonction `fib1` ci-dessous applique une méthode de calcul itératif des termes de la suite de Fibonacci en utilisant deux variables intermédiaires `U` et `V` pour mémoriser les deux valeurs précédentes de la suite avant de passer au terme suivant :

```
sage: def fib1 (n) :
....:   if n==0 or n==1 : return 1
....:   else
....:     U=1 ; V=1 # les deux termes précédents, ici u0 et u1
....:     for k in [2..n] : W=U+V ; U=V ; V=W # ou U,V = V,U+V
....:     return V
sage: fib1 (8)                                        # renvoie 34
```

La boucle applique à partir de $n = 2$ la relation $u_n = u_{n-1} + u_{n-2}$. Cette méthode est efficace mais sa programmation doit transformer la définition de la suite de façon à l'adapter aux manipulations des variables.

Au contraire la fonction récursive `fib2` suit de beaucoup plus près la définition mathématique de cette suite, ce qui facilite sa programmation et sa compréhension :

```
sage: def fib2 (n) :
....:   if 0 <= n <= 1 : return 1          # pour n==0 ou n==1
....:   else : return fib2(n-1) + fib2(n-2)
```

Le résultat de cette fonction est la valeur renvoyée par l'instruction conditionnelle : 1 pour $n = 0$ et $n = 1$, et la somme `fib2(n-1)+fib2(n-2)` sinon ; chaque branche du test comporte une instruction `return`.

Cette méthode est cependant moins efficace car beaucoup de calculs sont inutilement répétés. Par exemple `fib2(5)` évalue `fib2(3)` et `fib2(4)` qui sont eux aussi calculés de la même manière. Ainsi Sage évalue deux fois `fib2(3)` et trois fois `fib2(2)`. Ce processus se termine lors de l'évaluation de `fib2(0)` ou de `fib2(1)` de valeur 1, et l'évaluation de `fib2(n)` consiste à

calculer finalement u_n par $1+1+\cdots+1$. Le nombre d'additions effectuées pour déterminer u_n est sa valeur, car $u_0 = u_1 = 1$; ce nombre est considérable et croît très rapidement. Aucun ordinateur, aussi rapide soit-il, ne peut calculer ainsi u_{50}. D'autres méthodes sont aussi possibles, par exemple mémoriser les calculs intermédiaires par la commande `cached_function` ou exploiter une propriété des puissances de matrices pour calculer le millionième terme de cette suite présentée dans le paragraphe suivant d'exponentiation rapide.

3.1.4 Algorithme d'exponentiation rapide

Une méthode naïve de calcul de a^n consiste à effectuer $n \in \mathbb{N}$ multiplications par a dans une boucle `for` :

```
sage: a=2; n=6; res=1      # 1 est l'élément neutre du produit.
sage: for k in [1..n] : res = res*a
sage: res                          # La valeur de res est 2^6=64.
```

Les puissances entières interviennent sous de très nombreux aspects en mathématique et en informatique ; ce paragraphe étudie une méthode générale de calcul d'une puissance entière a^n plus rapide que celle-ci. La suite $(u_n)_{n\in\mathbb{N}}$ définie ci-dessous vérifie $u_n = a^n$; ce résultat se démontre par récurrence à partir des égalités $a^{2k} = (a^k)^2$ et $a^{k+1} = a\,a^k$:

$$u_n = \begin{cases} 1 & \text{si } n = 0, \\ u_{n/2}^2 & \text{si } n \text{ est pair et strictement positif,} \\ a\,u_{n-1} & \text{si } n \text{ est impair.} \end{cases}$$

Par exemple pour u_{11} : $u_{11} = a\,u_{10}$, $u_{10} = u_5^2$, $u_5 = a\,u_4$, $u_4 = u_2^2$, $u_2 = u_1^2$, $u_1 = a\,u_0 = a$;
ainsi : $u_2 = a^2$, $u_4 = u_2^2 = a^4$, $u_5 = a\,a^4 = a^5$, $u_{10} = u_5^2 = a^{10}$, $u_{11} = a\,a^{10} = a^{11}$.

Cette formule correspond bien à celle d'une suite car le calcul de u_n ne fait intervenir que des termes u_k avec $k \in \{0 \cdots n-1\}$, et donc celui-ci est bien décrit en un nombre fini d'opérations.

En outre cet exemple montre que la valeur de u_{11} est obtenue par l'évaluation des 6 termes u_{10}, u_5, u_4, u_2, u_1 et u_0, et effectue uniquement 6 multiplications. Le calcul de u_n nécessite entre $\log n \,/\, \log 2$ et $2\log n \,/ \log 2$ multiplications car après une ou deux étapes — selon que n est pair ou impair — u_n s'exprime en fonction de u_k, avec $k \leq n/2$. Cette méthode est donc incomparablement plus rapide que la méthode naïve quand n est grand ; une vingtaine de termes pour $n = 10^4$ et non 10^4 produits :

indices traités :	10 000	5 000	2 500	1 250	625	624	312	156	78
	39	38	19	18	9	8	4	2	1

Cette méthode n'est cependant pas toujours la plus rapide ; les calculs suivants sur b, c, d et f effectuent 5 produits pour calculer a^{15}, alors que les opérations sur u, v, w, x et y nécessitent 6 multiplications sans compter le produit par 1 :

$b = a^2 \quad c = ab = a^3 \quad d = c^2 = a^6 \quad f = cd = a^9 \quad df = a^{15}$: 5 produits ;
$u = a^2 \quad v = au = a^3 \quad w = v^2 = a^6$
$x = aw = a^7 \quad y = x^2 = a^{14} \quad ay = a^{15}$: 6 produits.

La fonction récursive `puiss1` calcule cette suite récurrente avec uniquement des multiplications pour programmer l'opérateur de puissance par un entier naturel :

```
sage: def puiss1 (a, n) :
....:   if n == 0 : return 1
....:   elif n % 2 == 0 : b = puiss1 (a,n/2) ; return b*b
....:   else : return a * puiss1(a,n-1)
....: puiss1 (2, 11)              # a pour résultat 2^11=2048
```

Le nombre d'opérations effectuées par cette fonction est le même que celui fait par un calcul à la main en reprenant les résultats déjà calculés. Au contraire si les instructions `b=puiss(a,n/2);return b*b` faites après le test de parité de n étaient remplacées par `puiss1(a,n/2)*puiss1(a,n/2)` Sage effectuerait beaucoup plus de calculs car — comme pour la fonction récursive `fib2` calculant la suite de Fibonacci — certaines opérations seraient inutilement répétées. Il y aurait en définitive n multiplications, autant que par la méthode naïve.

Par ailleurs la commande `return puiss1(a*a,n/2)` peut remplacer de façon équivalente ces deux instructions.

Le programme ci-dessous applique un algorithme similaire pour effectuer le même calcul par une méthode itérative :

```
sage: def puiss2 (u, k) :
....:   v = 1
....:   while k != 0 :
....:     if k % 2 == 0 : u = u*u ; k = k/2
....:     else : v = v*u ; k = k-1
....:   return v
sage: puiss2 (2, 10)              # a pour résultat 2^10=1024
```

Le fait que la valeur de `puiss2(a,n)` est a^n se démontre en vérifiant qu'itération après itération les valeurs des variables `u`, `v` et `k` sont liées par l'égalité $v\,u^k = a^n$, que l'entier k soit pair ou impair. À la première itération $v = 1$, $u = a$ et $k = n$; après la dernière itération $k = 0$, donc $v = a^n$.

Les valeurs successives de la variable `k` sont entières et positives, et elles forment une suite strictement décroissante. Cette variable ne peut donc

prendre qu'un nombre fini de valeurs avant d'être nulle et de terminer ainsi l'instruction de boucle.

En dépit des apparences — la première fonction est programmée récursivement, et la seconde de façon itérative — ces deux fonctions traduisent presque le même algorithme : la seule différence est que la première évalue a^{2k} par $(a^k)^2$ alors que la seconde calcule a^{2k} par $(a^2)^k$ lors de la modification de la variable `u`.

Cette méthode ne se limite pas au calcul des puissances positives de nombres à partir de la multiplication, mais s'adapte à toute loi de composition interne associative. Cette loi doit être associative afin de vérifier les propriétés usuelles des produits itérés. Ainsi en remplaçant le nombre 1 par la matrice unité $\mathbf{1}_n$ les deux fonctions précédentes évalueraient les puissances positives de matrices carrées. Ces fonctions illustrent comment programmer efficacement l'opérateur puissance « ^ » à partir de la multiplication, et ressemblent à la méthode implantée dans Sage.

Par exemple une puissance de matrice permet d'obtenir des termes d'indices encore plus grands que précédemment de la suite de Fibonacci :

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, \qquad X_n = \begin{pmatrix} u_n \\ u_{n+1} \end{pmatrix}, \qquad AX_n = X_{n+1} \qquad A^n X_0 = X_n.$$

Le programme Sage correspondant tient en deux lignes, et le résultat recherché est la première coordonnée du produit matriciel $A^n X$ qui fonctionne effectivement même pour $n = 10^7$; ces deux programmes sont équivalents et leur efficacité provient du fait que Sage applique essentiellement cette méthode d'exponentiation rapide :

```
sage: def fib3 (n) :
....:   A = matrix ([[0,1],[1,1]]) ; X0 = vector ([1,1])
....:   return (A^n*X0)[0]

sage: def fib4 (n) :
....:   return (matrix([[0,1],[1,1]])^n * vector([1,1]))[0]
```

Cette dernière fonction est de type purement fonctionnel car elle n'utilise aucune variable intermédiaire, mais seulement des constantes et les paramètres des fonctions.

3.1.5 Affichage et saisie

L'instruction `print` est la principale commande d'affichage. Par défaut, les arguments sont affichés les uns à la suite des autres séparés par un espace ; Sage passe automatiquement à la ligne à la fin de la commande :

```
sage: print 2^2, 3^3, 4^4 ; print 5^5, 6^6
  4 27 256
  3125 46656
```

Une virgule à la fin de la commande `print` omet ce passage à la ligne, et la prochaine instruction `print` continue sur la même ligne :

```
sage: for k in [1..10] : print '+', k,
  + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

Il est possible d'afficher des résultats sans espace intermédiaire en transformant ceux-ci en une chaîne de caractères par la fonction `str(..)`, et en effectuant la concaténation des chaînes de caractères par l'opérateur « `+` » :

```
sage: print 10,0.5 ; print 10+0.5 ; print 10.0, 5
sage: print 10+0,5 ; print str(10)+str(0.5)
  10 0.500000000
  10.500000000
  10.000000000 5
  10 5
  100.500000000
```

La seconde section de ce chapitre sur les listes et les structures présente les chaînes de caractères de manière plus approfondie.

La commande `print` permet aussi de formater l'affichage des nombres pour les présenter en tableaux, l'exemple suivant à partir du motif `%.d` et de l'opérateur `%` affiche la table des carrés les uns sous les autres :

```
sage: for k in [1..15] : print '%2d^2=%3d' % (k, k^2)
  1^2=  1
  2^2=  4
.........
 15^2=225
```

L'opérateur `%` insère les nombres placés à droite dans la chaîne de caractères à gauche à la place des signes de conversion comme `%3d` ou `%1.4f`. Dans l'exemple précédent le terme `%3d` complète à gauche par des espaces la chaîne de caractères représentant l'entier k^2 de façon à occuper au minimum trois caractères. De même le motif `%1.4f` dans `'pi=%1.4f' % n(pi)` affiche `'pi=3.1416'` avec un chiffre avant le séparateur décimal et quatre après.

Dans un terminal, la fonction `raw_input('message')` affiche le texte `message`, attend une saisie au clavier validée par un passage à la ligne, et renvoie la chaîne de caractères correspondante.

3.2 Listes et structures composées

Cette seconde section du chapitre détaille les structures de données composées de Sage : les listes — de type modifiable ou immuable —, les chaînes de caractères, les ensembles et les dictionnaires.

Les données élémentaires de Sage sont les nombres entiers `Integer`, les nombres rationnels `Rational`, les approximations numériques de précision variable, et les nombres complexes construits partir de `I` ; ils ont été présentés dans le chapitre 1.

3.2.1 Définition des listes et accès aux éléments

La notion de liste en informatique et celle de n-uplet en mathématiques permettent l'énumération d'objets mathématiques. Cette notion de couple — avec $(a, b) \neq (b, a)$ — et de n-uplet précise la position de chaque élément au contraire de la notion d'ensemble.

Sage définit les listes en plaçant les éléments entre crochets `[...]` séparés par des virgules. L'affectation du triplet $(10, 20, 30)$ à la variable `L` s'effectue de la manière suivante, et la liste vide, sans élément, est simplement définie ainsi :

```
sage: L = [10, 20, 30] ; L    # de résultat [10, 20, 30]
sage: []                      # est la liste vide sans élément
```

Les coordonnées des listes sont énumérées dans l'ordre à partir de l'indice 0, puis 1, 2, etc. L'accès à la coordonnée de rang k d'une liste L s'effectue par `L[k]`, mathématiquement ceci correspond à la projection canonique de la liste considérée comme un élément d'un produit cartésien sur la coordonnée d'ordre k. La fonction `len` renvoie le nombre d'éléments d'une liste :

```
sage: L[1]; len(L); len([])  # de résultats 20, 3 et 0 pour []
```

La modification d'une coordonnée est obtenue de la même manière, par affectation de la coordonnée correspondante, ainsi la commande suivante modifie le troisième terme de la liste indicé par 2 :

```
sage: L[2] = 33 ; L            # la liste L devient [10,20,33]
```

Les indices strictement négatifs accèdent aux éléments de la liste comptés à partir du dernier terme :

```
sage: L=[11,22,33]; L[-1]; L[-2]; L[-3] # énumère 33, 22 et 11
```

La commande `L[p:q]` extrait la sous-liste `[L[p],L[p+1],....,L[q-1]]` qui est vide si $q \leq p$. Des indices négatifs permettent d'extraire les derniers termes de la liste ; enfin la référence `L[p:]` construit la sous-liste `L[p:len(L)]` à partir de l'indice p jusqu'à la fin, et `L[:q]=L[0:q]` énumère les éléments à partir du début jusqu'à l'indice q exclu :

```
sage: L=[0,11,22,33,44,55]
sage: L[2:4]; L[-4:4]; L[2:-2]  # ont le même résultat [22,33]
sage: L[:4]                     # [0,11,22,33]
sage: L[4:]                     # [44,55]
```

De la même manière que la commande `L[n]=...` modifie un élément de la liste, l'affectation `L[p:q]=[....]` remplace la sous-liste entre les indices p compris et q exclu :

```
sage: L = [0,11,22,33,44,55,66,77]
sage: L[2:6] = [12,13,14]              # remplace [22,33,44,55]
```

Ainsi `L[:1]=[]` et `L[-1:]=[]` suppriment respectivement le premier et le dernier terme d'une liste, et réciproquement `L[:0]=[a]` et `L[len(L):]=[a]` insèrent un élément a respectivement en tête et à la fin de la liste. Plus généralement les termes d'une liste vérifient ces égalités :

$$\texttt{L} = (\ell_0, \ell_1, \ell_2, \cdots, \ell_{n-1}) = (\ell_{-n}, \ell_{1-n}, \cdots \ell_{-2}, \ell_{-1}) \qquad \text{avec n=}\texttt{len(L)},$$
$$\ell_k = \ell_{k-n} \quad \text{pour } 0 \le k < n, \qquad \ell_j = \ell_{n+j} \quad \text{pour } -n \le j < 0.$$

L'opérateur `in` teste l'appartenance d'un élément à une liste. Sage effectue le test d'égalité de deux listes par « `==` » qui compare les éléments un par un. Ces deux sous-listes avec des indices positifs ou négatifs sont égales :

```
sage: L = [1,3,5,7,9,11,13,15,17,19]
sage: L[3:len(L)-5] == L[3-len(L):-5]  #       de réponse True
sage: [5 in L, 6 in L]                 # renvoie [True, False]
```

Les exemples ci-dessus concernent des listes d'entiers, mais les éléments des listes peuvent être n'importe quels objets Sage, nombres, expressions, autres listes, etc.

3.2.2 Opérations globales sur les listes

L'opérateur d'addition « `+` » effectue la concaténation de deux listes, et l'opérateur de multiplication « `*` » associé à un entier itère cette concaténation :

```
sage: L = [10,20,30] ; L + [1,2,3]  # renvoie [10,20,30,1,2,3]
sage: 4 * [1,2,3]                   # [1,2,3, 1,2,3, 1,2,3, 1,2,3]
```

La concaténation de ces deux sous-listes d'indice limite k reconstruit la liste initiale :

$$\begin{aligned}\texttt{L} = \texttt{L[:k]+L[k:]} &= (\ell_0, \ell_1, \ell_2, \cdots, \ell_{n-1}) \\ &= (\ell_0, \ell_1, \ell_2, \cdots, \ell_{k-1}) + (\ell_k, \ell_{k+1}, \ell_{k+2}, \cdots, \ell_{n-1}).\end{aligned}$$

L'exemple suivant illustre cette propriété :

```
sage: L = 5*[10,20,30] ; L[:3]+L[3:] == L    # de réponse True
```

L'opérateur composé de deux points « `..` » automatise la construction des listes d'entiers sans énumérer explicitement tous leurs éléments. L'exemple suivant construit une liste faite d'énumérations d'entiers et d'éléments isolés :

```
sage: [1..3, 7, 10..13]        # renvoie [1,2,3,7,10,11,12,13]
```

La suite de ce paragraphe décrit comment construire par compréhension l'image d'une liste par une application et une sous-liste d'une liste. Les fonctions associées sont `map` et `filter`, et la construction `[..for..x..in..]`. Les mathématiques font souvent intervenir des listes constituées des images par une application f de ses éléments :

$$(a_0, a_1, a_2, \cdots, a_{n-1}) \mapsto (f(a_0), f(a_1), \cdots, f(a_{n-1})).$$

La commande `map` construit cette image ; l'exemple suivant applique la fonction trigonométrique cos à une liste d'angles usuels :

```
sage: map (cos, [0, pi/6, pi/4, pi/3, pi/2])
  [1, 1/2*sqrt(3), 1/2*sqrt(2), 1/2, 0]
```

Il est aussi possible d'utiliser une fonction de l'utilisateur définie par `def`, ou de déclarer directement une fonction par `lambda` ; la commande ci-dessous est équivalente à la précédente et applique la fonction définie par $t \mapsto \cos t$:

```
sage: map (lambda t : cos(t), [0, pi/6, pi/4, pi/3, pi/2])
```

La commande `lambda` est suivie de la déclaration du ou des paramètres séparés par des virgules, et ne peut comporter après le deux-point qu'une et une seule expression qui est le résultat de la fonction sans utiliser l'instruction `return`.

Cette fonction `lambda` peut aussi comporter un test, les codes suivants sont équivalents :

```
fctTest1 = lambda x : res1 if cond else res2
def fctTest2 (x) :
  if cond : return res1
  else : return res2
```

Les trois commandes `map` suivantes sont équivalentes, la composition des applications $N \circ \cos$ étant effectuée d'une façon ou d'une autre :

```
sage: map (lambda t : N(cos(t)), [0, pi/6, pi/4, pi/3, pi/2])
sage: def ncos (t) : return (N(cos(t)))
sage: map (ncos, [0, pi/6, pi/4, pi/3, pi/2])
sage: map (N, map (cos, [0, pi/6, pi/4, pi/3, pi/2]))
```

La commande `filter` construit une sous-liste des éléments vérifiant une condition, l'exemple suivant applique le test de primalité `is_prime` aux entiers jusqu'à 70 :

```
sage: filter (is_prime, [1..70])
  [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67]
```

La fonction de test peut aussi être définie à l'intérieur de la commande `filter`. L'exemple ci-dessous, par des essais exhaustifs, détermine toutes les racines quatrièmes de 7 modulo le nombre premier 37 ; cette équation comporte quatre solutions 3, 18, 19 et 34 :

```
sage: p=37 ; filter (lambda(n) : n^4 % p == 7, [0..p-1])
```

Par ailleurs, la commande `[..for..x..in..]` construit par compréhension une liste ; ces deux commandes énumérent de façon équivalente les entiers impairs de 1 à 99 :

```
sage: map (lambda n:2*n+1, [0..49])
sage: [2*n+1 for n in [0..49]]
```

Cette commande est indépendante de la commande de boucle `for`. La condition `if` associée à `for` aboutit à cette construction équivalente à la fonction `filter` :

```
sage: filter (is_prime, [1..70])
sage: [p for p in [1..70] if is_prime(p)]
```

Les deux exemples suivants combinent les tests `if` et `filter` avec `for` pour déterminer une liste des nombres premiers qui sont congrus à 1 modulo 4, puis une liste de carrés de nombres premiers :

```
sage: filter (is_prime, [4*n+1 for n in [0..20]])
sage: [n^2 for n in [1..20] if is_prime(n)]
  [5, 13, 17, 29, 37, 41, 53, 61, 73]
  [4, 9, 25, 49, 121, 169, 289, 361]
```

Dans le premier cas le test `is_prime` est effectué après le calcul $4n+1$ alors que dans le second le test est effectué avant le calcul du carré n^2.

La fonction `reduce` opère par associativité de la gauche vers la droite sur les éléments d'une liste ; définissons ainsi la loi de composition interne $\star$:

$$x \star y = 10x + y, \qquad \text{alors } ((1 \star 2) \star 3) \star 4 = (12 \star 3) \star 4 = 1234.$$

Le premier argument est une fonction à deux paramètres, le deuxième est la liste des arguments, et l'éventuel troisième argument permet de valider le résultat sur une liste vide :

```
sage: reduce (lamba x,y : 10*x+y, [1,2,3,4])   # réponse: 1234
sage: reduce (lamba x,y : 10*x+y, [9,8,7,6],1) #         98761
```

L'exemple ci-dessous calcule un produit d'entiers impairs, le troisième paramètre correspond généralement à l'élément neutre de l'opération appliquée pour obtenir un résultat valide sur une liste vide :

```
sage: L = [2*n+1 for n in [0..9]]
sage: reduce (lambda x,y : x*y, L, 1)
```

Les fonctions `sum` et `prod` de Sage appliquent directement l'opérateur `reduce` pour calculer des sommes et des produits ; le résultat est le même dans les trois exemples ci-dessous, et la commande sur une liste autorise en outre d'ajouter un second terme optionnel représentant l'élément neutre, l'élément unité du produit et élément nul pour la somme, par exemple une matrice unité pour un produit matriciel :

```
sage: prod ([2*n+1 for n in [0..9]], 1) # une liste avec for
sage: prod ( 2*n+1 for n in [0..9])     # sans liste
sage: prod (n for n in [0..9] if n%2 == 1)
```

La fonction `any` associée à l'opérateur `or` et la fonction `all` à l'opérateur `and` sont de principe et de syntaxe équivalente. Cependant l'évaluation se termine dès que le résultat `True` ou `False` d'un des termes impose ce résultat sans effectuer l'évaluation des termes suivants :

```
sage: def fct (x) : return 4/x == 2
sage: all (fct(x) for x in [2,1,0])# réponse False sans erreur
sage: any (fct(x) for x in [2,1,0]) # réponse True sans erreur
```

En revanche la construction de la liste `[fct(x) for x in [2,1,0]]` et la commande `all([fct(x) for x in [2,1,0]])` provoquent des erreurs car tous les termes sont évalués, y compris le dernier avec $x = 0$.

L'imbrication de plusieurs commandes `for` dans les listes permet de construire le produit cartésien de deux listes ou de définir des listes de listes. Les résultats de ces exemples montrent que l'opérateur `for` est évalué de la gauche vers la droite et de l'intérieur vers l'extérieur des listes :

```
sage: [[x,y] for x in [1..2] for y in [6..9]]
  [[1,6],[1,7],[1,8],[1,9],[2,6],[2,7],[2,8],[2,9]]
sage: [10*x+y for x in [1..3] for y in [6..9]]
  [16, 17, 18, 19, 26, 27, 28, 29, 36, 37, 38, 39]

sage: [[10*x+y for x in [1..3]] for y in [6..9]]
  [[16,26,36], [17,27,37], [18,28,38], [19,29,39]]
sage: [10*x+1 for x in [1..3,6..9]]
  [11,21,31,61,71,81,91]

sage: map (lambda x,y : [x,y], [1..3],[6..9])
  [[1,6], [2,7], [3,8]]
```

Cette dernière commande `map` avec plusieurs listes arguments avance de façon synchronisée dans ces listes.

Enfin la commande `flatten` permet de concaténer des listes sur un ou plusieurs niveaux :

```
sage: L = [[1,2,[3]],[4,[5,6]],[7,[8,[9]]]]
sage: flatten (L, max_level=1)
  [1, 2,[3],4,[5,6], 7, [8,[9]]]
sage: flatten (L, max_level=2)
  [1, 2, 3, 4, 5, 6, 7, 8, [9]]
sage: flatten (L)      # dans ce cas =flatten (L, max_level=3)
  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ces manipulations élémentaires de listes interviennent de façon très utiles dans les autres branches de Sage ; l'exemple suivant calcule les premières dérivées itérées de $x\,e^x$; le premier argument de `diff` est l'expression à dériver, et le ou les suivants correspondent à la variable de dérivation, ces paramètres peuvent aussi être la liste des variables par rapport auxquelles ces dérivées sont effectuées :

```
sage: factor (diff (x*exp(x), [x,x]))
sage: map (lambda n : factor (diff (x*exp(x), n*[x])), [0..6])
sage: [factor (diff (x*exp(x), n*[x])) for n in [0..6]]
```

La commande `diff` possède plusieurs syntaxes. Les paramètres suivant la fonction f peuvent aussi bien être la liste des variables de dérivation que l'énumération de ces variables, ou l'ordre de la dérivée :

`diff(f(x),x,x,x)`, `diff(f(x),[x,x,x])`, $texttt{diff}(f(x), x, 3)$.

On peut aussi employer `diff(f(x),3)` pour les fonctions à une seule variable. Ces résultats se vérifient directement par la formule de Leibniz de dérivée itérée d'un produit de deux termes où les dérivées d'ordre 2 ou plus de x sont nulles :

$$(xe^x)^{(n)} = \sum_{k=0}^{n} \binom{n}{k} x^{(k)} (e^x)^{(n-k)} = (x+n)e^x.$$

3.2.3 Principales méthodes sur les listes

La méthode `reverse` renverse l'ordre des éléments d'une liste, et la méthode `sort` transforme la liste initiale en une liste triée :

```
sage: L=[1,8,5,2,9] ; L.reverse() ; L  # L devient [9,2,5,8,1]
sage: L.sort() ; L                     #       L = [1,2,5,8,9]
sage: L.sort(reverse=True) ; L         #       L = [9,8,5,2,1]
```

Ces deux méthodes opèrent à l'intérieur de la liste, et l'ancienne valeur de `L` est perdue.

Un premier argument optionnel à `sort` permet de choisir la relation d'ordre appliquée sous la forme d'une fonction `Ordre(x,y)` à deux paramètres. Le résultat doit être du type `int` des entiers « directement manipulés par

l'ordinateur » ; il est strictement négatif, nul ou positif, par exemple -1, 0 ou 1, selon que $x \prec y$, $x = y$ ou $x \succ y$. La liste transformée $(x_0, x_1, \cdots, x_{n-1})$ vérifie $x_0 \preceq x_1 \preceq \cdots \preceq x_{n-1}$.

L'ordre lexicographique de deux listes de nombres de même longueur est similaire à l'ordre alphabétique et est défini par cette équivalence en ignorant les premiers termes lorsqu'ils sont égaux deux à deux :

$$\begin{array}{ll} & P = (p_0, p_1, \cdots p_{n-1}) \prec_{lex} Q = (q_0, q_1, \cdots q_{n-1}) \\ \iff & \exists r \in \{0, \cdots, n-1\} \qquad (p_0, p_1, \cdots, p_{r-1}) = (q_0, q_1, \cdots, q_{r-1}) \\ & \qquad\qquad\qquad\qquad \texttt{et } p_r < q_r. \end{array}$$

La fonction suivante compare deux listes supposées de même longueur. Malgré la boucle *a priori* sans fin `while True`, les commandes `return` sortent directement de cette boucle et terminent la fonction. Le résultat est -1, 0 ou 1 selon que $P \prec_{lex} Q$, $P = Q$ ou $P \succ_{lex} Q$:

```
sage: def alpha (P,Q) :       # len(P) == len(Q) par hypothèse
....:   i=0
....:   while True :
....:     if i == len P : return int(0)
....:     elif P[i] < Q[i] : return int(-1)
....:     elif P[i] > Q[i] : return int(1)
....:     else : i = i+1
....:
sage: alpha ([2,3,4,6,5], [2,3,4,5,6]) # de réponse 1
```

La commande suivante trie cette liste de listes de même longueur en suivant l'ordre lexicographique. Cette fonction correspond par ailleurs à l'ordre implanté dans Sage pour comparer deux listes ; la commande `L.sort()` sans paramètre optionnel est équivalente :

```
sage: L = [[2,2,5],[2,3,4],[3,2,4],[3,3,3],[1,1,2],[1,2,7]]
sage: L.sort (cmp=alpha) ; L
  [[1,1,2],[1,2,7],[2,2,5],[2,3,4],[3,2,4],[3,3,3]]
```

La définition de l'ordre lexicographique homogène consiste d'abord à comparer les termes de même poids avant d'appliquer l'ordre lexicographique, où le poids est la somme des coefficients :

$$\begin{array}{l} P = (p_0, p_1, \cdots p_{n-1}) \prec_{lexH} Q = (q_0, q_1, \cdots q_{n-1}) \\ \iff \displaystyle\sum_{k=0}^{n-1} p_k < \sum_{k=0}^{n-1} q_k \ \texttt{ ou } \Big(\sum_{k=0}^{n-1} p_k = \sum_{k=0}^{n-1} q_k \ \texttt{ et } \ P \prec_{lex} Q \Big) \end{array}$$

Le code ci-dessous implante cet ordre homogène :

```
sage: def homogLex (P,Q) :
....:  sp = sum (P) ; sq = sum (Q)
....:  if sp < sq : return int(-1)
....:  elif sp > sq : return int(1)
....:  else : return alpha (P, Q)
sage: homogLex ([2,3,4,6,4], [2,3,4,5,6]) # de réponse -1
```

La fonction `sorted` de Sage est une fonction au sens mathématique du terme, elle prend une liste comme premier argument et, sans la modifier, renvoie comme résultat une liste triée, contrairement à la méthode `sort` qui modifie la liste en place.

Sage propose d'autres méthodes sur les listes, insertion d'un élément en fin de liste, concaténation en fin de liste, et dénombrement du nombre de répétitions d'un élément :

`L.append(x)`	est équivalent à `L[len(L):]=[x]`
`L.extend(L1)`	est équivalent à `L[len(L):]=L1`
`L.insert(i,x)`	est équivalent à `L[i:i]=[x]`
`L.count(x)`	est équivalent à `len (select (lambda t : t==x, L))`

Les commandes `L.pop(i)` et `L.pop()` suppriment l'élément d'indice i ou le dernier d'une liste, et renvoient la valeur de cet élément ; ces deux fonctions décrivent leur fonctionnement :

```
def pop1 (L, i) :
  a=L[i]
  L[i:i+1]=[]
  return a
```

```
def pop2 (L) :
  return pop1 (L,len(L)-1)
```

Par ailleurs `L.index(x)` renvoie l'indice du premier terme égal à x, et `L.remove(x)` enlève le premier élément de valeur x de la liste. Ces commandes provoquent une erreur si x n'appartient pas à la liste. Enfin la commande `del L[p:q]` est équivalente à `L[p:q]=[]`.

Contrairement à de nombreux autres langages informatiques, ces fonctions modifient la liste `L` sans créer de nouvelles listes.

3.2.4 Exemples de manipulation de listes

L'exemple suivant construit la liste des termes pairs et celle des termes impairs d'une liste donnée. Cette première solution parcourt deux fois la liste et effectue deux fois ces tests :

```
sage: def fct1 (L) :
....:   return [filter (lambda n : n % 2 == 0, L),\
....:           filter (lambda n : n % 2 == 1, L)]
sage: fct1 ([1..10])
  [[2,4,6,8,10], [1,3,5,7,9]]
```

Cette deuxième solution parcourt une seule fois la liste initiale et construit petit à petit ces deux listes résultats :

```
sage: def fct2 (L) :
....:   res0=[] ; res1=[]
....:   for k in L :
....:     if k%2==0 : res0.append(k) # ou res0[len(res0):]=[k]
....:     else : res1.append(k)      # ou res1[len(res1):]=[k]
....:   return [res0, res1]
```

Ce programme remplace la boucle `for` et les variables auxiliaires par un appel récursif et un paramètre supplémentaire :

```
sage: def fct3a (L,res0,res1) :
....:   if L==[] : return [res0, res1]
....:   elif L[0]%2==0 : return fct3a(L[1:],res0+[L[0]],res1)
....:   else : return fct3a (L[1:], res0, res1+[L[0]])
sage: def fct3 (L) : return fct3a (L, [], [])
```

Les paramètres `res0` et `res1` contiennent les premiers éléments déjà triés, et la liste paramètre `L` perd un terme à chaque appel récursif.

Le deuxième exemple ci-dessous extrait toutes les suites croissantes d'une liste de nombres ; trois variables sont nécessaires, la pemière `res` mémorise les suites croissantes déjà obtenues, la variable `debut` indique la position où la suite croissante en cours commence, et la variable `k` est l'indice de boucle :

```
sage: def sousSuites (L) :
....:   if L==[] : return []
....:   res = [] ; debut=0 ; k=1
....:   while k < len(L) : # 2 termes consécutifs sont définis.
....:     if L[k-1] > L[k] :
....:       res.append (L[debut:k]) ; debut = k
....:     k = k+1
....:   res.append (L[debut:k])
....:   return res
sage: sousSuites([1,4,1,5])           # renvoie [[1,4], [1,5]]
sage: sousSuites([4,1,5,1])           # puis [[4], [1,5], [1]]
```

Le corps de la boucle permet de passer au terme suivant de la liste. Si le test est vérifié, alors la sous-suite croissante en cours se termine, et il faut passer à une nouvelle sous-suite, sinon elle se prolonge au terme suivant.

L'instruction après la boucle ajoute au résultat final la sous-suite croissante en cours de parcours qui possède au moins un élément.

3.2.5 Chaînes de caractères

Les chaînes de caractères sont délimitées par des guillemets simples `'...'` ou éventuellement doubles `"..."`. Les chaînes délimitées par des guillemets simples peuvent contenir des guillemets doubles, et réciproquement. Les chaînes peuvent aussi être délimitées par des triples guillemets `'''...'''` et, dans ce cas peuvent être saisies sur plusieurs lignes et contenir des guillemets simples ou doubles.

```
sage: S='Ceci est une chaîne de caractères.'
```

Le caractère d'échappement est le signe \, il permet d'inclure des passages à la ligne par `\n`, des guillemets par `\"` ou par `\'`, le caractère de tabulation par `\t`, le signe d'échappement par `\\`. Les chaînes de caractères peuvent contenir des caractères accentués, et plus généralement des caractères Unicode quelconques :

```
sage: S ; print S
  'ceci est une cha\xc3\xaene de caract\xc3\xa8res.'
  ceci est une chaîne de caractères.
```

La comparaison des chaînes de caractères s'effectue en fonction du code interne de chaque caractère qui place les lettres accentuées à fin. La longueur d'une chaîne est obtenue par la fonction `len`, et la concaténation de chaînes se fait par les signes d'addition « `+` » et de multiplication « `*` ».

L'accès à une sous-chaîne de caractères de `S` s'effectue comme pour les listes par des crochets `S[n]`, `S[p:q]`, `S[p:]` et `S[:q]`, et le résultat est une chaîne de caractères. Le langage ne permet pas de modifier la chaîne initiale par une affectation de cette forme, pour cette raison le type *chaîne de caractères* est dit immuable.

La fonction `str` convertit son argument en une chaîne de caractères. La méthode `split` découpe une chaîne de caractères en listes de sous-chaînes au niveau des espaces :

```
sage: S='un deux trois quatre cinq six sept' ; L=S.split() ; L
  ['un', 'deux', 'trois', 'quatre', cinq', 'six', 'sept']
```

La bibliothèque très complète `re` des expressions régulières de Python peut aussi être utilisée pour la recherche des sous-chaînes et la reconnaissance de motifs et de mots.

3.2.6 Structure partagée ou dupliquée

Une liste entre crochets `[..]` peut être modifiée par des affectations sur ses éléments, par un changement du nombre de termes de la liste, ou des méthodes comme le tri `sort` ou le renversement `reverse`.

L'affectation d'une variable à une liste ne duplique pas la structure mais partage les mêmes données ; dans l'exemple suivant les listes `L1` et `L2` restent identiques ; la modification de l'une est visible sur l'autre :

```
sage: L1 = [11,22,33] ; L2 = L1 ;
sage: L1[1] = 222 ; L2.sort() ; L1,L2
  [11, 33, 222], [11, 33, 222]   # Les deux listes sont égales
sage: L1[2:3]=[]; L2[0:0]=[6,7,8] # [11,33] puis [6,7,8,11,33]
sage: L1 ; L2                     # L1 et L2 restent les mêmes
```

Au contraire les images de listes par `map`, les constructions de sous-listes par `L[p:q]`, `filter` ou `[..for..if..]`, la concaténation par `+` et `*`, et l'applatissement de listes par `flatten` dupliquent la structure des données.

Dans l'exemple précédent remplacer sur la première ligne `L2=L1` par l'une de ces six commandes change complètement les résultats suivants car les modifications d'une liste ne se propagent pas à l'autre. Les deux structures deviennent indépendantes ; ainsi l'affectation `L2=L1[:]` recopie la sous-liste de `L1` du premier au dernier terme, et donc duplique la structure complète de `L1` :

```
L2=[11,22,33]  L2=copy(L1)  L2=L1[:]
L2=[]+L1       L2=L1+[]     L2=1*L1
```

Le test du partage des structures de Sage s'effectue par l'opérateur binaire `is` ; si la réponse au test est vraie, alors toutes les modifications portent sur les deux variables à la fois :

```
sage: L1 = [11,22,33] ; L2 = L1 ; L3 = L1[:]
sage: [L1 is L2, L2 is L1, L1 is L3, L1==L3]
  [True, True, False, True]
```

Les opérations de copie opèrent sur un seul niveau de listes. Ainsi la modification à l'intérieur d'une liste de listes se propage malgré la copie de la structure au premier niveau :

```
sage: La = [1,2,3] ; L1 = [1,La] ; L2 = copy(L1)
sage: L1[1][0] = 5          # [1,[5,2,3]] pour L1 et L2
sage: [L1==L2, L1 is L2, L1[1] is L2[1]] # [True, False, True]
```

L'instruction suivante recopie la structure sur deux niveaux, alors que la fonction `copyRec` duplique récursivement les listes à tous les niveaux :

```
sage: map (copy, L)
sage: def copyRec (L)
....:  if type (L) == list : return map (copyRec, L)
....:  else : return L
```

L'ordre lexicographique inverse est défini à partir de l'ordre lexicographique sur des n-uplets énumérés à l'envers en renversant l'ordre appliqué à chaque élément :

$$\begin{array}{ll} & P = (p_0, p_1, \cdots p_{n-1}) \prec_{lexInv} Q = (q_0, q_1, \cdots q_{n-1}) \\ \iff \exists r \in \{0, \cdots, n-1\} & (p_{r+1}, \cdots, p_{n-1}) = (q_{r+1}, \cdots, q_{n-1}) \\ & \texttt{et } p_r > q_r. \end{array}$$

L'algorithme consiste donc à renverser l'ordre des éléments et l'ordre de comparaison. Cette commande renvoie `int(-1)`, `int(0)` ou `int(1)` pour être adaptée à la commande `sort`.

La programmation de l'ordre lexigraphique inverse peut se faire à partir de la fonction `alpha` définie précédemment qui implante l'ordre lexicographique. La recopie des listes P et Q est nécessaire pour effectuer le tri sans modifier les données triées. Plus précisément la fonction `lexInverse` renverse l'ordre des n-uplets par `reverse` et renverse l'ordre final par le résultat renvoyé $-(P_1 \prec_{lex} Q_1)$:

```
sage: def lexInverse (P, Q) :
....:   P1 = copy(P) ; P1.reverse()
....:   Q1 = copy(Q) ; Q1.reverse()
....:   return - alpha (P1, Q1)
```

Les modifications sur une liste passée en argument d'une fonction se répercutent de façon globale sur la liste car les fonctions ne recopient pas les structures de listes passées en argument. Ainsi une fonction effectuant uniquement `P.reverse()` à la place de `P1=copy(P)` et `P1.reverse()` modifie définitivement la liste P ; cet effet appelé *effet de bord* n'est généralement pas souhaité.

La variable P est une variable locale de la fonction indépendemment de tout autre variable globale de même nom P, mais cela est sans rapport avec les modifications apportées à l'intérieur d'une liste passée en argument.

3.2.7 Données modifiables ou immuables

Les listes permettent de structurer et de manipuler des données pouvant être modifiées. Pour cette raison ces structures sont qualifiées de modifiables ou de mutables.

Au contraire, Python permet aussi de définir des données figées ou immuables. La structure correspondante aux listes est appelée séquence ou tuple d'après son nom anglais, et est notée avec des parenthèses `(..)` à la place des crochets `[..]`. Une séquence à un seul argument est définie en plaçant explicitement une virgule après son argument.

```
sage: S0 = () ; S1 = (1,) ; S2 = (1,2)
sage: [1 in S1, 1==(1)]        # de réponse [True, True]
```

Les opérations d'accès aux séquences sont essentiellement les mêmes que celles sur les listes, par exemple la construction d'images de séquence par `map` ou d'extraction de sous-séquences par `filter`. Dans tous les cas le résultat est une liste, et cette commande `for` transforme une séquence en liste :

```
sage: S1=(1,4,9,16,25) ; [k for k in S1]
```

La commande `zip` regroupe terme à terme plusieurs listes ou séquences de façon équivalente à la commande `map` suivante :

```
sage: L1=[0..4] ; L2=[5..9]
sage: zip (L1, L2)   # renvoie [(0,5),(1,6),(2,7),(3,8),(4,9)]
sage: map (lambda x,y:(x,y), L1, L2)
```

3.2.8 Ensembles finis

Contrairement aux listes, la notion d'ensemble tient uniquement compte de la présence ou non d'un élément, sans définir sa position ni le nombre de répétitions de cet élément. Sage manipule les ensembles de cardinal fini à partir de la fonction `Set` appliquée à la liste des éléments. Le résultat est affiché entre accolades :

```
sage: E = Set([1,2,4,8,2,2,2]) ; F = Set([7,5,3,1]) ; E,F
  {8, 1, 2, 4}, {1, 3, 5, 7}
```

Les éléments d'un ensemble doivent être de type immuable, ceci empêche la construction d'ensemble de listes, mais permet les ensembles de séquences.

L'opérateur d'appartenance `in` teste l'appartenance d'un élément à un ensemble, et Sage autorise les opérations de réunion d'ensembles par `+` ou `|`, d'intersection par `&`, de différence `-`, et de différence symétrique par `^^` :

```
sage: 5 in E, 5 in F, E+F==F|E ; E&F, E-F, E^^F
  False, True, True
  {1}, {8, 2, 4}, {2, 3, 4, 5, 7, 8}
```

La fonction `len(E)` renvoie le cardinal d'un tel ensemble fini. Les opérations `map`, `filter` et `for..if...` s'appliquent aux ensembles comme aux séquences, et les résultats sont des listes. L'accès à un élément s'effectue par `E[k]`. Les deux commandes ci-dessous construisent de façon équivalente la liste des éléments d'un ensemble :

```
sage: [E[k] for k in [0..len(E)-1] ; [t for t in E]
```

La fonction suivante définit l'inclusion d'un ensemble E dans F à partir de la réunion :

```
sage: def inclus (E,F) : return E+F==F
```

Contrairement aux listes, les ensembles sont de type immuable et ne sont donc pas modifiables ; leurs éléments doivent aussi être immuables, les ensembles de tuples et les ensembles d'ensembles sont donc possibles mais on ne peut pas construire des ensembles de listes :

```
sage: Set ([Set([]),Set([1]),Set([2]),Set([1,2])])
sage: Set ([    (),     (1,),    (2,),    (1,2) ])
  {(1, 2), (2,), (), (1,)}
  {{1, 2}, {}, {2}, {1}}
```

Cet fonction énumère tous les sous-ensembles d'un ensemble par une méthode récursive :

```
sage: def Parties (EE) :
....:   if EE==Set([]) : return Set([Set([])])
....:   else :
....:     return avecOuSansElt (EE[0], Parties(Set(EE[1:])))
sage: def avecOuSansElt (a,E) :
....:   return Set (map (lambda F : Set([a])+F, E)) + E
```

La fonction `avecOuSansElt(a,E)` prend un ensemble E de sous-ensembles, et construit l'ensemble ayant deux fois plus d'éléments qui sont ces mêmes sous-ensembles et ceux auquels l'élément a a été ajouté. La construction récursive commence avec l'ensemble à un élément $E = \{\emptyset\}$.

```
sage: Parties (Set([1,2,3]))    # 8 ensembles, de [] à [1,2,3]
  {{3}, {1,2}, {}, {2,3}, {1}, {1,3}, {1,2,3}, {2}}
```

3.2.9 Dictionnaires

Enfin Python, et donc Sage, intègre la notion de dictionnaire. Comme un annuaire, un dictionnaire associe une valeur à chaque entrée.

Les entrées d'un dictionnaire sont de n'importe quel type immuable, nombres, chaînes de caractères, séquences, etc. La syntaxe est comparable à celle des listes par des affectations à partir du dictionnaire vide `dict()` pouvant être abrégé en `{}` :

```
sage: D={}; D['un']=1; D['deux']=2; D['trois']=3; D['dix']=10
sage: D['deux']+D['trois']                          # de valeur 5
```

L'exemple précédent détaille donc comment ajouter une entrée dans un dictionnaire, et comment accéder à un champ par `D[..]`.

L'opérateur `in` teste si une entrée fait partie d'un dictionnaire, et les commandes `del D[x]` et `D.pop(x)` effacent l'entrée x de ce dictionnaire.

L'exemple suivant indique comment un dictionnaire peut représenter une application sur un ensemble fini :

$$E = \{a_0, a_1, a_2, a_3, a_4, a_5\}, \quad \begin{array}{lll} f(a_0) = b_0, & f(a_1) = b_1, & f(a_2) = b_2, \\ f(a_3) = b_0, & f(a_4) = b_3, & f(a_5) = b_3. \end{array}$$

Les méthodes sur les dictionnaires sont comparables celles portant sur les autres structures énumérées ; le code ci-dessous implante la fonction précédente et construit l'ensemble de départ E et l'ensemble image $\mathrm{Im}\, f = f(E)$ par les méthodes `keys` et `values` :

```
sage: D = {'a0':'b0','a1':'b1','a2':'b2',\
....:      'a3':'b0','a4':'b3','a5':'b3'}
sage: E  = Set (D.keys()) ; Imf = Set (D.values())
sage: Imf == Set (map (lambda t:D[t], E))     # est équivalent
```

Cette dernière commande traduit directement la définition mathématique $\mathrm{Imf} = \{f(x)|x \in E\}$. Les dictionnaires peuvent aussi être construits à partir de listes ou de séquences `[clé,valeur]` par la commande suivante :

```
dict(['a0','b0'], ['a1','b1'], ...)
```

Les deux commandes suivantes appliquées aux entrées du dictionnaire ou au dictionnaire lui-même sont par construction équivalentes à la méthode `D.values()` :

```
map (lambda t :D[t],D)    map (lambda t :D[t],D.keys())
```

Le test suivant sur le nombre d'éléments détermine si l'application représentée par `D` est injective, `len(D)` étant le nombre d'entrées dans le dictionnaire :

```
sage: def injective (D) :
....:   len (D) == len (Set (D.values()))
```

Les deux premières commandes ci-dessous construisent l'image directe $f(F)$ et l'image réciproque $f^{-1}(G)$ des sous-ensembles F et G d'une application définie par le dictionnaire D ; la dernière définit un dictionnaire `DR` correspondant à l'application réciproque f^{-1} d'une application f supposée bijective :

```
sage: Set ([D[t] for t in F])
sage: Set ([t for t in D if D[t] in G])
sage: DR = dict ((D[t],t) for t in D)
```

4 Graphiques

La visualisation de fonctions d'une ou deux variables, d'une série de données, facilite la perception de phénomènes mathématiques ou physiques et permet de conjecturer des résultats efficacement. Dans ce chapitre, on illustre sur des exemples les capacités graphiques de Sage.

4.1 Courbes en 2D

Une courbe plane peut être définie de plusieurs façons : comme graphe d'une fonction d'une variable, par un système d'équations paramétriques, par une équation en coordonnées polaires, ou par une équation implicite. Nous présentons ces quatre cas, puis donnons quelques exemples de tracés de données.

4.1.1 Représentation graphique de fonctions

Pour tracer le graphe d'une fonction symbolique ou d'une fonction Python sur un intervalle $[a, b]$, on dispose de deux syntaxes : `plot(f(x), a, b)` ou `plot(f(x), x, a, b)`.

```
sage: plot(x * sin(1/x), x, -2, 2, plot_points=500)
```

Parmi les nombreuses options de la fonction `plot`, citons :
- `plot_points` (valeur par défaut : 200) : détermine le nombre minimal de points calculés ;
- `xmin` et `xmax` : détermine les bornes de l'intervalle sur lequel est tracé la fonction ;

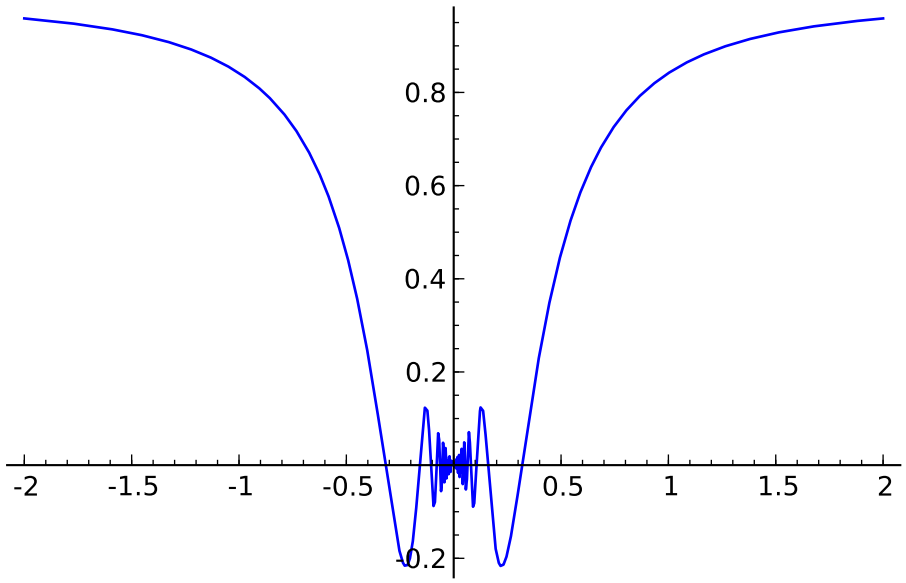


FIG. 4.1 – Graphe de $x \mapsto x \sin \frac{1}{x}$.

- `color` : détermine la couleur du tracé, soit par un triplet RGB, une chaîne de caractère (par ex. `'blue'`), ou par une couleur HTML (par ex. `'#aaff0b'`) ;
- `detect_poles` (valeur par défaut : `False`) : permet de tracer ou non l'asymptote verticale correspondante au pôle d'une fonction ;
- `alpha` permet de régler la transparence du trait ;
- `thickness` détermine l'épaisseur du trait ;
- `linestyle` détermine la nature du trait : tracé en pointillés (`:`), traits et pointillés (`-.`), trait continu (`-`).

Pour visualiser le tracé, on peut placer l'objet graphique dans une variable, puis utiliser la commande `show`, en précisant par exemple les valeurs minimales et maximales de l'ordonnée (`g.show(ymin=-1, ymax=3)`) ou alors en choisissant le rapport d'affinité (`g.show(aspect_ratio=1)` pour un tracé en repère orthonormé).

La figure produite peut être exportée grâce à la commande `save` vers différents formats : `.png`, `.ps`, `.eps`, `.svg` et `.sobj` :

```
g.save(nom, aspect_ratio=1, xmin=-1, xmax=3, ymin=-1, ymax=3)
```

Pour inclure une telle figure dans un document LaTeX à l'aide de la commande `includegraphics`, on choisit l'extension `eps` (PostScript encapsulé) si le document est compilé avec `latex` et l'extension `pdf` (plutôt que `png` pour obtenir une meilleure résolution) si le document est compilé avec `pdflatex`.

Exemple. Traçons sur un même graphique la fonction sinus et ses premiers polynômes de Taylor en 0.

```
def p(x, n): return(taylor(sin(x), x, 0, n))
xmax = 15 ; n = 15
g = plot(sin(x), x, -xmax, xmax)
for d in range(n):
     g += plot(p(x, 2 * d + 1), x, -xmax, xmax,\
       color=(1.7*d/(2*n), 1.5*d/(2*n), 1-3*d/(4*n)))
g.show(ymin=-2, ymax=2)
```

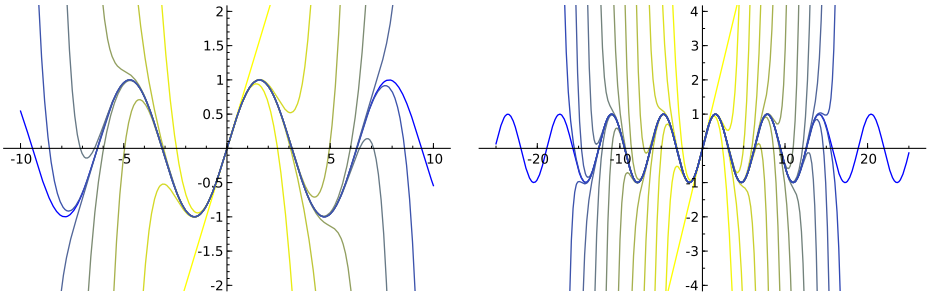


FIG. 4.2 – Quelques polynômes de Taylor de sinus en 0.

On aurait également pu effectuer une animation, pour observer combien le polynôme de Taylor de la fonction sin en 0 approche de mieux en mieux cette fonction lorsque le degré augmente. Si l'on souhaite sauvegarder l'animation, il suffit de l'enregistrer au format `gif`.

```
sage: a = animate([[sin(x), taylor(sin(x), x, 0, 2*k+1)]\
          for k in range(0, 14)], xmin=-14, xmax=14,\
          ymin=-3, ymax=3, figsize=[8, 4])
sage: a.show(); a.save('chemin/animation.gif')
```

Revenons à la fonction `plot` pour observer, par exemple, le phénomène de Gibbs.

Exemple. Traçons la somme partielle d'ordre 20 de la série de Fourier de la fonction créneau.

```
f2(x) = 1; f1(x) = -1
f = Piecewise([[(-pi,0),f1],[(0,pi),f2]])
S = f.fourier_series_partial_sum(20,pi); S
g = plot(S, x, -8, 8, color='blue')
scie(x) = x - 2 * pi * floor((x + pi) / (2 * pi))
g += plot(scie(x) / abs(scie(x)), x, -8, 8, color='red')
```

Dans le code ci-dessus, `f` est une fonction par morceaux définie sur $[-\pi;\pi]$ à l'aide de l'instruction `piecewise`. Pour représenter le prolongement de `f` par 2π-périodicité, le plus simple est d'en chercher une expression valable

pour tout réel (en l'occurrence `scie/abs(scie)`). La somme des 20 premiers termes de la série de Fourier vaut :

$$S = 4\,\frac{\sin(x)}{\pi} + \frac{4}{3}\,\frac{\sin(3\,x)}{\pi} + \frac{4}{5}\,\frac{\sin(5\,x)}{\pi} + \frac{4}{7}\,\frac{\sin(7\,x)}{\pi} + \frac{4}{9}\,\frac{\sin(9\,x)}{\pi} + \frac{4}{11}\,\frac{\sin(11\,x)}{\pi} + \frac{4}{13}\,\frac{\sin(13\,x)}{\pi} + \frac{4}{15}\,\frac{\sin(15\,x)}{\pi} + \frac{4}{17}\,\frac{\sin(17\,x)}{\pi} + \frac{4}{19}\,\frac{\sin(19\,x)}{\pi}$$

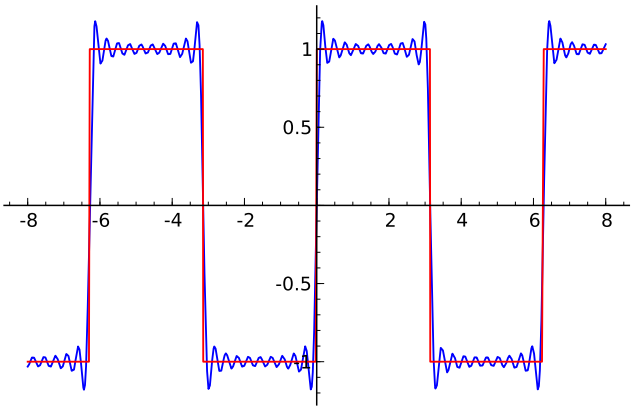


FIG. 4.3 – Décomposition de la fonction créneaux en série de Fourier.

4.1.2 Courbes paramétrées

Les tracés de courbes paramétrées ($x = f(t)$, $y = g(t)$) sont réalisés par la commande `parametric_plot((f(t), g(t)), (t, a, b))` où $[a, b]$ est l'intervalle parcouru par le paramètre.

Représentons par exemple la courbe paramétrée d'équations :

$$\begin{cases} x(t) = \cos(t) + \frac{1}{2}\cos(7t) + \frac{1}{3}\sin(17t), \\ y(t) = \sin(t) + \frac{1}{2}\sin(7t) + \frac{1}{3}\cos(17t). \end{cases}$$

```
sage: t = var('t')
sage: x = cos(t) + cos(7*t)/2 + sin(17*t)/3
sage: y = sin(t) + sin(7*t)/2 + cos(17*t)/3
sage: g = parametric_plot( (x, y), (t, 0, 2*pi))
sage: g.show(aspect_ratio=1)
```

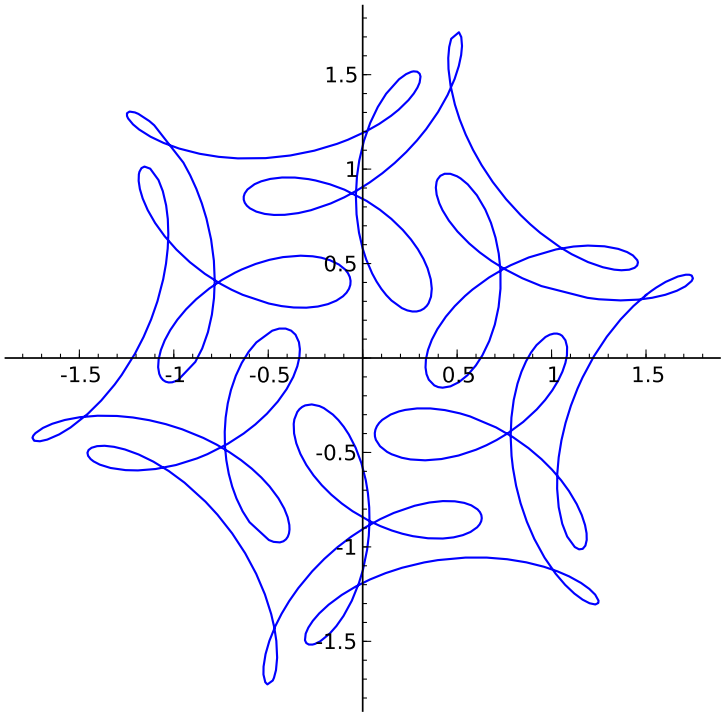


FIG. 4.4 – Courbe $\begin{cases} x(t) = \cos(t) + \frac{1}{2}\cos(7t) + \frac{1}{3}\sin(17t), \\ y(t) = \sin(t) + \frac{1}{2}\sin(7t) + \frac{1}{3}\cos(17t). \end{cases}$

4.1.3 Courbes en coordonnées polaires

Les tracés de courbes en coordonnées polaires $\rho = f(\theta)$ sont réalisés par la commande `polar_plot(rho(theta),(theta,a,b))` où $[a, b]$ est l'intervalle parcouru par le paramètre.

Représentons par exemple les conchoïdes de rosace d'équation polaire $\rho(\theta) = 1 + e \cdot \cos n\theta$ lorsque $n = 20/19$ et $e \in \{2, 1/3\}$.

```
t = var('t'); e, n = 2, 20/19
g1 = polar_plot(1+e*cos(n*t),(t,0,n*38*pi),plot_points=5000)
e, n = 1/3, 20/19
g2 = polar_plot(1+e*cos(n*t),(t,0,n*38*pi),plot_points=5000)
g1.show(aspect_ratio=1); g2.show(aspect_ratio=1)
```

Exercice 12. Représenter la famille de conchoïdes de Pascal d'équation polaire $\rho(\theta) = a + \cos\theta$ en faisant varier le paramètre a de 0 à 1 avec un pas de $0,1$.

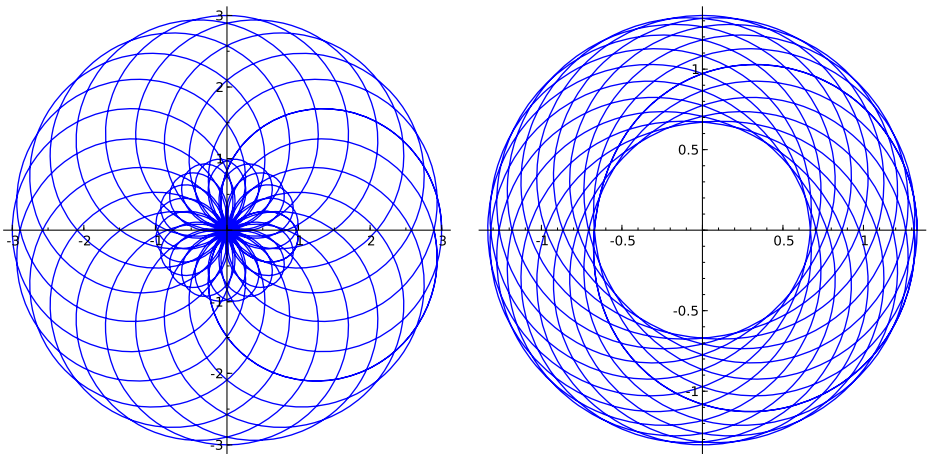


FIG. 4.5 – Rosaces d'équation $\rho(\theta) = 1 + e \cdot \cos n\theta$.

4.1.4 Courbe définie par une équation implicite

Pour représenter une courbe donnée par une équation implicite, on utilise la fonction `implicit_plot(f(x, y), (x, a, b), (y, c, d))` ; cependant, on peut aussi utiliser la commande `complex_plot` qui permet de visualiser en couleur les lignes de niveau d'une fonction à deux variables.

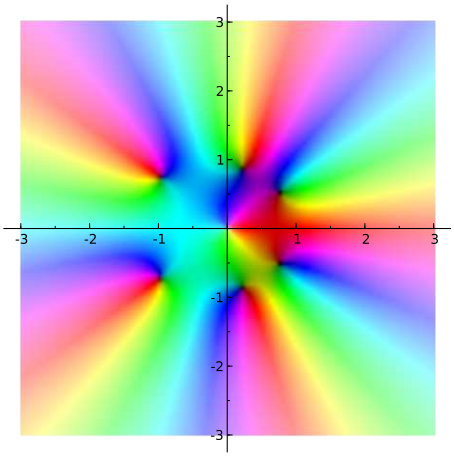
Exemple. Représentons la courbe donnée par l'équation implicite suivante :

$$\mathcal{C} = \left\{ z \in \mathbb{C} \ , \ |\cos(z^4)| = 1 \right\}$$

```
sage: var('z'); g1 = complex_plot(abs(cos(z^4)) - 1,\
                (-3, 3), (-3, 3), plot_points=400)
sage: f = lambda x, y : (abs(cos((x + I * y) ** 4)) - 1)
sage: g2 = implicit_plot(f, (-3, 3), (-3, 3), plot_points=400)
sage: g1.show(aspect_ratio=1); g2.show(aspect_ratio=1)
```

Voici un autre exemple de tracé de fonction donnée par une équation complexe :

```
sage: f(z) = z^5 + z - 1 + 1/z
sage: complex_plot(f,\
      (-3, 3), (-3, 3))
```



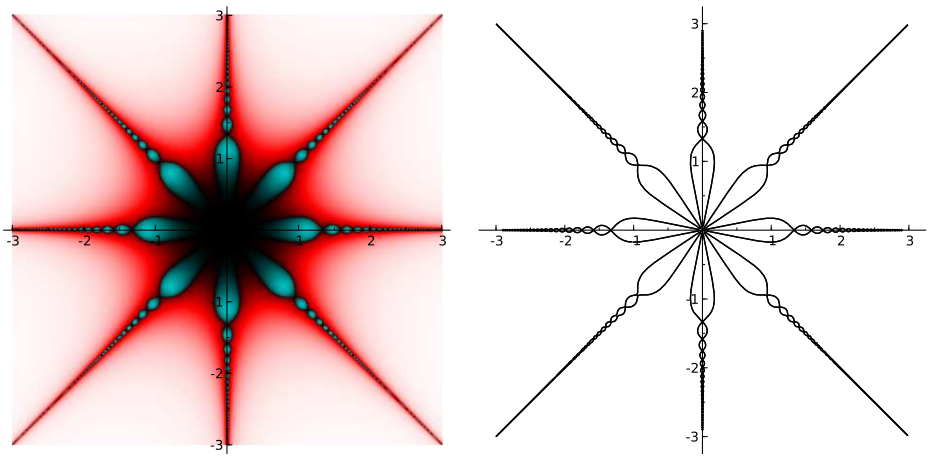


FIG. 4.6 – Courbe définie par l'équation $\left|\cos(z^4)\right| = 1$.

4.1.5 Tracé de données

Pour tracer un diagramme en rectangles, on dispose de deux fonctions assez différentes. Tout d'abord, la commande `bar_chart` prend en argument une liste d'entiers et trace simplement des barres verticales dont la hauteur est donnée par les éléments de la liste (dans leur ordre d'apparition dans la liste). Notons que l'option `width` permet de choisir la largeur des rectangles.

```
sage: bar_chart([randrange(15) for i in range(20)], color='red')
sage: bar_chart([x^2 for x in range(1,20)], width=0.2)
```

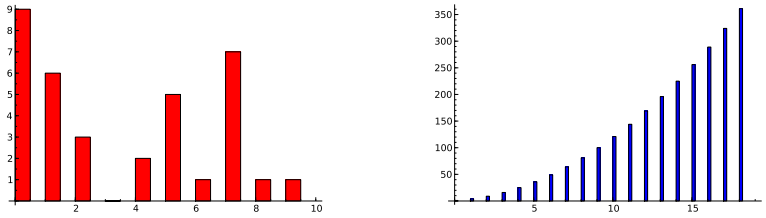
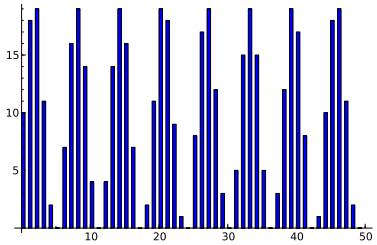


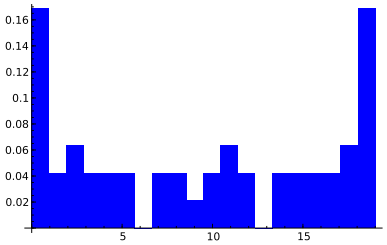
FIG. 4.7 – Diagrammes en rectangles.

En revanche, pour tracer l'histogramme des fréquences d'une donnée statistique donnée sous forme d'une liste de flottants, on utilise la fonction `plot_histogram` : les valeurs de la liste sont triées et réparties dans des intervalles (le nombre d'intervalles étant fixé par la variable `bins` dont la valeur par défaut vaut 50) ; la hauteur de la barre pour chaque intervalle étant égale à la fréquence correspondante.

```
sage: liste = [10 + floor(10*sin(i)) for i in range(100)]
sage: bar_chart(liste)
sage: finance.TimeSeries(liste).plot_histogram(bins=20)
```



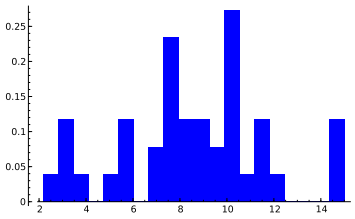
(A) Tracé avec `bar_chart`



(B) Tracé avec `plot_histogram`

Il est fréquent que la liste statistique à étudier soit stockée dans une feuille de calcul obtenue à l'aide d'un tableur. Le module `csv` de Python permet alors d'importer les données depuis un fichier enregistré au format `csv`. Par exemple, supposons que l'on souhaite tracer l'histogramme des notes d'une classe se trouvant dans la colonne 3 du fichier nommé `ds01.csv`. Pour extraire les notes de cette colonne, on utilise les instructions suivantes (en général les premières lignes comportent du texte d'où la nécessité de gérer les erreurs éventuelles lors de la conversion en flottant grâce à l'instruction `try`) :

```
import csv
reader = csv.reader(open("ds01.csv"))
notes = []; liste = []
for ligne in reader:
    notes.append(ligne[3])
for i in notes:
    try:
        f = float(i)
    except ValueError:
        pass
    else:
        liste.append(f)
finance.TimeSeries(liste).plot_histogram(bins=20)
```



Pour tracer une liste de points reliés (resp. un nuage de points), on utilise la commande `line(p)` (resp. `point(p)`), `p` désignant une liste de listes à deux éléments désignant abscisse et ordonnée des points à tracer.

Exemple. (*Marche aléatoire*) Partant d'un point origine O une particule se déplace d'une longueur ℓ, à intervalle de temps régulier t, à chaque fois dans une direction quelconque, indépendante des directions précédentes. Représentons un exemple de trajectoire pour une telle particule. Le segment rouge relie la position initiale à la position finale.

```
from random import *
n, l, x, y = 10000, 1, 0, 0; p = [[0, 0]]
```

```
for k in range(n):
    theta = (2 * pi * random()).n(digits=5)
    x, y = x + l * cos(theta), y + l * sin(theta)
    p.append([x, y])
g1 = line([p[n], [0, 0]], color='red', thickness=2)
g1 += line(p, thickness=.4); g1.show(aspect_ratio=1)
```

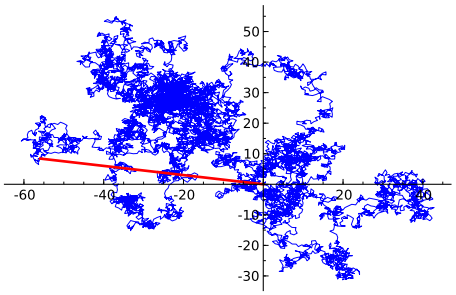


FIG. 4.8 – Marche aléatoire

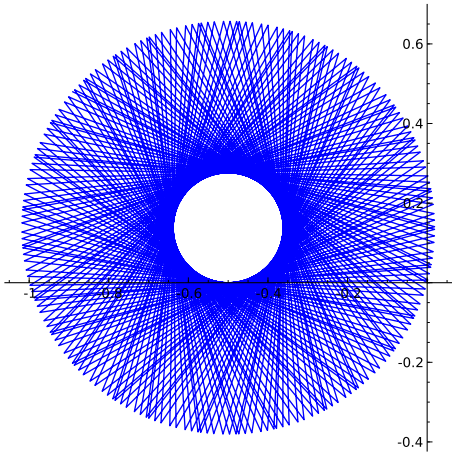
Exemple. (*Suites équiréparties*) Étant donnée une suite réelle $(u_n)_{n\in\mathbb{N}^*}$, on construit la ligne polygonale dont les sommets successifs sont les points d'affixe $z_N = \sum_{n\leqslant N} e^{2i\pi u_n}$. Si la suite est équirépartie modulo 1, la ligne brisée ne s'éloigne pas trop rapidement de l'origine. On peut alors conjecturer la régularité de la répartition de la suite à partir du tracé de la ligne brisée. Traçons la ligne polygonale dans les cas suivants :

- $u_n = n\sqrt{2}$ et $N = 200$,
- $u_n = n\ln(n)\sqrt{2}$ et $N = 10000$,
- $u_n = \mathrm{E}(n\ln(n))\sqrt{2}$ et $N = 10000$ (où E désigne la partie entière),
- $u_n = p_n\sqrt{2}$ et $N = 10000$ (ici p_n est le n-ème nombre premier).

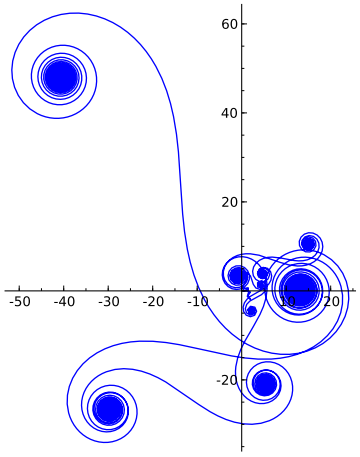
La figure 4.9 s'obtient de la manière suivante (ici pour $u_n = n\sqrt{2}$) :

```
length = 200; n = var('n')
u(n) = n * sqrt(2)
z(n) = exp(2 * I * pi * u(n))
vertices = [CC(0, 0)]
for n in range(1, length):
    vertices.append(vertices[n - 1] + CC(z(n)))
line(vertices).show(aspect_ratio=1)
```

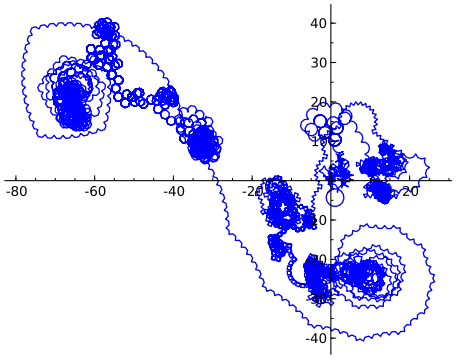
On remarque que la première courbe est particulièrement régulière, ce qui permet de prévoir que l'équirépartition de $n\sqrt{2}$ est de nature déterministe. Dans le cas de la suite $u_n = n\ln(n)\sqrt{2}$ l'examen des valeurs prises donne l'impression d'un engendrement aléatoire. Cependant, la courbe associée est remarquablement bien structurée. La troisième courbe, qui pourrait *a priori* ressembler plus à la quatrième, présente le même type de structures



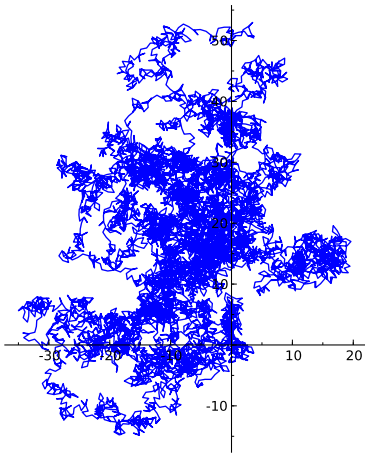
(A) Cas $u_n = n\sqrt{2}$



(B) Cas $u_n = n\ln(n)\sqrt{2}$



(C) Cas $u_n = \mathrm{E}(n\ln(n))\sqrt{2}$



(D) Cas $u_n = p_n\sqrt{2}$

FIG. 4.9 – Suites équiréparties

que la deuxième. Enfin, le tracé de la quatrième courbe fait apparaître la nature profondément différente de la répartition modulo $1/\sqrt{2}$ de la suite des nombres premiers : les spirales ont disparu et l'allure ressemble à la courbe que l'on obtiendrait dans le cas d'une suite de nombres aléatoires u_n. Il semble donc que « les nombres premiers occupent tout le hasard qui leur est imparti... »

Pour une interprétation détaillée des courbes obtenues, on se reportera au « Que sais-je ? » *Les nombres premiers* de Gérald Tenenbaum et Michel Mendès France [TMF00].

Exercice 13. (*Tracé des termes d'une suite récurrente*) On considère la

suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} u_0 = a, \\ \forall n \in \mathbb{N},\ u_{n+1} = \left| u_n^2 - \frac{1}{4} \right|. \end{cases}$$

Représenter graphiquement le comportement de la suite en construisant une liste formée des points $[[u_0, 0], [u_0, u_1], [u_1, u_1], [u_1, u_2], [u_2, u_2], \dots]$.

4.1.6 Tracé de solution d'équation différentielle

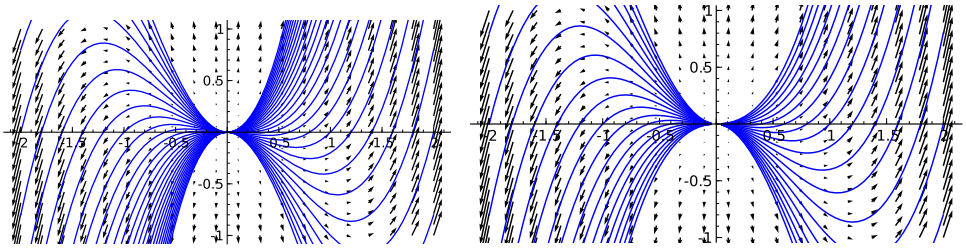
On peut combiner les commandes précédentes pour tracer des solutions d'équations ou de systèmes différentiels. Pour résoudre symboliquement une équation différentielle ordinaire, on utilise la fonction `desolve` dont l'étude fait l'objet du chapitre 11. Pour résoudre numériquement une équation différentielle, Sage nous fournit plusieurs outils : `desolve_rk4` (qui utilise la même syntaxe que la fonction `desolve` et qui suffit pour résoudre les équations différentielles ordinaires rencontrées en Licence), `odeint` (qui provient du module SciPy) et enfin `ode_solver` (qui provient de la librairie GSL dont l'utilisation est détaillée dans la section 6.2). Les fonctions `desolve_rk4` et `odeint` renvoient une liste de points qu'il est alors aisé de tracer à l'aide de la commande `line` ; c'est celles que nous utiliserons dans cette section pour tracer des solutions numériques.

Exemple. (*Équation différentielle linéaire, du premier ordre, non résolue*) Traçons les courbes intégrales de l'équation différentielle $xy' - 2y = x^3$.

```
1   x = var('x'); y = function('y',x)
2   DE = x*diff(y, x) == 2*y + x^3
3   desolve(DE, [y,x])              # answer : (c + x)*x^2
4   sol = []
5   for i in srange(-2, 2, 0.2):
6       sol.append(desolve(DE, [y, x], ics=[1, i]))
7       sol.append(desolve(DE, [y, x], ics=[-1, i]))
8   g = plot(sol, x, -2, 2)
9   y = var('y')
10  g += plot_vector_field((x, 2*y+x^3), (x, -2, 2), (y, -1, 1))
11  g.show(ymin=-1, ymax=1)
```

Pour diminuer le temps de calcul, il serait préférable ici de saisir « à la main » la solution générale de l'équation et de créer une liste de solutions particulières (comme cela est fait dans la correction de l'exercice 14) plutôt que de résoudre plusieurs fois de suite l'équation différentielle avec des conditions initiales différentes. On aurait également pu effectuer une résolution numérique de cette équation (à l'aide de la fonction `desolve_rk4`) pour en tracer les courbes intégrales ; il suffit pour cela de remplacer les lignes 3 à 8 (comprise) du script précédent par les lignes suivantes :

```
    g = Graphics()              # crée un graphique vide
    for i in srange(-1, 1, 0.1):
        g += line(desolve_rk4(DE, y, ics=[1, i],\
                              step=0.05, end_points=[0,2]))
        g += line(desolve_rk4(DE, y, ics=[-1, i],\
                              step=0.05, end_points=[-2,0]))
```



(A) Résolution symbolique (B) Résolution numérique

FIG. 4.10 – Tracé des courbes intégrales de $xy' - 2y = x^3$.

Comme on le voit sur l'exemple précédent, la fonction `desolve_rk4` prend en argument une équation différentielle (ou le membre de droite f de l'équation écrite sous forme résolue $y' = f(y,t)$), le nom de la fonction inconnue, la condition initiale, le pas et l'intervalle de résolution. L'argument optionnel `output` permet de préciser la sortie de la fonction : la valeur par défaut `'list'` renvoie une liste (ce qui est utile si on veut superposer des graphiques comme dans notre exemple), `'plot'` affiche le tracé de la solution et `'slope_field'` y ajoute le tracé des pentes des courbes intégrales.

Exercice 14. Tracer les courbes intégrales de l'équation $x^2y' - y = 0$.

Donnons à présent un exemple d'utilisation de la fonction `odeint` du module SciPy.

Exemple. (*Équation différentielle non linéaire, résolue, du premier ordre*) Traçons les courbes intégrales de l'équation $y'(t) + \cos(y(t) \cdot t) = 0$.

```
import scipy; from scipy import integrate
f = lambda y, t: - cos(y * t)
t = srange(0, 5, 0.1); p = Graphics()
for k in srange(0, 10, 0.15):
      y = integrate.odeint(f, k, t)
      p += line(zip(t, flatten(y)))
t = srange(0, -5, -0.1); q = Graphics()
for k in srange(0, 10, 0.15):
      y = integrate.odeint(f, k, t)
```

```
        q += line(zip(t, flatten(y)))
y = var('y')
v = plot_vector_field((1, -cos(x * y)), (x,-5,5), (y,-2,11))
g = p + q + v; g.show()
```

La fonction `odeint` prend en argument le second membre de l'équation différentielle (écrite sous forme résolue), une ou plusieurs conditions initiales, et enfin l'intervalle de résolution ; elle renvoie ensuite un tableau du type `numpy.ndarray` que l'on aplatit à l'aide de la commande `flatten` [1] déjà vue au § 3.2.2 et que l'on réunit avec le tableau `t` grâce à la commande `zip` avant d'effectuer le tracé de la solution approchée. Pour ajouter le champ des vecteurs tangents aux courbes intégrales, on a utilisé la commande `plot_vector_field`.

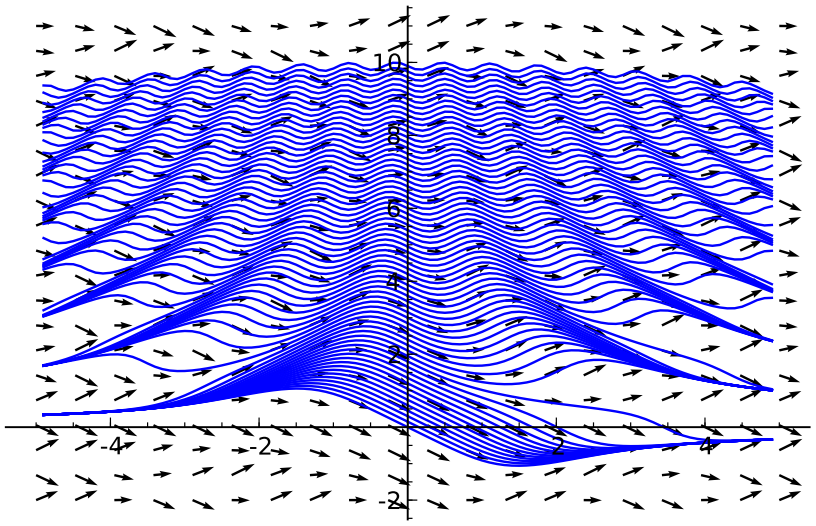


FIG. 4.11 – Tracé des courbes intégrales de $y'(t) + \cos(y(t) \cdot t) = 0$.

Exemple. (*Modèle proie-prédateur de Lokta-Volterra*) On souhaite représenter graphiquement l'évolution d'une population de proies et de prédateurs évoluant suivant un système d'équation du type Lokta-Volterra :

$$\begin{cases} \frac{du}{dt} = au - buv, \\ \frac{dv}{dt} = -cv + dbuv. \end{cases}$$

où u désigne le nombre de proies (par ex. des lapins), v le nombre de prédateurs (par ex. des renards). De plus, a, b, c, d sont des paramètres qui décrivent l'évolution des populations : a caractérise la croissance naturelle

[1] On aurait pu utiliser également la fonction `ravel` de NumPy qui évite de créer un nouvel objet liste et donc optimise l'utilisation de la mémoire.

du nombre de lapins en l'absence de renards, b la décroissance du nombre de lapins due à la présence de prédateurs, c la décroissance du nombre de renards en l'absence de proies, enfin d indique combien il faut de lapins pour qu'apparaisse un nouveau renard.

```
import scipy; from scipy import integrate
a, b, c, d = 1., 0.1, 1.5, 0.75
def dX_dt(X, t=0):   # Renvoie l'augmentation des populations
    return [ a*X[0] -   b*X[0]*X[1] ,
            -c*X[1] + d*b*X[0]*X[1] ]
t = srange(0, 15, .01)                      # échelle de temps
X0 = [10, 5]  # conditions initiales : 10 lapins et 5 renards
X = integrate.odeint(dX_dt, X0, t)     # résolution numérique
lapins, renards =  X.T         # raccourcis de  X.transpose()
p = line(zip(t, lapins), color='red') # tracé du nb de lapins
p += text("Lapins",(12,37), fontsize=10, color='red')
p += line(zip(t, renards), color='blue')# idem pr les renards
p += text("Renards",(12,7), fontsize=10, color='blue')
p.axes_labels(["temps", "population"]); p.show(gridlines=True)

### Deuxième graphique :
n = 11;  L = srange(6, 18, 12 / n); R = srange(3, 9, 6 / n)
CI = zip(L, R)                # liste des conditions initiales
def g(x,y):
    v = vector(dX_dt([x, y]))  # pour un tracé plus lisible,
    return v/v.norm()          # on norme le champ de vecteurs
x, y = var('x, y')
q = plot_vector_field(g(x, y), (x, 0, 60), (y, 0, 36))
for j in range(n):
    X = integrate.odeint(dX_dt, CI[j], t)        # résolution
    q += line(X, color=hue(.8-float(j)/(1.8*n))) #  graphique
q.axes_labels(["lapins","renards"]); q.show()
```

Exercice 15. (*Modèle proie-prédateur*) Réécrire le script précédent en utilisant `desolve_system_rk4` à la place de `odeint`.

Exercice 16. (*Un système différentiel autonome*) Tracer les courbes intégrales du système différentiel suivant :

$$\begin{cases} \dot{x} = y, \\ \dot{y} = 0.5y - x - y^3. \end{cases}$$

Exercice 17. (*Écoulement autour d'un cylindre avec effet Magnus*) On superpose à un écoulement simple autour d'un cylindre de rayon a, un vortex de paramètre α, ce qui modifie la composante orthoradiale de vitesse. On se

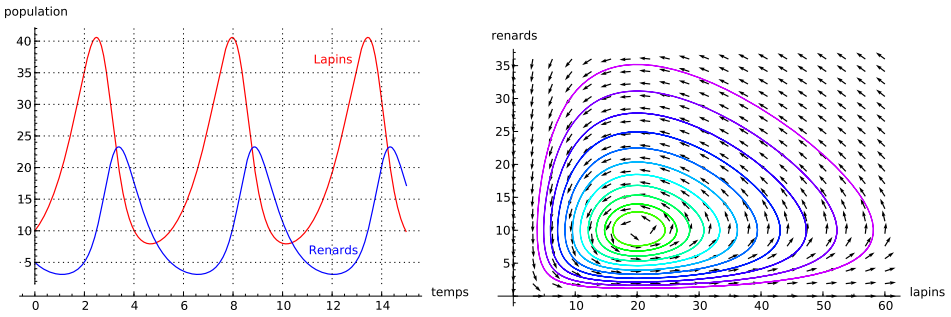


FIG. 4.12 – Étude d'un système proies-prédateurs.

place dans un repère centré sur le cylindre, et on travaille en coordonnées cylindriques dans le plan $z = 0$, autrement dit en coordonnées polaires. Les composantes de la vitesse sont alors :

$$v_r = v_0 \cos(\theta)\left(1 - \frac{a^2}{r^2}\right); \quad \text{et} \quad v_\theta = -v_0 \sin(\theta)\left(1 + \frac{a^2}{r^2}\right) + 2\frac{\alpha a v_0}{r}.$$

Les lignes de courant (confondues avec les trajectoires, car l'écoulement est stationnaire) sont parallèles à la vitesse. On cherche une expression paramétrée des lignes de champs ; il faut alors résoudre le système différentiel :

$$\frac{dr}{dt} = v_r \quad \text{et} \quad \frac{d\theta}{dt} = \frac{v_\theta}{r}.$$

On utilise des coordonnées non dimensionnées, c'est-à-dire rapportées à a, rayon du cylindre, ce qui revient à supposer $a = 1$. Tracer les lignes de courant de cet écoulement pour $\alpha \in \{0.1; 0.5; 1; 1.25\}$.

L'utilisation de l'effet Magnus a été proposé pour mettre au point des systèmes de propulsion composés de gros cylindres verticaux en rotation capables de produire une poussée longitudinale lorsque le vent est sur le côté du navire (ce fut le cas du navire Baden-Baden mis au point par Anton Flettner qui traversa l'Atlantique en 1926).

4.1.7 Développée d'une courbe

On donne à présent un exemple de tracé de développée d'un arc paramétré (on rappelle que la développée est l'enveloppe des normales ou, de manière équivalente, le lieu des centres de courbure de la courbe).

Exemple. (*Développée de la parabole*) Trouvons l'équation de la développée de la parabole $\mathcal{P}$ d'équation $y = x^2/4$ et traçons sur un même graphique la parabole $\mathcal{P}$, quelques normales à $\mathcal{P}$ et sa développée.

Pour déterminer un système d'équations paramétriques de l'enveloppe d'une famille de droites Δ_t définies par des équations cartésiennes de la

forme $\alpha(t)x + \beta(t)y = \gamma(t)$, on exprime le fait que la droite Δ_t est tangente à l'enveloppe en $(x(t), y(t))$:

$$\begin{cases} \alpha(t)x + \beta(t)y = \gamma(t), & (1) \\ \alpha(t)x' + \beta(t)y' = 0. & (2) \end{cases}$$

On dérive l'équation (1) et en combinant avec (2), on obtient le système :

$$\begin{cases} \alpha(t)x + \beta(t)y = \gamma(t), & (1) \\ \alpha'(t)x + \beta'(t)y = \gamma'(t). & (3) \end{cases}$$

Dans notre cas, la normale (N_t) à la parabole $\mathcal{P}$ en $M(t, t^2/4)$ a pour vecteur normal $\overrightarrow{v} = (1, t/2)$ (qui est le vecteur tangent à la parabole) ; elle a donc pour équation :

$$\begin{pmatrix} x - t \\ y - t^2/4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ t/2 \end{pmatrix} = 0 \iff x + \frac{t}{2}y = t + \frac{t^3}{8}$$

autrement dit, $(\alpha(t), \beta(t), \gamma(t)) = (1, t/2, t + t^3/8)$. On peut alors résoudre le système précédent avec la fonction `solve` :

```
x, y, t = var('x, y, t')
alpha(t) = 1; beta(t) = t / 2; gamma(t) = t + t^3 / 8
env = solve([alpha(t) * x + beta(t) * y == gamma(t),\
    diff(alpha(t), t) * x + diff(beta(t), t) * y == \
    diff(gamma(t), t)], [x,y])
```

$$\left[\left[x = -\frac{1}{4}\,t^3, y = \frac{3}{4}\,t^2 + 2\right]\right]$$

D'où une représentation paramétrique de l'enveloppe des normales :

$$\begin{cases} x(t) = -\frac{t^3}{4}, \\ y(t) = 2 + \frac{3}{4}t^2. \end{cases}$$

On peut alors effectuer le tracé demandé, en traçant quelques normales à la parabole (plus précisément, on trace des segments $[M, M + 18\overrightarrow{n}]$ où $M(u, u^2/4)$ est un point de $\mathcal{P}$ et $\overrightarrow{n} = (-u/2, 1)$ un vecteur normal à $\mathcal{P}$) :

```
f(x) = x^2 / 4
p = plot(f, -8, 8, rgbcolor=(0.2,0.2,0.4)) # trace la parabole
for u in srange(0, 8, 0.1): # trace des normales à la parabole
    p += line([[u, f(u)], [-8*u, f(u) + 18]], thickness=.3)
    p += line([[-u, f(u)], [8*u, f(u) + 18]], thickness=.3)
p += parametric_plot((env[0][0].rhs(),env[0][1].rhs()),
    (t, -8, 8),color='red')              # trace la développée
p.show(xmin=-8, xmax=8, ymin=-1, ymax=12, aspect_ratio=1)
```

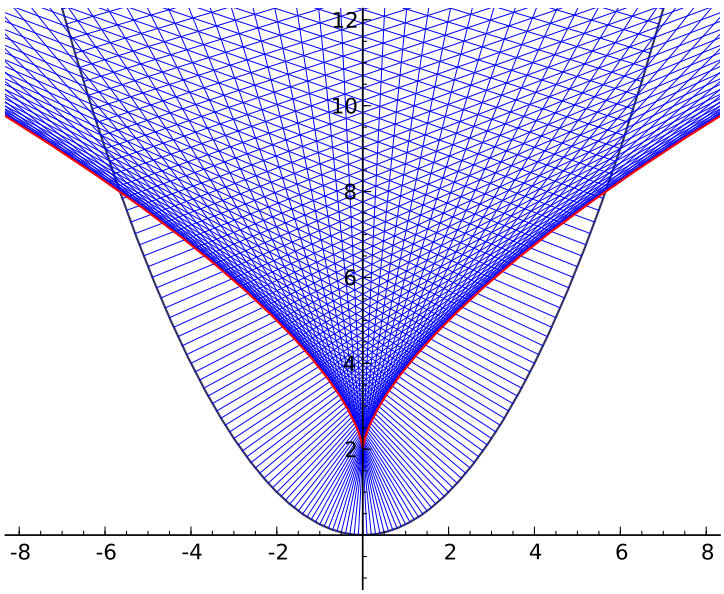


FIG. 4.13 – La développée de la parabole.

Comme nous l'avons rappelé initialement, la développée d'une courbe est aussi le lieu de ses centres de courbures. À l'aide de la fonction `circle` traçons quelques cercles osculateurs à la parabole. On rappelle que le centre de courbure Ω en un point $M_t = (x(t), y(t))$ de la courbe, a pour coordonnées :

$$x_\Omega = x + -y' \frac{x'^2 + y'^2}{x'y'' - x''y'}, \qquad \text{et} \qquad y_\Omega = y + x' \frac{x'^2 + y'^2}{x'y'' - x''y'}.$$

et que le rayon de courbure en M_t vaut :

$$R = \frac{(x'^2 + y'^2)^{\frac{3}{2}}}{x'y'' - x''y'}.$$

```
var('t'); p = 2
x(t) = t; y(t) = t^2 / (2 * p)
f(t) = [x(t), y(t)]
df(t) = [x(t).diff(t), y(t).diff(t)]
d2f(t) = [x(t).diff(t, 2), y(t).diff(t, 2)]
T(t) = [df(t)[0] / df(t).norm(), df[1](t) / df(t).norm()]
N(t) = [-df(t)[1] / df(t).norm(), df[0](t) / df(t).norm()]
R(t) =  (df(t).norm())^3 / \
        (df(t)[0] * d2f(t)[1] -df(t)[1] * d2f(t)[0])
Omega(t) = [f(t)[0] + R(t)*N(t)[0], f(t)[1] + R(t)*N(t)[1]]
g = parametric_plot(f(t), t, -8, 8, color='green', thickness=2)
for u in srange(.4, 4, .2):
```

```
    g += line([f(t = u), Omega(t = u)], color='red', alpha = .5)
    g += circle(Omega(t = u), R(t = u), color='blue')
g.show(aspect_ratio=1, xmin=-12, xmax=7, ymin=-3, ymax=12)
```

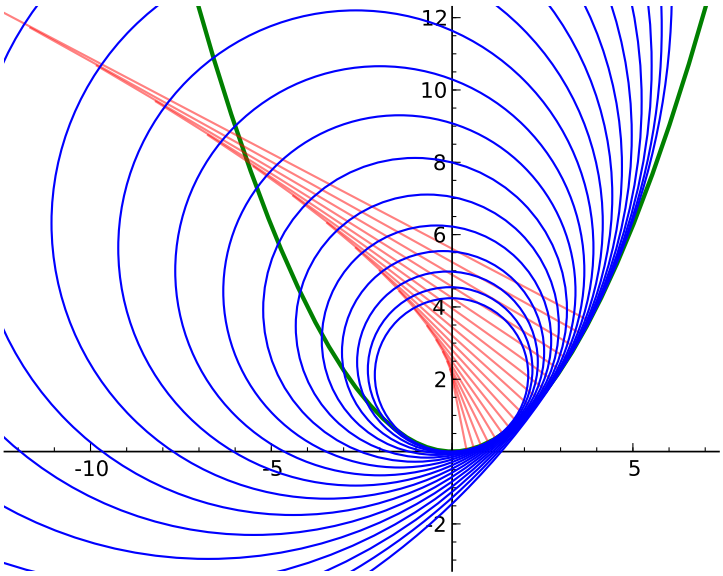


FIG. 4.14 – Cercles osculateurs à la parabole.

Le tableau 4.1 résume les fonctions utilisées dans cette section. On y signale de plus la commande `text` qui permet de placer une chaîne de caractère dans un graphique, ainsi que la commande `polygon` qui permet de tracer des polygones.

4.2 Courbes en 3D

Pour le tracé de surfaces en 3 dimensions, on dispose de la commande `plot3d(f(x,y),(x,a,b),(y,c,d))`. La surface obtenue est alors visualisée par défaut grâce à l'application *Jmol* ; cependant, on peut également utiliser le *Tachyon 3D Ray Tracer* grâce à l'argument optionnel suivant `g.show(viewer='tachyon')`. Voici un premier exemple de tracé d'une surface paramétrée (Fig. 4.15) :

```
sage: u, v = var('u, v')
sage: h = lambda u,v: u^2 + 2*v^2
sage: f = plot3d(h, (u,-1,1), (v,-1,1), aspect_ratio=[1,1,1])
```

La visualisation de la surface représentative d'une fonction de deux variables permet de guider l'étude d'une telle fonction, comme nous allons le voir dans l'exemple suivant.

Type de tracé	Fonction
Courbe représentative d'une fonction	`plot`
Courbe paramétrée	`parametric_plot`
Courbe définie par une équation polaire	`polar_plot`
Courbe définie par une équation implicite	`implicit_plot`
Lignes de niveau d'une fonction complexe	`complex_plot`
Objet graphique vide	`Graphics()`
Courbes intégrales d'équation différentielle	`odeint, desolve_rk4`
Diagramme en bâtonnets	`bar_chart`
Diagramme des fréquence d'une série statistique	`plot_histogram`
Tracé d'une ligne brisée	`line`
Tracé d'un nuage de points	`points`
Cercle	`circle`
Polygone	`polygon`
Texte	`text`

TABLEAU 4.1 – Récapitulatif des fonctions graphiques.

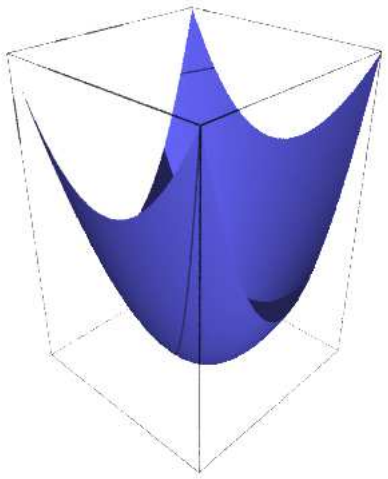


FIG. 4.15 – La surface paramétrée par $(u, v) \mapsto u^2 + 2v^2$.

Exemple. (*Une fonction discontinue dont les dérivées directionnelles existent partout !*) Étudier l'existence en $(0, 0)$ des dérivées directionnelles et la conti-

nuité de l'application f de $\mathbb{R}^2$ dans $\mathbb{R}$ définie par :

$$f(x,y) = \begin{cases} \frac{x^2 y}{x^4+y^2} & \text{si } (x,y) \neq (0,0), \\ 0 & \text{si } (x,y) = (0,0). \end{cases}$$

Pour $H = \binom{\cos\theta}{\sin\theta}$, l'application $\varphi(t) = f(tH) = f(t\cos\theta, t\sin\theta)$ est dérivable en $t = 0$ pour toute valeur de θ ; en effet,

```
sage: f(x, y) = x^2 * y / (x^4 + y^2)
sage: var('t theta')
sage: limit(f(t * cos(theta), t * sin(theta)) / t, t=0)
cos(theta)^2/sin(theta)
```

Donc f admet en $(0,0)$ des dérivées directionnelles suivant n'importe quel vecteur. Pour mieux se représenter la surface représentative de f, on peut commencer par chercher quelques lignes de niveau ; par exemple la ligne de niveau associée à la valeur $\frac{1}{2}$:

```
sage: solve(f(x,y) == 1/2, y)
[y == x^2]
sage: a = var('a'); h = f(x, a*x^2).simplify_rational(); h
a/(a^2 + 1)
```

Le long de la parabole d'équation $y = ax^2$ privée de l'origine, f est constante à $f(x, ax^2) = \frac{a}{1+a^2}$. On trace alors la fonction $h\colon a \longmapsto \frac{a}{1+a^2}$:

```
sage: plot(h, a, -4, 4)
```

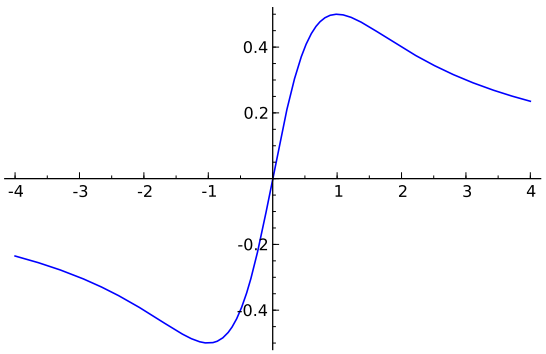


FIG. 4.16 – Une coupe verticale de la surface étudiée.

La fonction h atteint son maximum en $a = 1$ et son minimum en $a = -1$. La restriction de f à la parabole d'équation $y = x^2$ correspond à la « ligne de crête » qui se trouve à l'altitude $\frac{1}{2}$; quant à la restriction de f à la parabole d'équation $y = -x^2$, elle correspond au fond du « thalweg » qui se trouve à l'altitude $-\frac{1}{2}$. En conclusion, aussi proche que l'on soit de $(0,0)$, on peut trouver des points en lesquels f prend respectivement les valeurs $\frac{1}{2}$ et $-\frac{1}{2}$. Par conséquent, la fonction n'est pas continue à l'origine.

```
p = plot3d(f(x, y), (x,-2,2), (y,-2,2), plot_points=[150,150])
```

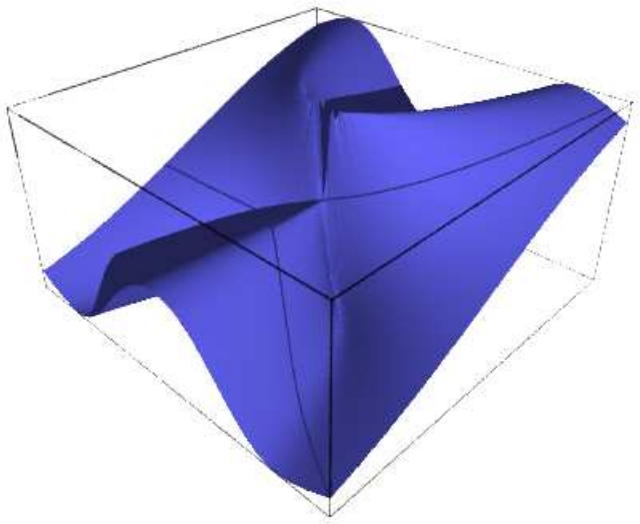


FIG. 4.17 – La surface représentative de $f\colon (x,y) \mapsto \frac{x^2 y}{x^4+y^2}$

On aurait également pu tracer des plans horizontaux pour visualiser les lignes de niveau de cette fonction en exécutant :

```
for i in range(1,4):
    p += plot3d(-0.5 + i / 4, (x, -2, 2), (y, -2, 2),\
                color=hue(i / 10), opacity=.1)
```

Parmi les autres commandes de tracé 3D, citons `implicit_plot3d` qui permet de tracer des surfaces définies par une équation implicite de la forme $f(x, y, z) = 0$. Traçons par exemple la surface de Cassini définie par l'équation implicite :

$$\left(a^2 + x^2 + y^2\right)^2 = 4\,a^2x^2 + z^4.$$

```
var('x, y, z'); a = 1
h = lambda x, y, z:(a^2 + x^2 + y^2)^2 - 4*a^2*x^2-z^4
f = implicit_plot3d(h, (x, -3, 3), (y, -3, 3), (z, -2, 2),\
                    plot_points=100, adaptative=True)
```

Enfin donnons un exemple de tracé de courbe dans l'espace à l'aide de la commande `line3d` :

```
g1 = line3d([(-10*cos(t)-2*cos(5*t)+15*sin(2*t),\
     -15*cos(2*t)+10*sin(t)-2*sin(5*t),\
      10*cos(3*t)) for t in srange(0,6.4,.1)],radius=.5)
```

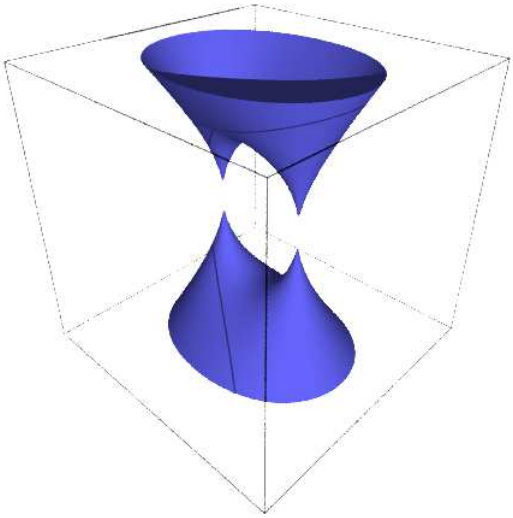


FIG. 4.18 – La surface de Cassini

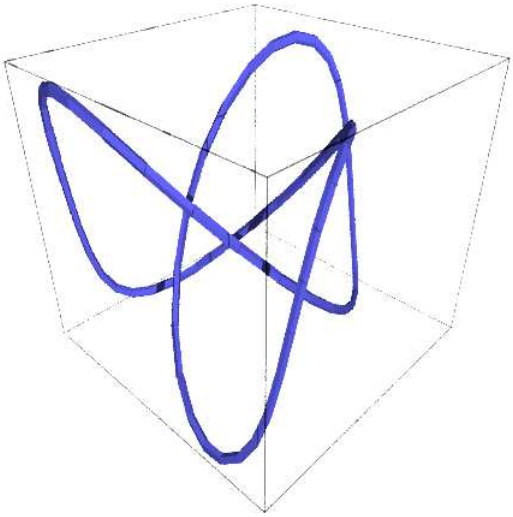


FIG. 4.19 – Un nœud dans l'espace

Deuxième partie

Calcul numérique

5

Algèbre linéaire numérique

On traite ici les aspects numériques de l'algèbre linéaire, l'algèbre linéaire symbolique étant présentée au chapitre 10. Pour des textes en français sur les méthodes et l'analyse numérique de l'algèbre linéaire, on pourra consulter les livres de M. Schatzman [Sch91], de P. Ciarlet [Cia82] ou ceux plus spécialisés de Lascaux-Théodor [LT93, LT94]. Le livre de Golub et Van Loan [GVL96], en anglais, est une véritable bible.

L'algèbre linéaire numérique joue un rôle prépondérant dans ce qu'il est convenu d'appeler le *calcul scientifique*, appellation impropre pour désigner des problèmes dont l'étude mathématique relève de l'analyse numérique : résolution approchée de systèmes d'équations différentielles, résolution approchée d'équations aux dérivées partielles, optimisation, traitement du signal, etc.

La résolution numérique de la plupart de ces problèmes, même linéaires, est fondée sur des algorithmes formés de boucles imbriquées ; au plus profond de ces boucles, il y a très souvent la résolution d'un système linéaire. On utilise souvent la méthode de Newton pour résoudre des systèmes algébriques non linéaires : là encore il faut résoudre des systèmes linéaires. La performance et la robustesse des méthodes d'algèbre linéaire numérique sont donc stratégiques.

Ce chapitre comporte trois sections : dans la première, on s'efforce de sensibiliser le lecteur à l'influence de l'inexactitude des calculs en algèbre linéaire ; la deuxième section (§5.2) traite, sans être exhaustive, des problèmes les plus classiques (résolution de systèmes, calcul de valeurs propres, moindres carrés) ; dans la troisième section (§5.3) on montre comment résoudre certains problèmes si on fait l'hypothèse que les matrices sont creuses. Cette dernière partie se veut plus une initiation à des méthodes qui font partie d'un domaine de recherche actif qu'un guide d'utilisation.

5.1 Calculs inexacts en algèbre linéaire

On s'intéresse aux problèmes classiques de l'algèbre linéaire (résolution de systèmes, calcul de valeurs et de vecteurs propres, etc.), résolus au moyen de calculs inexacts. La première source d'inexactitude vient de ce qu'on va utiliser une approximation flottante des nombres (réels ou complexes), et donc non seulement travailler sur des objets connus approximativement, mais aussi entacher tous les calculs d'erreurs. Les différents types de nombres flottants utilisables dans Sage sont décrits au chapitre 7.

Soit par exemple à résoudre le système $Ax = b$ où A est une matrice à coefficients réels. Quelle erreur δx commet-on si on perturbe A de δA ou b de δb ? On apporte quelques éléments de réponse dans ce chapitre.

Normes de matrices et conditionnement

Soit $A \in \mathbb{R}^{n\times n}$ (ou $\mathbb{C}^{n\times n}$). On équipe $\mathbb{R}^n$ (ou $\mathbb{C}^n$) d'une norme $\|x\|$, par exemple la norme $\|x\|_\infty = \max |x_i|$ ou $\|x\|_1 = \sum_{i=1}^n |x_i|$, ou encore la norme euclidienne $\|x\|_2 = (\sum_{i=1}^n x_i^2)^{1/2}$; alors la quantité

$$\|A\| = \max_{\|x\|=1} \frac{\|A\ x\|}{\|x\|}$$

définit une norme sur l'ensemble des matrices $n \times n$. On dit qu'il s'agit d'une norme *subordonnée* à la norme définie sur $\mathbb{R}^n$ (ou $\mathbb{C}^n$). Le *conditionnement* de A est défini par $\kappa(A) = \|A^{-1}\| \cdot \|A\|$. Le résultat fondamental est que, si on fait subir à A une (petite) perturbation δA et à b une perturbation δb, alors la solution x du système linéaire $Ax = b$ est perturbée de $\|\delta x\|$ qui vérifie :

$$\frac{\|\delta x\|}{\|x\|} \leqslant \frac{\kappa(A)}{1 - \kappa(A)\|\delta A\|/\|A\|} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right).$$

Les normes $\|\cdot\|_\infty$ et $\|\cdot\|_1$ sont faciles à calculer : $\|A\|_\infty = \max_{1\leqslant i\leqslant n}(\sum_{j=1}^n |A_{ij}|)$ et $\|A\|_1 = \max_{1\leqslant j\leqslant n}(\sum_{i=1}^n |A_{ij}|)$. En revanche la norme $\|\cdot\|_2$ ne s'obtient pas simplement car $\|A\|_2 = \sqrt{\rho({}^t A\, A)}$, le rayon spectral ρ d'une matrice A étant le maximum des modules de ses valeurs propres.

La norme de Frobenius est définie par

$$\|A\|_F = \left(\sum_{i=1}^n \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2}.$$

À la différence des normes précédentes, ce n'est pas une norme subordonnée. On vérifie aisément que $\|A\|_F^2 = \mathrm{trace}({}^t A\, A)$.

Le calcul des normes de matrice dans `Sage`. Les matrices possèdent une méthode `norm(p)`. Selon la valeur de l'argument p on obtiendra :

$p = 1$:	$\|A\|_1$,	$p = 2$:	$\|A\|_2$,
$p =$ `Infinity` :	$\|A\|_\infty$,	$p =$ `'frob'` :	$\|A\|_F$.

Cette méthode n'est applicable que si les coefficients de la matrice peuvent être convertis en nombres complexes `CDF`. Noter qu'on écrit `A.norm(Infinity)` mais `A.norm('frob')`. Avec `A.norm()` on obtiendra la norme $\|A\|_2$ (valeur par défaut).

Erreurs et conditionnement : une illustration, avec des calculs exacts, puis approchés

Utilisons la possibilité de `Sage` de faire des calculs exacts quand les coefficients sont rationnels. On considère la matrice de Hilbert

$$A_{ij} = 1/(i + j - 1),\ i, j = 1, \ldots, n.$$

Le programme suivant calcule exactement le conditionnement des matrices de Hilbert, en norme $\|\cdot\|_\infty$:

```
sage: def cond_hilbert(n):
....:     A = matrix(QQ, [[1/(i+j+1) for j in range(n)]
....:                                for i in range(n)])
....:     return A.norm(Infinity) * (A^-1).norm(Infinity)
```

Voici les résultats, en fonction de `n` :

`n`	conditionnement
2	27
4	28375
8	33872791095
16	5.06277478751e+22
32	1.35710782493e+47

On remarque l'accroissement extrêmement rapide du conditionnement en fonction de n. On peut montrer que $\kappa(A) \simeq e^{7n/2}$, ce qui évidemment, est une quantité qui croît très vite. Toujours en calculant exactement dans l'ensemble des rationnels, on peut perturber la matrice A, et comparer la solution d'un système linéaire original à celle du même système perturbé :

```
sage: x = vector(QQ,[1 for i in range(0,n)])
sage: y = A*x
sage: for k in range(0,n):
sage:     A[k,k]=(1/(2*k+1))*(1+1/(10^5)) #perturber la matrice
sage: sol = A\y
sage: diff = max(float(sol[i]-x[i]) for i in range(0,n))
```

On obtient :

`n`	erreur (`diff`)
2	1.9999200e-05
4	0.00597609561
8	3.47053530779
16	63.2816091951
32	20034.3477421

et donc les calculs sont rapidement entachés d'une erreur inacceptable.

Calculons maintenant avec des matrices et des vecteurs à coefficients flottants. Cette fois, on ne perturbe pas explicitement la matrice A, mais l'arithmétique flottante va introduire de petites perturbations. On refait le calcul précédent : le second membre étant calculé par $y = Ax$, on cherche à retrouver x en résolvant le système linéaire $As = y$:

```
sage: n = 8
sage: A = matrix(RR, [[1/(i+j+1) for j in range(n)]
....:                           for i in range(n)])
sage: x = vector(RR,[1 for i in range(0,n)])
sage: y = A*x
#comparer la solution exacte x a celle du systeme perturbe s:
sage: s = A.solve_right(y)
sage: diff = [float(s[i]-x[i]) for i in range(0,n)]
```

On obtient :

`n`	erreur (`diff`)
2	2.22044604925e-16
4	3.05977465587e-13
8	6.82028985288e-07
16	8.34139063331
32	257.663242705

On voit que pour $n = 16$ par exemple, l'erreur sur la solution (en norme infinie) est telle qu'on ne calcule plus rien du tout de pertinent (avec le choix particulier de x que nous avons fait ($x = 1$), les erreurs absolues et relatives en norme infinie coïncident).

Remarques

Pourquoi donc alors calculer avec des nombres flottants ? La question des performances n'est pas forcément pertinente depuis qu'il existe des bibliothèques implémentant efficacement des algorithmes d'algèbre linéaire en arithmétique rationnelle (Linbox, utilisée par Sage), algorithmes qui, sans être aussi performants que leur équivalent flottant, pourraient avantageusement être utilisés, pour certains problèmes comme la résolution de systèmes

linéaires de taille modérée. Mais c'est ici qu'intervient une deuxième source d'incertitudes : dans les applications réelles, les coefficients ne peuvent être connus qu'approximativement : par exemple la résolution d'un système d'équations non linéaires par la méthode de Newton fait naturellement apparaître des termes calculés de manière inexacte.

Les systèmes linéaires mal conditionnés (sans être aussi caricaturaux que la matrice de Hilbert) sont plutôt la règle que l'exception : il est fréquent (en physique, en chimie, en biologie, etc.) de rencontrer des systèmes d'équations différentielles ordinaires de la forme $du/dt = F(u)$ où la matrice jacobienne $DF(u)$, matrice des dérivées partielles $\partial F_i(u)/\partial u_j$, définit un système linéaire mal conditionné : les valeurs propres sont réparties dans un ensemble très vaste, ce qui entraîne le mauvais conditionnement de $DF(u)$; cette propriété traduit le fait que le système modélise des phénomènes mélangeant plusieurs échelles de temps. Malheureusement, en pratique, il faut résoudre des systèmes linéaires dont la matrice est $DF(u)$.

Tous les calculs (décomposition de matrices, calcul d'éléments propres, convergence de méthodes itératives) sont affectés par le conditionnement. Il convient donc d'avoir cette notion présente à l'esprit dès que l'on calcule en utilisant une représentation flottante des nombres réels.

5.2 Matrices pleines

5.2.1 Résolution de systèmes linéaires

Ce qu'il ne faut (presque) jamais faire : utiliser les formules de Cramer. Un raisonnement par récurrence montre que le coût du calcul du déterminant d'une matrice $n \times n$ est de l'ordre $n!$ multiplications (et autant d'additions). Pour résoudre un système de taille n, ce sont $n + 1$ déterminants qu'il faut calculer. Prenons $n = 20$:

```
sage: n = 20
sage: cout = (n+1)*factorial(n)
```

nous obtenons la valeur respectable de `51090942171709440000` multiplications. Supposons que notre calculateur effectue 3 10^9 multiplications par seconde (ce qui est réaliste), et calculons la durée approximative du calcul :

```
sage: v = 3*10^9
sage: float(cout/v/3600/24/365)
 540.02771617315068
```

Il faudra donc **`540 ans`** (environ) pour effectuer le calcul ! Bien sûr vous pouvez utiliser les formules de Cramer pour résoudre un système 2×2, mais pas bien au delà ! Toutes les méthodes utilisées en pratique ont en commun un coût polynomial, c'est à dire de l'ordre de n^p, avec p petit ($p = 3$, en général).

Méthodes pratiques. La résolution de systèmes linéaires $Ax = b$ est le plus souvent basée sur une factorisation de la matrice A en un produit de deux matrices $A = M_1 M_2$, M_1 et M_2 définissant des systèmes linéaires faciles à résoudre. Pour résoudre $Ax = b$, on résout alors successivement $M_1 y = b$, puis $M_2 x = y$.

Par exemple M_1 et M_2 peuvent être deux matrices triangulaires ; dans ce cas, une fois la factorisation effectuée, il faut résoudre deux systèmes linéaires à matrice triangulaire. Le coût de la factorisation est bien plus élevé que celui de la résolution des deux systèmes triangulaires (par exemple $O(n^3)$ pour la factorisation LU contre $O(n^2)$ pour la résolution des systèmes triangulaires). Il convient donc, dans le cas de la résolution de plusieurs systèmes avec la même matrice, de ne calculer qu'une seule fois la décomposition. Bien entendu, on n'inverse *jamais* une matrice pour résoudre un système linéaire, l'inversion demandant la factorisation de la matrice, puis la résolution de n systèmes au lieu d'un seul.

5.2.2 Résolution directe

Voici la manière la plus simple de procéder :

```
sage: A = Matrix(RDF, [[-1,2],[3,4]])
sage: b =  vector(RDF,[2,3])
sage: x = A\b
sage: print x
 (-0.2, 0.9)
```

Dans Sage, les matrices possèdent une méthode `solve_right` pour la résolution de systèmes linéaires (basée sur la décomposition LU) ; c'est cette commande qui est appelée ci-dessus. On aurait aussi pu écrire :

```
sage: x = A.solve_right(b)
```

La syntaxe `x = A\b` de la première version est pratiquement identique à ce qu'on utilise dans les systèmes de calcul Matlab, Octave ou Scilab. La méthode utilise la décomposition LU.

5.2.3 La décomposition LU

```
sage: A = Matrix(RDF, [[-1,2],[3,4]])
sage: P, L, U = A.LU()
```

Cette méthode fournit les facteurs L et U ainsi que la matrice de permutation P, tels que $A = PLU$. Ce sont les choix de pivots qui imposent la création de la matrice P. La matrice L est triangulaire inférieure, à diagonale unité, et U est une matrice triangulaire supérieure. Cette décomposition est directement dérivée de la méthode de Gauss (décrite en §10.2.1). En effet, en négligeant pour la simplicité de l'exposé les problèmes de choix de pivots,

la méthode de Gauss consiste à transformer la matrice A (d'ordre n) pour faire apparaître des coefficients nuls sous la diagonale de la première colonne, puis de la deuxième colonne et ainsi de suite. Cette élimination peut s'écrire

$$L_{n-1} \dots L_2 L_1 A = U.$$

En posant $L = (L_{n-1} \dots L_2 L_1)^{-1}$, on obtient bien $A = LU$ et on vérifie sans difficultés que L est triangulaire inférieure à diagonale unité.

Notons que Sage garde en mémoire la factorisation de A : la commande `A.LU_valid()` va répondre `True` si et seulement si la factorisation LU a déjà été calculée. Mieux, la commande `a.solve_right(b)` ne calculera la factorisation que si c'est nécessaire, c'est-à-dire si elle n'a pas été calculée auparavant, ou si la matrice `A` a changé.

Exemple. Créons une matrice de taille 1000, aléatoire et un vecteur de taille 1000 :

```
sage: A = random_matrix(RDF,1000)
sage: b = vector(RDF,[i for i in range(0,1000)])
```

factorisons `A` :

```
sage: %time A.LU()
 CPU times: user 0.23 s, sys: 0.01 s, total: 0.24 s
 Wall time: 0.29 s
```

et à présent résolvons le système $Ax = b$:

```
sage: %time x = A.solve_right(b)
 CPU times: user 0.10 s, sys: 0.00 s, total: 0.10 s
 Wall time: 0.12 s
```

La résolution est plus rapide, parce qu'elle a tenu compte de la factorisation précédemment calculée.

5.2.4 La décomposition de Cholesky des matrices réelles symétriques définies positives

Une matrice symétrique A est dite définie positive si pour tout vecteur x non nul, ${}^t x\, Ax > 0$. Pour toute matrice symétrique définie positive, il existe une matrice triangulaire inférieure C telle que $A = C\, {}^t C$. Cette factorisation est appelée décomposition de Cholesky. Dans Sage, elle se calcule en appelant la méthode `cholesky_decomposition()`. Exemple :

```
 #fabriquer une matrice A symétrique presque sûrement définie
 #positive
sage: m = random_matrix(RDF,10)
sage: A = transpose(m)*m
sage: C = A.cholesky_decomposition()
```

Il faut noter qu'on ne peut pas tester en un coût raisonnable si une matrice symétrique est définie positive ; si on applique la méthode de Cholesky avec une matrice qui ne l'est pas, la décomposition ne pourra pas être calculée et une exception `ValueError` sera lancée par `cholesky_decomposition()` au cours du calcul.

Pour résoudre un système $Ax = b$ à l'aide de la décomposition de Cholesky, on procède comme avec la décomposition LU. Une fois la décomposition calculée, on exécute `A.solve_right(b)`. Là aussi, la décomposition n'est pas recalculée.

Pourquoi utiliser la décomposition de Cholesky plutôt que la décomposition LU pour résoudre des systèmes à matrice symétrique définie positive ? bien sûr, la taille mémoire nécessaire pour stocker les facteurs est deux fois plus petite, ce qui n'est pas très important, mais c'est surtout du point de vue du compte d'opérations que la méthode de Cholesky s'avère avantageuse : en effet, pour une matrice de taille n, la factorisation de Cholesky coûte n extractions de racines carrées, $n(n-1)/2$ divisions, $(n^3-n)/6$ additions et autant de multiplications. En comparaison la factorisation LU coûte aussi $n(n-1)/2$ divisions, mais $(n^3-n)/3$ additions et multiplications.

5.2.5 La décomposition QR

Soit $A \in \mathbb{R}^{n\times m}$, avec $n \geqslant m$. Il s'agit ici de trouver deux matrices Q et R telles que $A = QR$ où $Q \in \mathbb{R}^{n\times n}$ est orthogonale (${}^tQ \cdot Q = I$) et $R \in \mathbb{R}^{n\times m}$ est triangulaire supérieure. Bien sûr, une fois la décomposition calculée, on peut s'en servir pour résoudre des systèmes linéaires si la matrice A est carrée et inversible, mais c'est surtout, comme on le verra, une décomposition intéressante pour la résolution de systèmes aux moindres carrés et pour le calcul de valeurs propres. Bien noter que A n'est pas forcément carrée. La décomposition existe si A est de rang maximum m. Exemple :

```
sage: A = random_matrix(RDF,6,5)
sage: Q,R = A.QR()
```

5.2.6 La décomposition en valeurs singulières

Il s'agit d'une décomposition peu enseignée, et pourtant riche en applications ! Soit A une matrice $n \times m$ à coefficients réels. Alors il existe deux matrices orthogonales $U \in \mathbb{R}^{n\times n}$ et $V \in \mathbb{R}^{m\times m}$, telles que

$$ {}^tU \cdot A \cdot V = \Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_p)$$

où $\sigma_1 \geqslant \sigma_2 \geqslant \ldots \geqslant \sigma_p \geqslant 0$ ($p \leqslant \min(m,n)$).

Les matrices U et V sont orthogonales : $U \cdot {}^tU = I$ et $V \cdot {}^tV = I$ et par conséquent :

$$A = U\Sigma\, {}^tV\,.$$

Exemple :

```
sage: A = matrix(RDF,[[1,3,2],[1,2,3],[0,5,2],[1, 1, 1]])
sage: U,Sig,V = A.SVD()
sage: A1=A-U*Sig*transpose(V)
sage: print(A1) # vérification
[  4.4408920985e-16   4.4408920985e-16   -8.881784197e-16]
[ 6.66133814775e-16   -8.881784197e-16  -4.4408920985e-16]
[-1.29063426613e-15   1.7763568394e-15  2.22044604925e-16]
[ 6.66133814775e-16 -6.66133814775e-16 -1.11022302463e-15]
```

(les calculs ne sont évidemment jamais exacts !)

On peut montrer que les valeurs singulières d'une matrice A sont les racines carrées des valeurs propres de ${}^tA\,A$. Il est facile de vérifier que, pour une matrice carrée de taille n, la norme euclidienne $\|A\|_2$ est égale à σ_1 et que, si la matrice est de plus non singulière, le conditionnement de A en norme euclidienne est égal à σ_1/σ_n.

5.2.7 Application aux moindres carrés

On voudrait résoudre le système surdéterminé $Ax = b$ où A est une matrice à coefficients réels, rectangulaire, à n lignes et m colonnes avec $n > m$. Évidemment ce système n'a pas, en général, de solution. On considère alors le problème de minimisation du carré de la norme euclidienne $\|\cdot\|_2$ du résidu :

$$\min_x \|Ax - b\|_2^2.$$

La matrice A peut même être de rang inférieur à m.

En résolvant les équations normales

En annulant la différentielle par rapport à x du problème de minimisation, on vérifiera sans trop de peine que la solution vérifie :

$${}^tA\,Ax = {}^tA\,b.$$

Supposant A de rang maximum m, on peut donc penser former la matrice ${}^tA\,A$ et résoudre le système ${}^tA\,Ax = {}^tA\,b$, par exemple en calculant la décomposition de Cholesky de ${}^tA\,A$. C'est même là l'origine du *procédé du commandant Cholesky*[1]. Quel est le conditionnement de ${}^tA\,A$? C'est ce qui va conditionner la précision des calculs. Les valeurs singulières de ${}^tA\,A$ sont les carrés des valeurs singulières de A ; le conditionnement en norme euclidienne est donc σ_1^2/σ_n^2, qui est facilement grand (pour une matrice carrée le conditionnement euclidien de ${}^tA\,A$ est égal au carré du conditionnement

[1] Polytechnicien et officier d'artillerie (1875-1918) ; la méthode a été inventée pour résoudre des problèmes de géodésie.

de A). On préfère donc des méthodes basées soit sur la décomposition QR de A, soit sur sa décomposition en valeurs singulières[2].

Malgré tout, cette méthode est utilisable pour de petits systèmes, pas trop mal conditionnés. Voici le code correspondant :

```
sage: A = matrix(RDF,[[1,3,2],[1,4,2],[0,5,2],[1,3,2]])
sage: B = vector(RDF,[1,2,3,4])
sage: Z = transpose(A)*A
sage: C = Z.cholesky_decomposition()
sage: R = transpose(A)*B
sage: Z.solve_right(R)
```

On obtient :

```
(-1.5, -0.5, 2.75)
```

Bien noter ici que la décomposition de Cholesky est *cachée* et que la résolution `Z.solve_right(C)` utilise cette décomposition, sans la recalculer.

Avec la décomposition QR

Supposons A de rang maximum[3], et soit $A = QR$. Alors

$$\|Ax - b\|_2^2 = \|QRx - b\|_2^2 = \|Rx - {}^tQ\,b\|_2^2.$$

On a : $R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$ où R_1 est un bloc triangulaire supérieur de taille m et ${}^tQ\,b = \begin{bmatrix} c \\ d \end{bmatrix}$ (c de taille m). Donc $\|Ax - b\|_2^2 = \|R_1x - c\|_2^2 + \|d\|_2^2$, et le minimum est obtenu pour x solution du système triangulaire $R_1x = c$:

```
sage: A = matrix(RDF,[[1,3,2],[1,4,2],[0,5,2],[1,3,2]])
sage: B = vector(RDF,[1,2,3,4])
sage: Q,R = A.QR()
sage: R1=R[0:3,0:3]
sage: B1=transpose(Q)*B
sage: C = B1[0:3]
sage: R1.solve_right(C)
```

On obtient là aussi :

```
(-1.5, -0.5, 2.75).
```

Calculons le conditionnement de ${}^tA\,A$ en norme infinie :

[2]Mais le commandant Cholesky n'avait pas d'ordinateur, et n'envisageait probablement que la résolution de petits systèmes, pour lesquels le mauvais conditionnement n'est pas vraiment un problème.

[3] On peut s'en affranchir en utilisant une méthode QR avec pivots.

```
sage: Z = A.transpose()*A
sage: Z.norm(Infinity)*(Z^-1).norm(Infinity)
1992.375
```

Le système étant de petite taille, n'est pas trop mal conditionné : la méthode QR et la méthode des équations normales (page 106) donnent donc (à peu près) le même résultat.

Avec la décomposition en valeurs singulières

La décomposition en valeurs singulières $A = U\Sigma\,{}^tV$ permet aussi de calculer la solution ; mieux elle est utilisable même si A n'est pas de rang maximum. Rappelons d'abord que Σ possède $p \leqslant m$ coefficients non nuls σ_i, positifs (et rangés par ordre décroissant). On note u_i et v_i les colonnes de U et V. On a alors :

$$\|Ax - b\|_2^2 = \|{}^tU\,AV\,{}^tV\,x - {}^tU\,b\|_2^2.$$

En posant $v = {}^tV\,x$, on a :

$$\|Ax - b\|_2^2 = \sum_{i=1}^{p}(\sigma_i v_i - {}^tu_i\,b)^2 + \sum_{i=p+1}^{m} {}^tu_i\,b.$$

Le minimum est donc atteint pour $v_i = ({}^tu_i\,b)/\sigma_i$, et donc :

$$x = \sum_{i=1}^{p} \frac{{}^tu_i\,b}{\sigma_i} v_i.$$

Voici le programme Sage :

```
A = matrix(RDF,[[1,3,2],[1,3,2],[0,5,2],[1,3,2]])
B = vector(RDF,[1,2,3,4])

U,Sig,V = A.SVD()

n = A.ncols()
X = vector(RDF,[0]*n)

for i in range(0,n):
    s = Sig[i,i]
    if s<1.e-12:
        break
    X+= U.column(i)*B/s*V.column(i)

print "Solution: "+str(X)
```

On trouve :

```
(0.237037037037, 0.451851851852, 0.37037037037)
```

Notons que ci-dessus, la matrice A est de rang 2 (ce qu'on peut vérifier à l'aide de la commande `A.rank()`) et qu'elle n'est donc pas de rang maximum (3) ; il y a donc plusieurs solutions au problème de moindres carrés et la démonstration donnée ci-dessus montre que x est la solution de norme euclidienne minimale.

Imprimons les valeurs singulières :

```
sage: n = 3
sage: for i in range(0,n):
 ....:     print Sig[i,i]
```

On obtient :

`8.30931683326`, `1.39830388849` `2.27183194012e-16`.

A étant de rang 2, la troisième valeur singulière est forcément 0. On a donc ici une erreur d'arrondi, due à l'approximation flottante. Pour éviter une division par 0, il convient de ne pas en tenir compte dans le calcul de valeurs singulières approchées trop proches de 0 (c'est la raison du test `if s<1.e-12` dans le programme).

Exemple. Parmi les merveilleuses applications de la décomposition en valeurs singulières (SVD, Singular Value Decomposition), voici un problème qu'on aura bien du mal à résoudre avec une autre méthode : soient A et $B \in \mathbb{R}^{n\times m}$ les résultats d'une expérience répétée deux fois. On se demande si B peut être *tournée* sur A, c'est-à-dire s'il existe une matrice orthogonale Q telle que $A = QB$. Évidemment, un bruit s'est ajouté aux mesures et le problème n'a pas en général de solution. Il convient donc de le poser aux moindres carrés ; pour cela, il faut donc calculer la matrice orthogonale Q qui minimise le carré de la norme de Frobenius :

$$\|A - QB\|_F^2.$$

On se souvient que $\|A\|_F^2 = \operatorname{trace}({}^tA\,A)$. Alors

$$\|A - QB\|_F^2 = \operatorname{trace}({}^tA\,A) + \operatorname{trace}({}^tB\,B) - 2\,\operatorname{trace}({}^tQ\,{}^tB\,A) \geqslant 0,$$

et il faut donc maximiser $\operatorname{trace}({}^tQ\,{}^tB\,A)$. On calcule alors la SVD de ${}^tB\,A$: on a donc ${}^tU\,({}^tB\,A)V = \Sigma$. Soient σ_i les valeurs singulières ; soit $O = {}^tV\,{}^tQ\,U$. Cette matrice est orthogonale et donc tous ses coefficients sont inférieurs ou égaux à 1. Alors :

$$\operatorname{trace}({}^tQ\,{}^tB\,A) = \operatorname{trace}({}^tQ\,U\Sigma V) = \operatorname{trace}(O\Sigma) = \sum_{i=1}^{m} O_{ii}\sigma_i \leqslant \sum_{i=1}^{m} \sigma_i.$$

et le maximum est atteint pour $Q = U\,{}^tV$.

```
A = matrix(RDF,[[1,2],[3,4],[5,6],[7,8]])

#B est obtenue en ajoutant un bruit aléatoire A et
# en lui appliquant une rotation R d'angle thêta:
theta = 0.7
R = matrix(RDF,[[cos(theta),sin(theta)],[-sin(theta),cos(theta)]])
B = (A+0.1*random_matrix(RDF,4,2))*transpose(R)

C = transpose(B)*A
U,Sigma,V = C.SVD()

Q = U*transpose(V)
#la perturbation aleatoire est faible: Q doit donc etre proche de R.
print(Q)
print(R)
```

ce qui donne pour `Q` et `R` :

```
[  0.76737943398  0.641193265954]
[-0.641193265954   0.76737943398]

[ 0.764842187284  0.644217687238]
[-0.644217687238  0.764842187284]
```

qui sont effectivement proches.

Exercice 18 (Racine carrée d'une matrice symétrique semi définie positive)**.** Soit A une matrice symétrique semi définie positive (c'est-à-dire qui vérifie ${}^t x\, Ax \geqslant 0$ pour tout x non nul). Montrer qu'on peut calculer une matrice X, elle aussi symétrique semi définie positive, telle que $X^2 = A$.

5.2.8 Valeurs propres, vecteurs propres

Jusqu'à présent, nous n'avons utilisé que des méthodes directes (décomposition LU, QR, de Cholesky), qui fournissent une solution en un nombre fini d'opérations (les quatre opérations élémentaires, plus la racine carrée pour la décomposition de Cholesky). Cela ne *peut pas* être le cas pour le calcul des valeurs propres : en effet (cf. page 114), on peut associer à tout polynôme une matrice dont les valeurs propres sont les racines ; mais on sait qu'il n'existe pas de formule explicite pour le calcul des racines d'un polynôme de degré supérieur ou égal à 5, formule que donnerait précisément une méthode directe. D'autre part, former le polynôme caractéristique pour en calculer les racines serait extrêmement coûteux (cf. page 102) ; notons toutefois que l'algorithme de Leverrier permet de calculer le polynôme caractéristique d'une matrice de taille n en $O(n^4)$ opérations, ce qui est malgré tout considéré comme bien trop coûteux. Les méthodes numériques utilisées pour le calcul de valeurs et de vecteurs propres sont toutes itératives.

On va donc construire des suites convergeant vers les valeurs propres (et les vecteur propres) et arrêter les itérations quand on sera assez proche de la solution[4].

La méthode de la puissance itérée

Soit A une matrice appartenant à $\mathbb{C}^{n\times n}$. On choisit une norme quelconque $\|.\|$ sur $\mathbb{C}^n$. En partant de x_0, on considère la suite des x_k définie par :

$$x_{k+1} = \frac{Ax_k}{\|Ax_k\|}.$$

Si les valeurs propres vérifient $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$, alors la suite des vecteurs x_k converge vers un vecteur propre associé à la valeur propre dominante $|\lambda_1|$. De plus la suite $\nu_k = {}^t x_{k+1}\, x_k$ converge vers $|\lambda_1|$. Cette hypothèse de séparation des valeurs propres peut être relâchée.

```
sage: n = 10
sage: A = random_matrix(RDF,n); A = A*transpose(A)
sage: # A verifie (presque surement) les hypotheses.
sage: X = vector(RDF,[1 for i in range(0,n)])
sage:
sage: for i in range(0,1000):
....:     Y = A*X
....:     Z = Y/Y.norm()
....:     lam = Z*Y
....:     s = (X-Z).norm()
....:     print i, "\ts=", s, "\tlambda=", lam
....:     if s<1.e-10: break
....:     X = Z
0       s= 16.1640760201        lambda= 75.9549361783
1       s= 0.411503846291       lambda= 8.21816164112
2       s= 0.283595513527       lambda= 10.7020239604
3       s= 0.143945984315       lambda= 11.7626944491
4       s= 0.0671326308606      lambda= 12.0292765606
5       s= 0.0313379335883      lambda= 12.0876762358
6       s= 0.0149590182273      lambda= 12.1006031137
7       s= 0.00733280989323     lambda= 12.1036013532
8       s= 0.00368707185825     lambda= 12.1043343433
9       s= 0.00189514202573     lambda= 12.104522518
10      s= 0.000991461650756    lambda= 12.1045728607
...
```

Maintenant, utilisons :

```
sage: A.eigenvalues()
[12.1045923186, 6.62564474772, 5.45163183814, 4.81356332812,
```

[4]Dans les exemples donnés ci–dessous, le problème du choix des tests d'arrêt des itérations est volontairement passé sous silence, pour des raisons de simplicité.

```
2.46643846586, 1.37770690836, 0.966017076179, 0.653324011458,
0.0859636271843, 0.0541281947143]
```

On a bien calculé la valeur propre dominante.

L'intérêt de cette méthode peut sembler limité, mais il apparaîtra pleinement lors de l'étude des matrices creuses. Elle inspire aussi ce qui suit, qui est très utile.

La méthode de la puissance inverse avec translation

On suppose connue une *approximation* μ d'une valeur propre λ_j (μ et $\lambda_j \in \mathbb{C}$). Comment calculer un vecteur propre associé à λ_j ?

On fait l'hypothèse que $\forall k \neq j,\ 0 < |\mu - \lambda_j| < |\mu - \lambda_k|$, et donc, λ_j est une valeur propre simple. On considère alors $(A - \mu I)^{-1}$, dont la plus grande valeur propre est $(\lambda_j - \mu)^{-1}$, et on applique la méthode de la puissance itérée avec cette matrice.

Prenons par exemple

```
A = matrix(RDF,[[1,3,2],[1,2,3],[0,5,2]])
```

En appelant la méthode `A.eigenvalues()`, on trouve les valeurs propres `6.39294791649, 0.560519476112, -1.9534673926`. On va chercher le vecteur propre associé à la deuxième valeur propre, en partant d'une valeur approchée `mu` :

```
A = matrix(RDF,[[1,3,2],[1,2,3],[0,5,2]])
mu = 0.50519
AT = A-mu*identity_matrix(RDF,3)

x = vector(RDF,[1 for i in range(0,A.nrows())])
P,L,U = AT.LU()
for i in range(1,10):
    y = AT.solve_right(x)
    x = y/y.norm()
    lamb = x*A*x
    print x
    print lamb
```

On obtient, pour `x`, la suite :

```
(0.960798555257, 0.18570664547, -0.205862036435)
(0.927972943625, 0.10448610518, -0.357699412529)
(0.927691613383, 0.103298273577, -0.358772542336)
(0.927684531828, 0.10329496332, -0.35879180587)
(0.927684564843, 0.103294755297, -0.358791780397)
(0.927684562912, 0.103294757323, -0.358791784805)
(0.927684562945, 0.103294757253, -0.358791784742)
(0.927684562944, 0.103294757254, -0.358791784744)
```

et pour `lamb`, la suite :

```
1.08914936279, 0.563839629189, 0.560558807639, 0.560519659839,
0.560519482021,0.560519476073, 0.560519476114,0.560519476112.
```

On peut faire plusieurs remarques :

- on ne calcule pas l'inverse de la matrice $A - \mu I$, mais on utilise sa factorisation LU, qui est calculée une fois pour toutes ;
- on profite des itérations pour améliorer l'estimation de la valeur propre ;
- la convergence est très rapide ; on peut effectivement montrer que (moyennant les hypothèses ci-dessus, plus le choix d'un vecteur de départ non orthogonal au vecteur propre q_j associé à λ_j), on a, pour les itérés $x^{(i)}$ et $\lambda^{(i)}$:

$$\|x^{(i)} - q_j\| = O(|\frac{\mu - \lambda_j}{\mu - \lambda_K}|^i)$$

et

$$\|\lambda^{(i)} - \lambda_j\| = O(|\frac{\mu - \lambda_j}{\mu - \lambda_K}|^{2i})$$

où λ_K est la seconde valeur propre par ordre de proximité à μ ;

- le conditionnement de $(A - \mu I)$ (relié au rapport entre la plus grande et la plus petite valeur propre de $(A - \mu I)$) est mauvais ; mais on peut montrer (cf. [Sch91], qui cite Parlett) que les erreurs sont malgré tout sans importance !

La méthode QR

Soit A une matrice non singulière. On considère la suite $A_0 = A$, A_1, A_2, …, A_k, A_{k+1}, …. Dans la forme la plus brute de la méthode QR, le passage de A_k à A_{k+1} s'effectue ainsi :

1. on calcule la décomposition QR de A_k : $A_k = Q_k R_k$,
2. on calcule $A_{k+1} = R_k Q_k$.

Programmons cette méthode avec pour A une matrice symétrique réelle :

```
m = matrix(RDF,[[1,2,3,4],[1,0,2,6],[1,8,4,-2],[1,5,-10,-20]])
Aref = transpose(m)*m
A = copy(Aref)

for i in range(0,20):
    Q,R = A.QR()
    A = R*Q
    print
    print A
```

```
[      585.03055862 -4.21184953094e-13  3.29676890333e-14  -6.6742634342e-14]
[-3.04040944835e-13       92.9142649915  6.15831330805e-14   4.0818545963e-16]
[-1.53407859302e-39  7.04777999653e-25       4.02290947948  2.07979726915e-14]
[ 1.15814401645e-82 -4.17619045151e-68  6.16774251652e-42  0.0322669089941]
```

On constate que la convergence est rapide, vers une matrice diagonale. Les coefficients diagonaux sont les valeurs propres de A. Vérifions :

```
print(Aref.eigenvalues())
[585.03055862, 92.9142649915, 0.0322669089941, 4.02290947948]
```

On peut prouver la convergence si la matrice est hermitienne définie positive. Si on calcule avec une matrice non symétrique, il convient de travailler dans $\mathbb{C}$, les valeurs propres étant à priori complexes, et, si la méthode converge, les parties triangulaires inférieures des A_k tendent vers zéro, tandis que la diagonale tend vers les valeurs propres de A.

La méthode QR nécessite beaucoup d'améliorations pour être efficace, ne serait-ce que parce que les décompositions QR successives sont coûteuses ; parmi les raffinements utilisés, on commence en général par réduire la matrice A à une forme plus simple (forme de Hessenberg : triangulaire supérieure plus une sous diagonale), ce qui rend les décompositions QR bien moins coûteuses ; ensuite, pour accélérer la convergence il faut pratiquer des translations $A := A + \sigma I$ sur A, astucieusement choisies (voir par exemple [GVL96]). Notons que c'est la méthode utilisée par Sage quand on travaille avec des matrices pleines `CDF` ou `RDF`.

En pratique

Les programmes donnés ci-dessus sont là à titre d'exemples pédagogiques ; on utilisera donc les méthodes fournies par Sage qui, dans la mesure du possible, fait appel aux routines optimisées de la bibliothèque Lapack. Les interfaces permettent d'obtenir soit uniquement les valeurs propres, soit les valeurs et les vecteurs propres. Exemple :

```
sage: A = matrix(RDF,[[1,3,2],[1,2,3],[0,5,2]])
sage: A.eigenmatrix_right()
(
[ 6.39294791649              0              0]
[             0 0.560519476112              0]
[             0              0  -1.9534673926],

[ 0.542484060111  0.927684562944 0.0983425466742]
[ 0.554469286109  0.103294757254 -0.617227053099]
[ 0.631090211687 -0.358791784744  0.780614827195]
)
```

calcule la matrice diagonale des valeurs propres et la matrice des vecteurs propres (les lignes sont les vecteurs propres).

Exemple (Calcul des racines d'un polynôme)**.** Étant donné un polynôme (à coefficients réels ou complexes) $p(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, il est

facile de vérifier que les valeurs propres de la matrice compagnon M, définie par $M_{i+1,i} = 1$ et $M_{i,n} = -a_i$, sont les racines de p, ce qui fournit donc une méthode pour obtenir les racines de p :

```
sage: def pol2companion(p):
....:     n = len(p)
....:     m = identity_matrix(RDF,n)
....:     for i in range(0,n):
....:         m[i,n-1]=-p[i]
....:     return m

sage: q = [1,-1,2,3,5,-1,10,11]
sage: comp = pol2companion(q); comp
[  0.0   0.0   0.0   0.0   0.0   0.0   0.0  -1.0]
[  1.0   0.0   0.0   0.0   0.0   0.0   0.0   1.0]
[  0.0   1.0   0.0   0.0   0.0   0.0   0.0  -2.0]
[  0.0   0.0   1.0   0.0   0.0   0.0   0.0  -3.0]
[  0.0   0.0   0.0   1.0   0.0   0.0   0.0  -5.0]
[  0.0   0.0   0.0   0.0   1.0   0.0   0.0   1.0]
[  0.0   0.0   0.0   0.0   0.0   1.0   0.0 -10.0]
[  0.0   0.0   0.0   0.0   0.0   0.0   1.0 -11.0]
sage: racines = comp.eigenvalues(); racines
[0.347521510119 + 0.566550553398*I, 0.347521510119 - 0.566550553398*I,
0.345023776962 + 0.439908702386*I, 0.345023776962 - 0.439908702386*I,
-0.517257614325 + 0.512958206789*I, -0.517257614325 -
0.512958206789*I, -1.36699716455, -9.98357818097]
```

Dans cet exemple le polynôme est représenté par la liste `q` de ses coefficients, de 0 à $n-1$. Le polynôme $x^2 - 3x + 2$ serait ainsi représenté par `q=[2,-3]`.

5.2.9 Ajustement polynomial : le retour du diable

Version continue

On voudrait approcher la fonction $f(x)$ par un polynôme $P(x)$ de degré $\leqslant n$, sur l'intervalle $[\alpha, \beta]$. On pose le problème aux moindres carrés.

$$\min_{a_0,\ldots,a_n \in \mathbb{R}} J(a_0, \ldots, a_n) = \int_\alpha^\beta (f(x) - \sum_{i=0}^{n} a_i x^i)^2 \, \mathrm{d}x.$$

En dérivant $J(a_0, \ldots, a_n)$ par rapport aux coefficients a_i, on trouve que les a_i $i = 0, \ldots, n$ sont solution d'un système linéaire $Ma = F$ ou $M_{i,j} = \int_\alpha^\beta x^i x^j \, dx$ et $F_j = \int_\alpha^\beta x^j f(x) \, dx$. On voit immédiatement en regardant le cas $\alpha = 0$ et $\beta = 1$ que $M_{i,j}$ est la matrice de Hilbert ! Mais il y a un remède : il suffit d'utiliser une base de polynômes orthogonaux (par exemple, si $\alpha = -1$ et $\beta = 1$ la base des polynômes de Legendre) : alors la matrice M devient l'identité.

Version discrète

On considère m observations $y_i,\ i = 1, \ldots, m$ d'un phénomène aux points $x_i,\ i = 1, \ldots, m$. On veut ajuster un polynôme $\sum_{i=0}^{n} a_i x^i$ de degré n au plus égal à $m-1$ parmi ces points. Pour cela, on minimise donc la fonctionnelle :

$$J(a_0, \ldots, a_n) = \sum_{j=1}^{m} (\sum_{i=0}^{n} a_i x_j^i - y_j)^2.$$

Ainsi écrit, le problème va donner une matrice très proche de la matrice de Hilbert et le système sera difficile à résoudre. Mais on remarque alors que $\langle P, Q\rangle = \sum_{j=1}^{m} P(x_j) \cdot Q(x_j)$ définit un produit scalaire sur les polynômes de degré $n \leqslant m-1$. On peut donc d'abord fabriquer n polynômes échelonnés en degré, orthonormés pour ce produit scalaire, et donc diagonaliser le système linéaire. En se souvenant[5] que le procédé de Gram-Schmidt se simplifie en une récurrence à trois termes pour le calcul de polynômes orthogonaux, on cherche le polynôme $P_{n+1}(x)$ sous la forme $P_{n+1}(x) = xP_n(x) - \alpha_n P_{n-1}(x) - \beta_n P_{n-2}(x)$: c'est ce que fait la procédure `orthopoly` ci-dessous (on représente ici les polynômes par la liste de leurs coefficients : par exemple `[1,-2,3]` représente le polynôme $1 - 2x + 3x^2$).

L'évaluation d'un polynôme par le schéma de Horner se programme ainsi :

```
def eval(P,x):
    if len(P)==0:
        return 0
    else:
        return P[0]+x*eval(P[1:],x)
```

On peut ensuite programmer le produit scalaire de deux polynômes :

```
def pscal(P,Q,lx):
    return float(sum(eval(P,s)*eval(Q,s) for s in lx))
```

et l'opération `P` ← `P-a*Q` pour deux polynômes `P` et `Q` :

```
def subs(P,a,Q):
    for i in range(0,len(Q)):
        P[i]-=a*Q[i]
```

Un programme un peu sérieux doit lancer une exception quand il est mal utilisé ; dans notre cas, on lance l'exception définie ci-dessous quand $n \geqslant m$:

```
class BadParamsforOrthop(Exception):
    def __init__(self,degreplusun,npoints):
        self.deg = degreplusun
        self.np = npoints
    def __str__(self):
        return "degre: " + repr(self.deg) + " nb. points: " +
               repr(self.np)
```

[5] le prouver n'est pas très difficile !

La procédure suivante calcule les n polynômes orthogonaux :

```
def orthopoly(n,x):
    if n > len(x):
        raise BadParamsforOrthop(n-1,len(x))
    orth = [[1./sqrt(float(len(x)))]]
    for p in range(1,n):
        nextp = copy(orth[p-1])
        nextp.insert(0,0)
        s = []
        for i in range(p-1,max(p-3,-1),-1):
            s.append(pscal(nextp,orth[i],x))
        j = 0
        for i in range(p-1,max(p-3,-1),-1):
            subs(nextp,s[j],orth[i])
            j += 1
        norm = sqrt(pscal(nextp,nextp,x))
        nextpn = [nextp[i]/norm for i in range(0,len(nextp))]
        orth.append(nextpn)
    return orth
```

Une fois les polynômes orthogonaux $(P_i(x),\ i = 0, \ldots, n)$ calculés, la solution, est donnée par $P(x) = \sum_{i=0}^{n} \gamma_i P_i(x)$, avec :

$$\gamma_i = \sum_{j=1}^{m} P_i(x_j) y_j,$$

ce qu'on peut évidemment rapatrier sur la base des monômes $x^i,\ i = 1, \ldots, n$. Exemple $(n = 15)$:

```
X=[100*float(i)/L for i in range(0,40)]
Y=[float(1/(1+25*X[i]^2)+0.25*random()) for i in range(0,40)]
n = 15
orth = orthopoly(n,X)
```

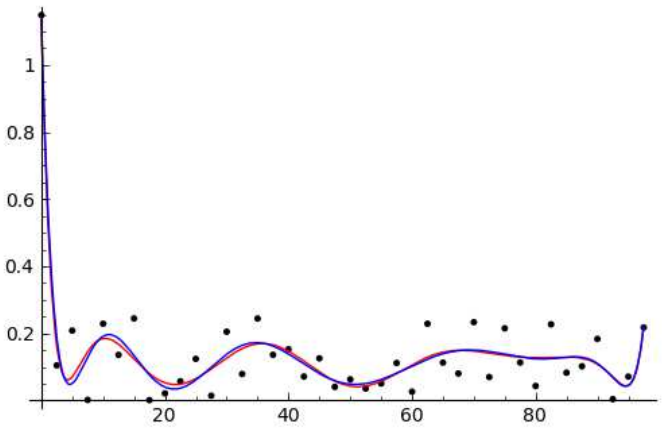
Calculons les coefficients de la solution sur la base des polynômes orthogonaux :

```
coeff= [sum(Y[j]*eval(orth[i],X[j]) for j in range(0,len(X)))
        for i in range(0,n)]
```

On peut ensuite transformer ce résultat dans la base des monômes $x^i,\ i = 0, \ldots, n$, afin par exemple de pouvoir en dessiner le graphe :

```
polmin=[0 for i in range(0,n)]
for i in range(0,n):
    add(polmin,coeff[i],orth[i])
p = lambda x: eval(polmin,x)
G = plot(p(x),x,0,X[len(X)-1])
show(G)
```

On ne détaille pas ici le calcul de l'ajustement naïf sur la base des monômes x^i, ni sa représentation graphique. On obtient :



Les deux courbes correspondant l'une à l'ajustement à base de polynômes orthogonaux, l'autre à la méthode naïve sont très proches, mais, en en calculant les résidus (la valeur de la fonctionnelle J) on trouve 0.1202 pour l'ajustement par les polynômes orthogonaux, et 0.1363 pour l'ajustement naïf.

5.2.10 Implantation et performances (pour les calculs avec des matrices pleines)

Les calculs avec des matrices à coefficients dans `RDF` sont effectués avec l'arithmétique flottante du processeur, ceux avec les matrices `RR` avec la bibliothèque MPFR. De plus dans le premier cas, Sage se sert de numpy/scipy, qui passe la main à la bibliothèque Lapack (codée en Fortran) et cette dernière bibliothèque utilise les BLAS[6] ATLAS optimisées pour chaque machine. Ainsi, on obtient, pour le calcul du produit de deux matrices de taille 1000 :

```
sage: a = random_matrix(RR,1000)
sage: b = random_matrix(RR,1000)
sage: %time a*b
CPU times: user 421.44 s, sys: 0.34 s, total: 421.78 s
Wall time: 421.79 s

sage:
sage: c = random_matrix(RDF,1000)
sage: d = random_matrix(RDF,1000)
sage: %time c*d
CPU times: user 0.18 s, sys: 0.01 s, total: 0.19 s
Wall time: 0.19 s
```

[6]Basic Linear Algebra Subroutines (produits matrice × vecteur, matrice × matrice, etc.).

soit un rapport de plus de 2000 entre les deux temps de calcul !

On peut aussi remarquer la rapidité des calculs avec des matrices à coefficients `RDF` : on vérifie immédiatement que le produit de deux matrices carrées de taille n coûte n^3 multiplications (et autant d'additions) ; ici, on effectue donc 10^9 multiplications en 0.18 seconde ; l'unité centrale de la machine de test battant à 3.1 Ghz., ce sont donc de environ 5 10^9 opérations qu'on effectue par seconde (en ne tenant pas compte des additions !) soit encore une vitesse de 5 gigaflops. On effectue donc *plus* d'une opération par tour d'horloge : ceci est rendu possible par l'appel presque direct de la routine correspondante de la bibliothèque ATLAS[7]. Notons qu'il existe un algorithme de coût inférieur à n^3 pour effectuer le produit de deux matrices : la méthode de Strassen. Elle n'est pas implantée en pratique (pour des calculs en flottant) pour des raisons de stabilité numérique. Le lecteur pourra vérifier, avec les programmes ci-dessus, que le temps de calcul avec Sage est bien proportionnel à n^3.

5.3 Matrices creuses

Les matrices creuses sont très fréquentes en calcul scientifique : le caractère creux (*sparsity* en anglais) est une propriété recherchée qui permet de résoudre des problèmes de grande taille, inaccessibles avec des matrices pleines.
Une définition approximative : on dira qu'une famille de matrices $\{M_n\}_n$ (de taille n) est un ensemble de matrices creuses si le nombre de coefficients non nuls de M_n est de l'ordre de $O(n)$.

Bien évidemment, ces matrices sont représentées en machine en utilisant des structures de données où ne sont stockés que les termes non nuls. En tenant compte du caractère creux des matrices, on veut bien sur gagner de la place mémoire et donc pouvoir manipuler de grandes matrices, mais aussi réduire fortement le côut des calculs.

5.3.1 Origine des systèmes creux

Problèmes aux limites

L'origine la plus fréquente est la discrétisation d'équations aux dérivées partielles. Considérons par exemple l'équation de Poisson (équation de la chaleur stationnaire) :

$$-\Delta u = f$$

où $u = u(x, y)$, $f = f(x, y)$,

$$\Delta u := \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

[7] Cette bibliothèque procède par blocs de taille automatiquement adaptée (déterminée par essais successifs) lors de la compilation : elle est en partie responsable du temps considérable qui est nécessaire à la compilation de Sage à partir du code source.

L'équation est posée dans le carré $[0,1]^2$, et munie de conditions aux limites $u = 0$ sur le bord du carré. L'analogue en dimension un est le problème

$$-\frac{\partial^2 u}{\partial x^2} = f, \tag{5.1}$$

avec $u(0) = u(1) = 0$.

Une des méthodes les plus simples pour approcher la solution de cette équation consiste à utiliser la méthode des différences finies : on découpe l'intervalle $[0,1]$ en un nombre fini N d'intervalles de pas h constant. On note u_i la valeur approchée de u au point $x_i = ih$. On approche la dérivée de u par $(u_{i+1} - u_i)/h$ et sa dérivée seconde par

$$\frac{(u_{i+1} - u_i)/h - (u_i - u_{i-1})/h}{h} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}.$$

On voit immédiatement que les $u_i,\ i = 1, \ldots, N$, qui approchent u aux points ih, sont solution d'un système linéaire n'ayant que 3 termes non nuls par ligne (et dont on peut vérifier que la matrice est symétrique définie positive).

En dimension 2, on peut plaquer une grille de pas h sur le carré unité et on obtient un système pentadiagonal (avec $4/h^2$ sur la diagonale, deux sur-diagonales et deux sous-diagonales dont les coefficients sont $-1/h^2$). En dimension 3, en procédant de la même manière dans un cube, on obtient un système où chaque ligne possède 7 coefficients non nuls. On a donc bien des matrices très creuses.

Marche aléatoire sur un grand graphe creux

On considère un graphe dans lequel chaque chaque sommet est relié à un petit nombre de sommets (petit par rapport au nombre total de sommets). Par exemple, on pourra se figurer un graphe dont les sommets sont les pages de l'internet : chaque page ne cite qu'un petit nombre d'autres pages (ce qui définit les arrêtes du graphe), mais c'est assurément un très grand graphe. Une marche aléatoire sur le graphe est décrite par une matrice stochastique, c'est à dire une matrice dont chaque coefficient est un réel compris entre 0 et 1 et dont la somme des coefficients de chaque ligne vaut 1. On montre qu'une telle matrice A a une valeur propre dominante égale à un. La distribution stationnaire de la marche aléatoire est le vecteur propre x à gauche associé à la valeur propre dominante, c'est à dire le vecteur qui vérifie $xA = x$. Une des applications les plus spectaculaires est l'algorithme *Pagerank* de *Google*, dans lequel le vecteur x sert à pondérer les résultats des recherches.

5.3.2 Sage et les matrices creuses

Sage permet de travailler avec des matrices creuses, en spécifiant `sparse = True` lors de la création de la matrice. Il s'agit d'un stockage sous forme de

dictionnaire. D'un autre coté, les calculs avec de grandes matrices creuses, à coefficients flottants (`RDF`, ou `CDF`) sont effectués par Sage avec la bibliothèque Scipy, qui offre ses propres classes de matrices creuses. Dans l'état actuel des choses, il n'existe pas d'interface entre les matrices `sparse` de Sage et les matrices creuses de Scipy. Il convient donc d'utiliser directement les objets de Scipy.

Les classes fournies par Scipy pour représenter des matrices matrices creuses sont :

- une structure sous forme de liste de listes (mais pas identique à celle utilisée par Sage) pratique pour créer et modifier des matrices, les `lil_matrix`,
- des structures immuables, ne stockant que les coefficients non nuls, et qui sont un standard de fait en algèbre linéaire creuse (formats `csr` et `csv`).

5.3.3 Résolution de systèmes linéaires

Pour des systèmes de taille modérée tels que ceux détaillés ci-dessus (en dimension 1 et 2), on peut utiliser une méthode directe, basée sur la décomposition LU. On peut se convaincre sans grande difficulté que, dans la décomposition LU d'une matrice A creuse, les facteurs L et U contiennent en général plus de termes non nuls à eux deux que A. Il faut utiliser des algorithmes de renumérotation des inconnues pour limiter la taille mémoire nécessaire, comme dans la bibliothèque SuperLU utilisée par Sage de manière transparente :

```
from scipy.sparse.linalg.dsolve import *
from scipy.sparse import lil_matrix
from numpy import array
n = 200
n2=n*n
A = lil_matrix((n2, n2))
h2 = 1./float((n+1)^2)
for i in range(0,n2):
    A[i,i]=4*h2
    if i+1<n2: A[i,i+1]=-h2
    if i>0:    A[i,i-1]=-h2
    if i+n<n2: A[i,i+n]=-h2
    if i-n>=0: A[i,i-n]=-h2
Acsc = A.tocsc()
B = array([1 for i in range(0,n2)])
solve = factorized(Acsc)#factorisation LU
S = solve(B) #resolution
```

Après avoir créé la matrice sous forme de `lil_matrix` il faut la convertir au format `csc`. Ce programme n'est pas particulièrement performant : la

construction de la `lil_matrix` est lente, les structures `lil_matrix` n'étant pas très efficaces. En revanche, la conversion en une matrice `csc` et la factorisation sont rapides et plus encore la résolution qui suit.

Méthodes itératives

Le principe de ces méthodes est de construire une suite qui converge vers la solution du système $Ax = b$. Les méthodes itératives modernes utilisent l'espace de Krylov K_n, espace vectoriel engendré par $b, Ab, A^2b \dots A^nb$. Parmi les méthodes les plus populaires, citons :

- la méthode du gradient conjugué : elle ne peut être utilisée que pour des systèmes dont la matrice A est symétrique définie positive. Dans ce cas $\|x\|_A = \sqrt{{}^t x\, Ax}$ est une norme, et l'itéré x_n est calculé de sorte à minimiser l'erreur $\|x - x_n\|_A$ entre la solution x et x_n pour $x_n \in K_n$ (il existe des formules explicites, faciles à programmer, cf. par exemple [LT94],[GVL96]).
- la méthode du résidu minimal généralisé (GMRES –generalized minimal residual–) : elle a l'avantage de pouvoir être utilisée pour des systèmes linéaires non symétriques. À l'itération n c'est la norme euclidienne du résidu $\|Ax_n - b\|_2$ qui est minimisée pour $x_n \in K_n$. On notera qu'il s'agit là d'un problème aux moindres carrés.

En pratique, ces méthodes ne sont efficaces que *préconditionnées :* au lieu de résoudre $Ax = b$, on résout $MAx = Mb$ où M est une matrice telle que MA est mieux conditionnée que A. L'étude et la découverte de préconditionneurs efficaces est une branche actuelle et riche de développements de l'analyse numérique. À titre d'exemple, voici la résolution du système étudié ci-dessus par la méthode du gradient conjugué , où le préconditioneur M, diagonal, est l'inverse de la diagonale de A. C'est un préconditionneur simple, mais peu efficace :

```
B = array([1 for i in range(0,n2)])
m = lil_matrix((n2, n2))
for i in range(0,n2):
    m[i,i]=1./A[i,i]
msc = m.tocsc()
X = cg(A,B,M = msc)
```

5.3.4 Valeurs propres, vecteurs propres

La méthode de la puissance itérée

La méthode de la puissance itérée est particulièrement adaptée au cas de très grandes matrices creuses ; en effet il suffit de savoir effectuer des produits matrice-vecteur (et des produits scalaires) pour savoir implanter l'algorithme. À titre d'exemple, revenons aux marches aléatoires sur un des

graphes creux, et calculons la distribution stationnaire, par la méthode de la puissance itérée :

```
from scipy import sparse, linsolve
from numpy.linalg import *
from numpy import array
from numpy.random import rand

def power(A,X):
    '''Puissance iteree'''
    for i in range(0,1000):
        Y = A*X
        Z = Y/norm(Y)
        lam = sum(X*Y)
        s = norm(X-Z)
        print i,"s= "+str(s)+" lambda= "+str(lam)
        if s<1.e-3:
            break
        X = Z
    return X

n = 1000
m = 5
#fabriquer une matrice stochastique de taille n
# avec m coefficients non nuls par ligne.
A1= sparse.lil_matrix((n, n))

for i in range(0,n):
    for j in range(0,m):
        l = int(n*rand())
        A1[l,i]=rand()

for i in range(0,n):
    s = sum(A1[i,0:n])
    A1[i,0:n] /= s

At = A1.transpose().tocsc()
X = array([rand() for i in range(0,n)])

# calculer la valeur propre dominante et le vecteur propre
# associe.
Y = power(At,X)
```

En exécutant ce script, on pourra s'amuser à chronométrer ses différentes parties, et on constatera que la quasi totalité du temps de calcul est passée dans la fabrication de la matrice ; la transposition n'est pas très coûteuse ; les itérations du calcul proprement dit occupent un temps négligeable sur les 7,5 secondes du calcul (sur la machine de test). Les stockages sous formes de

listes de matrices de grandes tailles ne sont pas très efficaces et ce genre de problème doit plutôt être traité avec des langages compilés et des structures de données adaptées.

Thème de réflexion : résolution de très grands systèmes non linéaires

La méthode la puissance itérée et les méthodes basées sur l'espace de Krylov partagent un propriété précieuse : il suffit de savoir calculer des produits matrice-vecteur pour pouvoir les implanter. On n'est même pas obligé de connaître la matrice, il suffit de connaître l'*action* de la matrice sur un vecteur. On peut donc faire calculs dans des cas où on ne connaît pas exactement la matrice ou des cas où on est incapable de la calculer. Les méthodes de Scipy acceptent en fait des *opérateurs linéaires* comme arguments. Voici une application à laquelle on pourra réfléchir (et qu'on pourra implémenter).

Soit $F : \mathbb{R}^n \to \mathbb{R}^n$. On veut résoudre $F(x) = 0$. La méthode de choix est la méthode de Newton, où on calcule les itérés $x_{n+1} = J(x_n)^{-1}.F(x_n)$ en partant de x_0. La matrice $J(x_n)$ est la jacobiennne de F en x_n, matrice des dérivées partielles de F en x_n. En pratique, on résoudra successivement $J(x_n)y_n = F(x_n)$, puis $x_{n+1} = x_n - y_n$. Il faut donc résoudre un système linéaire. Si F est un objet un peu compliqué à calculer, alors ses dérivées sont en général encore beaucoup plus difficiles à calculer et à coder et ce calcul peut même s'avérer pratiquement impossible. On a donc recours à la dérivation numérique : si e_j est le vecteur de $\mathbb{R}^n$ partout égal à 0 sauf sa j-ième composante égale à 1, alors $(F(x+he_j)-F(x))/h$ fournit une (bonne) approximation de la j-ième colonne de la matrice jacobienne. Il faudra donc faire $n+1$ évaluations de F pour obtenir J de manière approchée, ce qui est très coûteux si n est grand. Et si on applique une méthode itérative de type Krylov pour résoudre le système $J(x_n)y_n = F(x_n)$? On remarque alors que $J(x_n)V \simeq (F(x_n + hV) - F(x_n))/h$ pour h assez petit, ce qui évite de calculer *toute* la matrice. Dans Scipy, il suffira donc de définir un opérateur « linéaire » comme étant l'application $V \to (F(x_n+hV)-F(x_n))/h$. Ce genre de méthode est couramment utilisée pour la résolution de grands systèmes non linéaires. La « matrice » n'étant pas symétrique, on utilisera par exemple la méthode GMRES.

6

Intégration numérique et équations différentielles

Ce chapitre traite le calcul numérique d'intégrales (§6.1) ainsi que la résolution numérique d'équations différentielles (§6.2) avec Sage. Nous rappelons des bases théoriques des méthodes d'intégration, puis nous détaillons les fonctions disponibles et leur usage (§6.1.1).

Le calcul symbolique d'intégrales avec Sage a été traité précédemment (§2.3.8), et ne sera que mentionné rapidement ici comme une possibilité de calculer la valeur numérique d'une intégrale. Cette approche « symbolique puis numérique », lorsqu'elle est possible, constitue une des forces de Sage et est à privilégier car le nombre de calculs effectués, et donc d'erreurs d'arrondis est en général moindre que pour les méthodes d'intégration numérique.

Nous donnons une rapide introduction aux méthodes classiques de résolution d'équations différentielles, puis le traitement d'un exemple (§6.2.1) débutera l'inventaire des fonctions disponibles en Sage (§6.2.2).

6.1 Intégration numérique

On s'intéresse au calcul numérique d'intégrales de fonctions réelles : pour une fonction $f : I \longrightarrow \mathbb{R}$ on veut calculer

$$\int_I f(x)\,\mathrm{d}x$$

où I est un intervalle de $\mathbb{R}$.

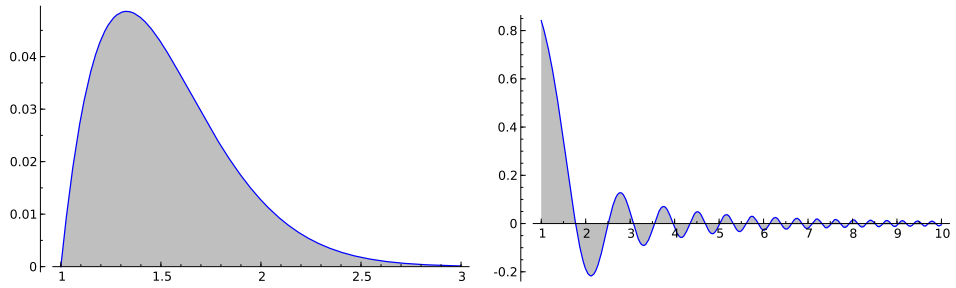


FIG. 6.1 – Les fonctions $x \mapsto \exp(-x^2)\log x$ et $x \mapsto \frac{\sin(x^2)}{x^2}$.

Par exemple, calculons

$$\int_1^3 \exp(-x^2)\ln(x)\,\mathrm{d}x.$$

```
sage: f(x) = exp(-x^2) * ln(x)
sage: N(integrate(f, x, 1, 3))
0.035860294991267694
sage: plot(f, 1, 3, fill='axis')
```

La fonction `integrate` calcule l'intégrale symbolique de l'expression, il faut demander explicitement une valeur numérique pour l'obtenir.

Il est aussi possible de calculer des intégrales sur un intervalle dont les bornes sont infinies :

```
sage: N(integrate(sin(x^2)/(x^2), x, 1, infinity))
0.2862504407259549
plot(sin(x^2)/(x^2), x, 1, 10, fill='axis')
```

Il existe plusieurs méthodes dans Sage pour calculer numériquement une intégrale, et si leurs implémentations différent techniquement, elles reposent toutes sur l'un des deux principes suivants :

- une interpolation polynomiale (méthode de Gauss-Kronrod en particulier),
- une transformation de fonction (méthode doublement exponentielle).

Méthodes interpolatoires

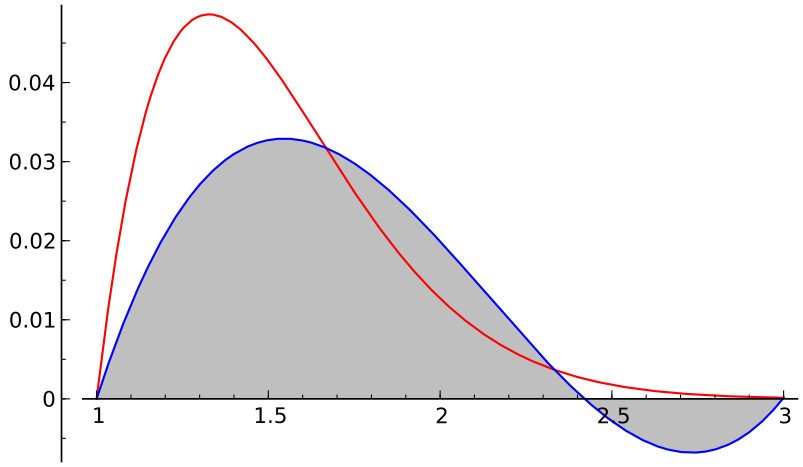
Dans ces méthodes, on évalue la fonction f à intégrer en un certain nombre n de points bien choisis $x_1, x_2, \ldots, x_n$, et on calcule une valeur approchée de l'intégrale de f sur $[a, b]$ par

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=1}^{n} w_i f(x_i).$$

Les coefficients w_i sont appelés les poids de la méthode, et ils sont déterminés par la condition que la méthode doit être exacte pour tout polynôme f de degré inférieur ou égal à $n-1$. Pour des points (x_i) fixés, les poids (w_i) sont uniquement déterminés par cette condition. On définit l'*ordre* de la méthode comme le degré maximal des polynômes qu'elle intègre exactement ; cet ordre est donc au moins $n-1$ par construction mais il peut être plus grand.

Ainsi la famille des méthodes d'intégration de Newton-Cotes (dont font partie les méthodes des rectangles, des trapèzes, la règle de Simpson) choisissent pour points d'intégration des points équirépartis sur l'intervalle $[a, b]$:

```
sage: fp = plot(f, 1, 3, color='red')
sage: n = 4
sage: interp_points = [(1+2*u/(n-1), N(f(1+2*u/(n-1)))) \
....: for u in xrange(n)]
sage: pp = plot(A.lagrange_polynomial(interp_points), 1, 3, \
....: fill='axis')
sage: show(fp+pp)
```



Pour les méthodes de type interpolatoire, on peut donc considérer que l'on calcule le polynôme d'interpolation de Lagrange de la fonction considérée et que l'intégrale de ce polynôme est choisie comme valeur approchée de l'intégrale de la fonction. Ces deux étapes sont en réalité condensées en une formule appelée « règle » de quadrature, le polynôme d'interpolation de Lagrange n'est jamais calculé explicitement. Le choix des points d'interpolation influe grandement sur la qualité de l'approximation polynomiale obtenue, et le choix de points équirépartis n'assure pas la convergence quand le nombre de points augmente (phénomène de Runge). La méthode d'intégration correspondante peut ainsi souffrir de ce problème illustré en Figure 6.2.

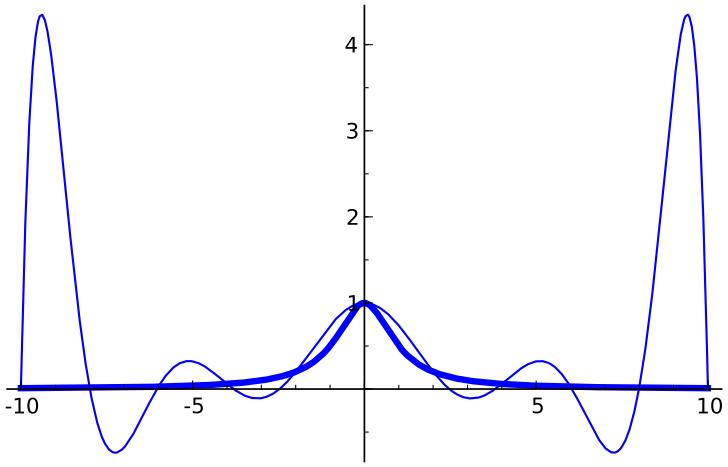


FIG. 6.2 – Interpolation par un polynôme de degré 10 (trait fin) de la fonction $x \mapsto 1/(1+x^2)$ (trait gras) sur 11 points équirépartis sur $[-10, 10]$. Le phénomène de Runge est apparent aux bornes.

Lorsqu'on demande à Sage de calculer numériquement une intégrale sur un intervalle quelconque, la méthode d'intégration n'est pas appliquée directement sur le domaine entier : on subdivise le domaine d'intégration en intervalles suffisamment petits pour que la méthode élémentaire donne un résultat assez précis (on parle de composition de méthodes). Comme stratégie de subdivision, on peut par exemple s'adapter dynamiquement à la fonction à intégrer : si on appelle $I_a^b(f)$ la valeur de $\int_a^b f(x)\,\mathrm{d}x$ calculée par la méthode d'intégration, on compare

$$I_0 = I_a^b(f)$$

avec

$$I_1 = I_a^{(a+b)/2}(f) + I_{(a+b)/2}^b(f)$$

et on arrête de subdiviser si $|I_0 - I_1|$ n'est pas significatif par rapport à la précision de calcul utilisée. C'est ici que la notion d'ordre d'une méthode est importante : pour une méthode d'ordre n, diviser l'intervalle d'intégration en 2 divise l'erreur théorique par 2^n, sans tenir compte des erreurs d'arrondis.

Une méthode de type interpolatoire particulière disponible dans Sage est la méthode de Gauss-Legendre. Dans cette méthode les n points d'intégration sont choisis comme les racines du polynôme de Legendre de degré n (avec un intervalle de définition correctement translaté à l'intervalle d'intégration considéré $[a, b]$). Les propriétés des polynômes de Legendre, orthogonaux pour le produit scalaire

$$\langle f, g\rangle = \int_a^b f(x)g(x)\,\mathrm{d}x$$

font que la méthode d'intégration obtenue calcule exactement les intégrales des polynômes de degré $2n-1$ inclus, au lieu de simplement jusqu'au degré $n-1$ comme on pouvait s'y attendre. De plus les poids d'intégration correspondants sont toujours positifs, ce qui rend la méthode moins vulnérable à des problèmes numériques du type *cancellation*[1].

Pour terminer sur les méthodes de type interpolatoire, la méthode de Gauss-Kronrod à $2n+1$ points est une « augmentation » de la méthode de Gauss-Legendre à n points :

- parmi les $2n+1$ points, n sont les points de la méthode de Gauss-Legendre ;
- la méthode est exacte pour tout polynôme de degré inférieur ou égal à $3n+1$.

On peut observer naïvement que les $3n+2$ inconnues (les $2n+1$ poids et les $n+1$ points ajoutés) sont *a priori* déterminés en exigeant que la méthode soit au moins d'ordre $3n+1$ (ce qui donne bien $3n+2$ conditions). Attention, les poids associés dans la méthode de Gauss-Kronrod aux n points de Gauss-Legendre n'ont aucune raison de coïncider avec ceux qui leur sont associés dans la méthode de Gauss-Legendre.

L'intérêt d'une telle méthode augmentée se manifeste lorsque l'on considère que le coût principal d'un algorithme d'intégration est le nombre d'évaluations de la fonction f à intégrer (en particulier si les points et poids sont tabulés). La méthode de Gauss-Kronrod étant en principe plus précise que celle de Gauss-Legendre, on peut utiliser son résultat I_1 pour valider le résultat I_0 de cette dernière et obtenir une estimation de l'erreur commise par $|I_1 - I_0|$, tout en minimisant le nombre d'appels à f. On peut comparer cette stratégie, particulière à la méthode de Gauss-Legendre, avec la stratégie plus générale de subdivision vue en page 128.

Méthodes doublement exponentielles

Les méthodes doublement exponentielles (DE) reposent sur le choix d'un changement de variable qui transforme un intervalle d'intégration borné à $\mathbb{R}$, et sur la très bonne précision obtenue par la méthode des trapèzes sur $\mathbb{R}$ pour les fonctions analytiques. Pour une fonction f intégrable sur $\mathbb{R}$ et un pas d'intégration h la méthode des trapèzes consiste à calculer

$$I_h = h \sum_{i=-\infty}^{+\infty} f(hi)$$

comme valeur approchée de $\int_{-\infty}^{+\infty} f(x)\,\mathrm{d}x$. Découverte en 1973 par Takahasi et Mori, la transformation doublement exponentielle est couramment utilisée

[1] Ce phénomène se produit lorsqu'une somme est significativement plus petite (en valeur absolue) que les termes de la somme : chaque erreur d'arrondi peut alors être plus grande que le résultat final, d'où une perte totale de précision.

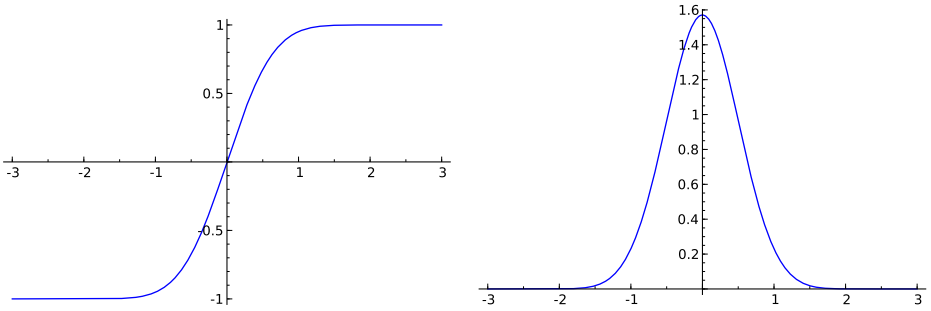


FIG. 6.3 – La transformation $\varphi(t) = \tanh(\frac{\pi}{2}\sinh t)$ utilisée dans la méthode doublement exponentielle (à gauche) et la décroissance de $\varphi'(x)$ (à droite).

par les logiciels d'intégration numérique. Une introduction à la transformation et à sa découverte est donnée dans [Mor05] ; l'essentiel en est restitué ici. On y trouve en particulier une explication de la surprenante précision obtenue par la méthode des trapèzes (elle est optimale dans un certain sens) pour les fonctions analytiques sur $\mathbb{R}$.

Pour calculer

$$I = \int_{-1}^{1} f(x)\,\mathrm{d}x$$

il est possible d'utiliser une transformation

$$x = \varphi(t)$$

où φ est analytique sur $\mathbb{R}$ et vérifie

$$\lim_{t\to-\infty} \varphi(t) = -1, \lim_{t\to\infty} \varphi(t) = 1$$

et alors

$$I = \int_{-\infty}^{\infty} f(\varphi(t))\varphi'(t)\,\mathrm{d}t.$$

En appliquant la formule des trapèzes à cette dernière expression on calcule

$$I_h^N = \sum_{k=-N}^{N} f(\varphi(kh))\varphi'(kh)$$

pour un certain pas de discrétisation h et en tronquant la somme aux termes de $-N$ à N. Le choix de transformation proposé est

$$\varphi(t) = \tanh\left(\frac{\pi}{2}\sinh t\right)$$

qui donne la formule

$$I_h^N = \sum_{k=-N}^{N} f\left(\tanh\left(\frac{\pi}{2}\sinh kh\right)\right)\frac{\frac{\pi}{2}\cosh kh}{\cosh^2(\frac{\pi}{2}\sinh kh)}.$$

La formule doit son nom à la décroissance doublement exponentielle de

$$\varphi'(t) = \frac{\frac{\pi}{2}\cosh t}{\cosh^2(\frac{\pi}{2}\sinh t)}$$

quand $|t| \to \infty$ (voir Figure 6.3). Le principe de la transformation est de concentrer l'essentiel de la contribution de la fonction à intégrer autour de 0, d'où la forte décroissance quand $|t|$ croît. Il y a un compromis à trouver dans le choix des paramètres et de la transformation φ utilisée : une décroissance plus rapide que doublement exponentielle diminue l'erreur de troncature mais augmente l'erreur de discrétisation.

Depuis la découverte de la transformation DE, cette méthode est appliquée seule ou avec d'autres transformations, en fonction de la nature de l'intégrande, de ses singularités et du domaine d'intégration. Un tel exemple est la décomposition sinus cardinal « sinc » :

$$f(x) \approx \sum_{k=-N}^{N} f(kh) S_{k,h}(x)$$

où

$$S_{k,h}(x) = \frac{\sin(\pi(x-kh)/h)}{\pi(x-kh)/h}$$

qui est utilisée conjointement à la méthode doublement exponentielle dans [TSM05], améliorant les formules précédentes qui utilisaient une transformation simplement exponentielle $\varphi(x) = \tanh(x/2)$. La fonction sinc est définie par

$$\text{sinc} = \begin{cases} 1 & \text{si } x = 0, \\ \frac{\sin(\pi x)}{\pi x} & \text{sinon} \end{cases}$$

et son graphe est donné en Figure 6.4.

Le choix de la transformation à utiliser détermine en grande partie la qualité du résultat en présence de singularités aux bornes (il n'y a cependant pas de bonne solution en cas de singularités à l'intérieur de l'intervalle). Nous verrons plus loin que dans la version de Sage considérée, seules les fonctions d'intégration de PARI utilisent des transformations doublement exponentielles avec la possibilité de préciser le comportement aux bornes.

6.1.1 Manuel des fonctions d'intégration disponibles

Nous allons maintenant voir plus en détail les différentes façons de calculer une intégrale numérique avec Sage, à travers quelques exemples de calculs d'intégrales :

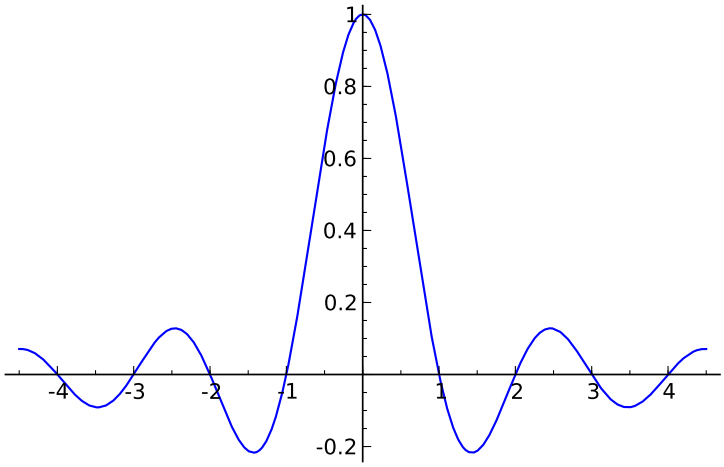


FIG. 6.4 – La fonction sinus cardinal.

$$
\begin{aligned}
I_1 &= \int_{17}^{42} \exp(-x^2)\log(x)\,\mathrm{d}x, \quad I_2 = \int_0^1 x\log(1+x)\,\mathrm{d}x = \frac{1}{4},\\
I_3 &= \int_0^1 \sqrt{1-x^2}\,\mathrm{d}x = \frac{\pi}{4},\\
I_4 &= \int_0^1 \max(\sin(x),\cos(x))\,\mathrm{d}x = \int_0^{\frac{\pi}{4}} \cos(x)\,\mathrm{d}x + \int_{\frac{\pi}{4}}^1 \sin(x)\,\mathrm{d}x\\
&= \sin\frac{\pi}{4} + \cos\frac{\pi}{4} - \cos 1 = \sqrt{2} - \cos 1,\\
I_5 &= \int_0^1 \sin(\sin(x))\,\mathrm{d}x, \quad I_6 = \int_0^\pi \sin(x)\exp(\cos(x))\,\mathrm{d}x = e - \frac{1}{e},\\
I_7 &= \int_0^1 \frac{1}{1+10^{10}x^2}\,\mathrm{d}x = 10^{-5}\arctan(10^5), \quad I_8 = \int_0^{1,1} \exp(-x^{100})\,\mathrm{d}x,\\
I_9 &= \int_0^{10} x^2\sin(x^3)\,\mathrm{d}x = \frac{1}{3}(1-\cos(1000)), \quad I_{10} = \int_0^1 \sqrt{x}\,\mathrm{d}x = \frac{2}{3}.
\end{aligned}
$$

Nous ne donnons pas une description exhaustive de l'usage des fonctions d'intégration traitées—cela se trouve dans l'aide en ligne—mais uniquement leur usage le plus courant.

`N(integrate(...))`

La première méthode que l'on peut utiliser avec Sage pour calculer numériquement est `N(integrate(...))` :

```
sage: N(integrate(exp(-x^2)*log(x), x, 17, 42))
2.5657285006962035e-127
```

Il faut remarquer qu'il n'est pas garanti que l'intégrale sera calculée numériquement par ce biais, en effet la fonction `integrate` demande une intégration symbolique. Si celle-ci est possible, alors Sage ne fera qu'évaluer numériquement l'expression symbolique obtenue :

```
sage: integrate(log(1+x)*x, x, 0, 1)
1/4
sage: N(integrate(log(1+x)*x, x, 0, 1))
0.250000000000000
```

`numerical_integral`

La fonction `numerical_integral` demande *explicitement* une intégration numérique de la fonction donnée en paramètre. Elle utilise pour ceci la bibliothèque numérique GSL qui implémente la méthode de Gauss-Kronrod pour un nombre de points d'intégration n fixé. Les points et poids sont précalculés, et la précision est limitée à la précision des nombres flottants machine (53 bits de mantisse). Le résultat est un couple composé du résultat calculé et d'une estimation de l'erreur :

```
sage: numerical_integral(exp(-x^2)*log(x), 17, 42)
(2.5657285006962035e-127, 3.3540254049238093e-128)
```

L'estimation de l'erreur n'est pas une borne garantie de l'erreur commise, mais une simple valeur indicative de la difficulté à calculer l'intégrale donnée. On constate que dans l'exemple ci-dessus l'estimation donnée de l'erreur est telle que l'on peut douter de presque tous les chiffres du résultat (sauf le premier).

Les arguments de `numerical_integral` permettent notamment :

- de choisir le nombre de points utilisés (six choix allant de 15 à 61 avec 61 par défaut) ;
- de demander une subdivision adaptative (choix par défaut), ou d'imposer une application directe sans composition de la méthode sur l'intervalle d'intégration (par l'ajout de `algorithm='qng'`) ;
- de borner le nombre d'appels à l'intégrande.

Empêcher GSL de procéder à une intégration adaptative peut entraîner une perte de précision :

```
sage: numerical_integral(exp(-x^100), 0, 1.1)
(0.99432585119150096, 4.0775730413694644e-09)
sage: numerical_integral(exp(-x^100), 0, 1.1, algorithm='qng')
(0.99432753857653178, 0.016840666914688864)
```

Lorsque la fonction `integrate` ne trouve pas d'expression analytique correspondant à l'intégrale demandée, elle renvoie une expression symbolique inchangée :

```
sage: integrate(exp(-x^2)*log(x), x, 17, 42)
integrate(e^(-x^2)*log(x), x, 17, 42)
```

et le calcul numérique effectué via `N` utilise `numerical_integral`. Ceci explique en particulier pourquoi le paramètre de la précision sera ignoré dans ces cas là :

```
sage: N(integrate(exp(-x^2)*log(x), x, 17, 42), 200)
2.5657285006962035e-127
```

mais on aura :

```
sage: N(integrate(sin(x)*exp(cos(x)), x, 0, pi), 200)
2.3504023872876029137647637011912016303114359626681917404591
```

car l'intégration symbolique est possible dans ce cas.

`sage.calculus.calculus.nintegral`

Pour les fonctions définies symboliquement, il est possible de demander à Maxima de calculer numériquement une intégrale :

```
sage: sage.calculus.calculus.nintegral(sin(sin(x)), x, 0, 1)
(0.43060610312069059, 4.7806881022870532e-15, 21, 0)
```

mais il est aussi possible d'appeler la méthode `nintegral` directement sur l'objet Maxima :

```
sage: g(x) = sin(sin(x))
sage: g.nintegral(x, 0, 1)
(0.43060610312069059, 4.7806881022870532e-15, 21, 0)
```

Maxima utilise la bibliothèque d'intégration numérique QUADPACK, qui comme GSL est limitée aux nombres flottants machine. La fonction `nintegral` utilise une stratégie de subdivision adaptative de l'intervalle d'intégration, et il est possible de préciser :

- la précision relative désirée pour la sortie ;
- le nombre maximal de sous-intervalles considérés pour le calcul.

La sortie est un tuple :

1. la valeur approchée de l'intégrale ;
2. une estimation de l'erreur absolue ;
3. le nombre d'appels à l'intégrande ;
4. un code d'erreur (0 si aucun problème n'a été rencontré, pour plus de détails sur les autres valeurs possibles vous pouvez consulter la documentation avec `sage.calculus.calculus.nintegral ?`).

`gp('intnum(...)')`

Le calculateur PARI est disponible depuis Sage et implémente aussi une commande pour l'intégration numérique de fonctions nommée `intnum` :

```
sage: gp('intnum(x=17, 42, exp(-x^2)*log(x))')
2.56572850056105148291735639613047859 00 E-127
```

La fonction `intnum` utilise la méthode doublement exponentielle.

Il est possible de choisir la précision du résultat en modifiant la précision globale de l'interprète PARI :

```
sage: gp('intnum(x=0, 1, sin(sin(x)))')
0.43060610312069060491237735524846578643
sage: gp.set_precision(50)
38
sage: gp('intnum(x=0, 1, sin(sin(x)))')
0.43060610312069060491237735524846578643360804182200
```

Un inconvénient majeur de cette méthode est que la fonction à intégrer est à donner sous la forme d'une chaîne en respectant la syntaxe de PARI, il n'est pas possible d'intégrer des fonctions arbitraires par ce biais.

La fonction `intnum` permet d'indiquer le comportement de l'intégrande aux bornes de l'intervalle d'intégration. L'exemple suivant illustre l'impact que cela peut avoir sur la précision du résultat. Intégrons $x \mapsto x^{-1/2}$ sans indication :

```
sage: gp('intnum(x=0, 1, x^(-1/2))')
1.99999999999999999999999999999981458433
```

En indiquant la nature de la singularité, à savoir que la fonction se comporte « comme » $x \mapsto x^{-1/2}$ en 0 :

```
sage: gp('intnum(x=[0, -1/2], 1, x^(-1/2))')
2.00000000000000000000000000000000000000
```

L'utilisateur est responsable de l'exactitude du comportement annoncé ; si par exemple nous indiquons (de façon erronée) que la fonction se comporte comme $x \mapsto x^{-1/42}$ en 0, l'erreur commise restera importante :

```
sage: gp('intnum(x=[0, -1/42], 1, x^(-1/2))')
1.99999999999999999999999999999996571353
```

`mpmath.quad*`

La bibliothèque `mpmath` est une bibliothèque de calcul numérique en précision arbitraire écrite en Python. Elle est capable de calculer avec des

nombres flottants réels et complexes, des matrices, et des intervalles de nombres flottants réels.

Elle dispose de fonctions d'intégration numérique (la principale étant `quad`) et est disponible en Sage, il suffit de l'importer :

```
sage: import mpmath
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.43060610312069059')
```

La valeur de sortie n'est pas garantie être exacte avec toute la précision demandée, comme on le constate sur l'exemple suivant :

```
sage: mpmath.mp.prec = 113
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.430606103120690604912377355248465809')
sage: mpmath.mp.prec = 114
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.430606103120690604912377355248465785')
```

Il est possible de spécifier la précision demandée en nombre de chiffres décimaux (`mpmath.mp.dps`) ou en nombre de bits (`mpmath.mp.prec`). Dans un souci de cohérence avec la fonction `N` de Sage nous nous restreignons à une précision en bits.

La fonction `mpmath.quad` peut faire appel soit à la méthode de Gauss-Legendre, soit à la transformation doublement exponentielle (c'est cette dernière qui est utilisée par défaut). Il est possible de spécifier directement la méthode à utiliser avec les fonctions `mpmath.quadgl` et `mpmath.quadts`.[2]

Une limitation importante pour l'utilisation des fonctions d'intégration de `mpmath` dans Sage est qu'elles ne savent pas manipuler directement des fonctions arbitraires définies en Sage :

```
sage: mpmath.quad(sin(sin(x)), [0, 1])
[...]
TypeError: no canonical coercion from \
 <type 'sage.libs.mpmath.ext_main.mpf'> to Symbolic Ring
```

La situation est cependant moins problématique que pour l'utilisation de PARI qui est elle limitée à une interaction en mode texte. Il est en effet possible d'ajouter les fonctions d'évaluation et de conversion nécessaires pour pouvoir intégrer via `mpmath.quad` des fonctions quelconques :

```
sage: def g(x):
....:     return max(sin(x), cos(x))
sage: mpmath.mp.prec = 100
sage: mpmath.quadts(lambda x: g(N(x, 100)), [0, 1])
mpf('0.87391241626303543595797908625 2')
```

[2] du nom de la transformation utilisée $\varphi : t \mapsto \tanh(\pi/2 \sinh(t))$ et vue précédemment.

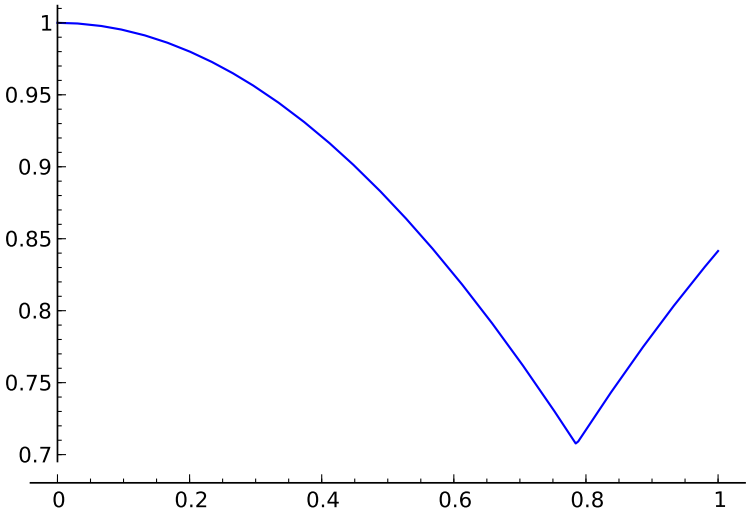


FIG. 6.5 – La fonction $x \mapsto \max(\sin(x), \cos(x))$. L'irrégularité en $\pi/4$ rend l'intégration numérique très problématique.

On constate que l'intégration de fonctions non régulières (comme l'exemple de I_4 ci-dessus) peut entraîner des pertes de précision conséquentes, même en demandant le résultat avec une grande précision :

```
sage: mpmath.mp.prec = 170
sage: mpmath.quadts(lambda x: g(N(x, 190)), [0, 1])
mpf('0.873910907574009752053930059819624763440541483541887 94')
sage: N(sqrt(2) - cos(1), 100)
0.87391125650495533140075211677
```

Seuls 5 chiffres sont corrects ici. On peut néanmoins aider `mpmath` en suggérant une subdivision de l'intervalle d'intégration (ici, au point irrégulier, cf. Figure 6.5) :

```
sage: mpmath.quadts(lambda x: g(N(x, 170)), \
....: [0, mpmath.pi / 4, 1])
mpf('0.87391125650495533140075211676672147483736145475902551')
```

Les fonctions discontinues représentent un « piège » classique pour les méthodes d'intégration ; néanmoins une stratégie d'adaptation automatique par subdivision, telle qu'évoquée plus haut, peut limiter les dégâts.

Exercice 19 (Calcul des coefficients de Newton-Cotes)**.** On cherche à calculer les coefficients de la méthode de Newton-Cotes à n points, celle-ci n'étant pas disponible dans Sage. On considère pour simplifier que l'intervalle d'intégration est $I = [0, n-1]$, les points d'intégration étant alors

$x_1 = 0, x_2 = 1, \ldots, x_n = n-1$. Les coefficients (w_i) de la méthode sont tels que l'équation

$$\int_0^{n-1} f(x)\,\mathrm{d}x = \sum_{i=0}^{n-1} w_i f(i) \tag{6.1}$$

est exacte pour tout polynôme f de degré inférieur ou égal à $n-1$.

1. On considère pour $i \in \{0, \ldots, n-1\}$ le polynôme $P_i(X) = \prod_{\substack{j=1 \\ j\neq i}}^{n-1}(X - x_i)$. En appliquant l'équation (6.1) à P_i, exprimer w_i en fonction de P_i.
2. En déduire une fonction `NCRule` qui à n associe les coefficients de la méthode de Newton-Cotes à n points sur l'intervalle $[0, n-1]$.
3. Montrer comment appliquer ces poids sur un segment d'intégration $[a, b]$ quelconque.
4. Écrire une fonction `QuadNC` qui calcule l'intégrale d'une fonction sur un segment de $\mathbb{R}$ donné en paramètre. Comparer les résultats avec les fonctions d'intégration disponibles en Sage sur les intégrales I_1 à I_{10}.

6.2 Équations différentielles numériques

On s'intéresse dans cette section à la résolution numérique d'équations différentielles ordinaires. Les fonctions de résolution disponibles dans Sage sont capables de traiter des systèmes d'équations de la forme :

$$\begin{cases} \frac{\mathrm{d}y_1}{\mathrm{d}t}(t) &= f_1(t, y_1(t), y_2(t), \ldots, y_n(t)) \\ \frac{\mathrm{d}y_2}{\mathrm{d}t}(t) &= f_2(t, y_1(t), y_2(t), \ldots, y_n(t)) \\ \qquad\vdots & \\ \frac{\mathrm{d}y_n}{\mathrm{d}t}(t) &= f_n(t, y_1(t), y_2(t), \ldots, y_n(t)) \end{cases}$$

avec des conditions initiales $(y_1(0), y_2(0), \ldots, y_n(0))$ connues.

Cette formalisation permet aussi de s'intéresser à des problèmes de dimension supérieure à 1, en introduisant des variables supplémentaires (voir l'exemple développé en (§6.2.1)). Elle ne permet cependant pas d'exprimer le système d'équations vérifié par la fonction ρ de Dickman :

$$\begin{cases} u\rho'(u) + \rho(u-1) &= 0 \quad \text{pour } u \geqslant 1 \\ \rho(u) &= 1 \quad \text{pour } 0 \leqslant u \leqslant 1. \end{cases}$$

Les outils de résolution d'équations différentielles ordinaires ne sont en effet pas adaptés à une telle équation (dite *à retard*).

Les méthodes de résolution numérique dites « à un pas » utilisent toutes un même principe général : pour un pas h et des valeurs de $y(t_0)$ et $y'(t_0)$ connues, on calcule une valeur approchée de $y(t_0+h)$ à partir d'une estimation

de $y'(t)$ prise sur l'intervalle $[t_0, t_0 + h]$. Par exemple la méthode la plus simple consiste à faire l'approximation :

$$\begin{aligned} \forall t \in [t_0, t_0 + h], y'(t) &\approx y'(t_0) \\ \int_{t_0}^{t_0+h} y'(t)\,\mathrm{d}t &\approx hy'(t_0) \\ y(t_0 + h) &\approx y(t_0) + hy'(t_0). \end{aligned}$$

On reconnaît ici la même approximation utilisée que dans la méthode d'intégration numérique des rectangles. La méthode obtenue est d'ordre 1, c'est-à-dire que l'erreur obtenue après un pas de calcul est en $\mathcal{O}(h^2)$ sous l'hypothèse que f est suffisamment régulière. De manière générale une méthode est d'ordre p si l'erreur qu'elle commet sur un pas de longueur h est en $\mathcal{O}(h^{p+1})$. La valeur obtenue en $t_1 = t_0 + h$ sert alors de point de départ pour progresser d'un pas de plus, aussi loin que désiré.

Cette méthode d'ordre 1, appelée méthode d'Euler, n'est pas réputée pour sa précision (tout comme la méthode des rectangles pour l'intégration) et il existe des méthodes d'ordre plus élevé, par exemple la méthode Runge-Kutta d'ordre 2 :

$$\begin{aligned} k_1 &= hf(t_n, y(t_n)) \\ k_2 &= hf(t_n + \frac{1}{2}h, y(t_n) + \frac{1}{2}k_1) \\ y(t_{n+1}) &\approx y(t_n) + k_2 + \mathcal{O}(h^3). \end{aligned}$$

Dans cette méthode, on essaie d'évaluer $y'(t_n + h/2)$ pour obtenir une meilleure estimation de $y(t_n + h)$.

On trouve aussi des méthodes multi-pas (c'est le cas par exemple des méthodes de Gear) qui consistent à calculer $y(t_n)$ à partir des valeurs déjà obtenues $(y(t_{n-1}), y(t_{n-2}), \ldots, y(t_{n-\ell}))$ pour un certain nombre de pas ℓ. Ces méthodes démarrent nécessairement par une phase d'initialisation avant qu'un nombre suffisant de pas soient calculés.

Tout comme la méthode de Gauss-Kronrod pour l'intégration numérique, il existe des méthodes hybrides pour la résolution d'équations différentielles. Ainsi la méthode de Dormand et Prince calcule avec les mêmes points d'approximation une valeur aux ordres 4 et 5, la deuxième servant pour l'estimation d'erreur de la première. On parle dans ce cas de méthode adaptative.

On distingue encore les méthodes dites explicites des méthodes implicites : dans une méthode explicite, la valeur de $y(t_{n+1})$ est donnée par une formule n'utilisant que des valeurs connues ; pour une méthode implicite il faut résoudre une équation. Prenons l'exemple de la méthode d'Euler implicite :

$$y(t_n + h) = y(t_n) + hf(t_n + h, y(t_n + h)).$$

On constate que la valeur recherchée $y(t_n + h)$ apparaît des deux côtés de l'équation ; si la fonction f est suffisamment complexe il faudra résoudre un système algébrique non linéaire, typiquement avec la méthode de Newton.

A priori, on s'attend à obtenir des résultats plus précis à mesure que l'on diminue le pas d'intégration ; outre le coût en calculs supplémentaires que cela représente, ce gain espéré en précision est toutefois tempéré par un plus grand nombre d'erreurs d'arrondi qui risquent, à la longue, de polluer le résultat.

6.2.1 Exemple de résolution

Considérons l'oscillateur de Van der Pol de paramètre μ vérifiant l'équation différentielle suivante :

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t) - \mu(1-x^2)\frac{\mathrm{d}x}{\mathrm{d}t}(t) + x(t) = 0.$$

Posons $y_0(t) = x(t)$ et $y_1(t) = \frac{\mathrm{d}x}{\mathrm{d}t}$, on obtient ce système d'ordre 1 :

$$\begin{cases} \frac{\mathrm{d}y_0}{\mathrm{d}t} &= y_1, \\ \frac{\mathrm{d}y_1}{\mathrm{d}t} &= \mu(1-y_0^2)y_1 - y_0. \end{cases}$$

Pour le résoudre, nous allons utiliser un objet « solveur » que l'on obtient avec la commande `ode_solver` :

```
sage: T = ode_solver()
```

Un objet solveur sert à enregistrer les paramètres et la définition du système que l'on cherche à résoudre ; il donne accès aux fonctions de résolution numérique d'équations différentielles disponibles dans la bibliothèque GSL, déjà mentionnée au sujet de l'intégration numérique.

Les équations du système sont renseignées sous la forme d'une fonction :

```
sage: def f_1(t,y,params):
....:     return[y[1],-y[0]-params[0]*y[1]*(y[0]**2-1.0)]
sage: T.function=f_1
```

Le paramètre `y` représente le vecteur des fonctions inconnues, et on doit renvoyer en fonction de t et d'un paramètre optionnel le vecteur des membres droits du système d'équations.

Certains des algorithmes de GSL nécessitent de plus de connaître la jacobienne du système (la matrice dont le terme (i, j) est $\frac{\partial f_i}{\partial y_j}$) :

```
sage: def j_1(t,y,params):
....:     return [ [0.0, 1.0],
....:              [-2.0*params[0]*y[0]*y[1]-1.0,
....:               -params[0]*(y[0]*y[0]-1.0)],
....:              [0.0,0.0] ]
sage: T.jacobian=j_1
```

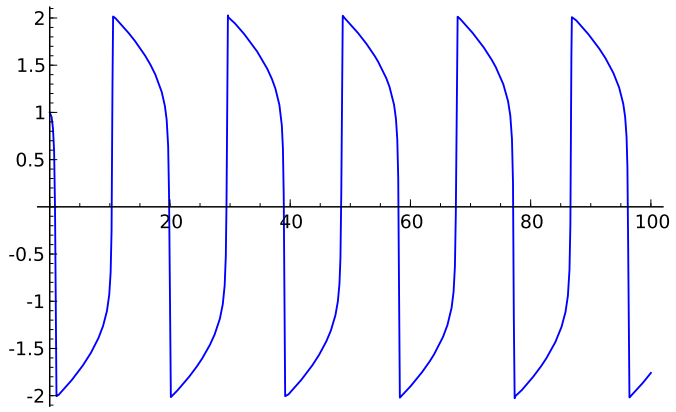
Il est maintenant possible de demander une résolution numérique. On choisit l'algorithme, l'intervalle sur lequel calculer la solution et le nombre de pas voulus (ce qui détermine h) :

```
sage: T.algorithm="rk8pd"
sage: T.ode_solve(y_0=[1,0],t_span=[0,100],params=[10.0],
....:             num_points=1000)
sage: f = T.interpolate_solution()
```

Ici on a pris l'algorithme de Runge-Kutta Dormand-Prince pour calculer la solution sur $[0, 100]$; les conditions initiales ainsi que la valeur des paramètres (ici un seul) sont précisées.

Pour afficher la solution :

```
sage: P = plot(f, 0, 100)
sage: show(P)
```



6.2.2 Fonctions de résolution disponibles

Nous avons déjà évoqué pour les objets solveur de GSL la méthode `rk8pd`. D'autres méthodes sont disponibles :

`rkf45` : Runga-Kutta-Felhberg, une méthode adaptative d'ordres 5 et 4 ;

`rk2` : Runge-Kutta adaptative d'ordres 3 et 2 ;

`rk4` : la méthode Runge-Kutta classique d'ordre 4 ;

`rk2imp` : une méthode de Runge-Kutta implicite d'ordre 2 avec évaluation au milieu de l'intervalle ;

`rk4imp` : une méthode de Runge-Kutta implicite d'ordre 4 avec évaluation aux « points Gaussiens »[3] ;

`bsimp` : la méthode implicite de Burlisch-Stoer ;

[3] Il s'agit des racines du polynôme de Legendre de degré 2 décalés sur l'intervalle $[t, t+h]$, nommées en référence à la méthode d'intégration de Gauss-Legendre.

`gear1` : la méthode implicite de Gear à un pas ;

`gear2` : la méthode implicite de Gear à deux pas.

Nous renvoyons le lecteur intéressé par les détails de toutes ces méthodes à [AP98].

Il faut noter que la limitation de GSL aux flottants machine—donc de précision fixée—évoquée pour l'intégration numérique reste valide pour les méthodes de résolution d'équations différentielles.

Maxima dispose aussi de ses routines de résolution numérique, avec sa syntaxe propre :

```
sage: x, y = var('x y')
sage: desolve_rk4(x*y*(2-y), y, ics=[0,1], end_points=[0, 1],
....:             step=0.5)
[[0, 1], [0.5, 1.12419127425], [1.0, 1.46159016229]]
```

La fonction `desolve_rk4` utilise la méthode de Runge-Kutta d'ordre 4 (la même que `rk4` pour GSL) et prend comme paramètres :

- le membre droit de l'équation $y'(t) = f(t, y(t))$;
- le nom de la variable fonction inconnue ;
- les conditions initiales `ics` ;
- l'intervalle de résolution `end_points` ;
- le pas `step`.

Nous ne développons pas la commande similaire `desolve_system_rk4`, déjà évoquée au chapitre §4 et qui s'applique à un système d'équations différentielles. Maxima se limite aussi à la précision machine.

Si l'on recherche des solutions calculées en précision arbitrairement grande, il est possible de se tourner vers `odefun` du module `mpmath` :

```
sage: import mpmath
sage: sol = mpmath.odefun(lambda t, y: y, 0, 1)
sage: sol(1)
mpf('2.7182818284590451')
sage: mpmath.mp.prec=100
sage: sol(1)
mpf('2.7182818284590452353602874802307')
sage: N(exp(1), 100)
2.7182818284590452353602874714
```

Les arguments de la fonction `mpmath.odefun` sont :

- les membres droits du système d'équations, sous la forme d'une fonction $(t, y) \mapsto f(t, y(t))$ comme pour la fonction `ode_solver`. La dimension du système est déduite automatiquement de la dimension de la valeur de retour de la fonction ;
- les conditions initiales t_0 et $y(t_0)$.

Par exemple pour ce système de dimension 2

$$\begin{cases} y_1' &= -y_2 \\ y_2' &= y_1 \end{cases}$$

dont les solutions sont $(\cos(t), \sin(t))$:

```
sage: f = mpmath.odefun(lambda t, y: [-y[1], y[0]], 0, [1, 0])
sage: f(3)
[mpf('-0.98999249660044545727157279473154'),
 mpf('0.14112000805986722210074480280816')]
sage: (cos(3.), sin(3.))
(-0.989992496600445, 0.141120008059867)
```

La fonction `mpmath.odefun` utilise la méthode de Taylor. Pour un degré p on calcule :

$$y(t_{n+1}) = y(t_n) + h\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) + \frac{h^2}{2!}\frac{\mathrm{d}^2y}{\mathrm{d}t^2}(t_n) + \ldots + \frac{h^p}{p!}\frac{\mathrm{d}^py}{\mathrm{d}t^p}(t_n) + \mathcal{O}(h^{p+1}).$$

La principale question est celle du calcul des dérivés de y. Pour ce faire, `mpmath.odefun` calcule des valeurs approchées

$$[\widetilde{y}(t_n + h), \ldots, \widetilde{y}(t_n + ph)] \approx [y(t_n + h), \ldots, y(t_n + ph)]$$

par p pas de la méthode peu précise d'Euler. On calcule ensuite

$$\begin{aligned} \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n) &\approx \frac{\widetilde{y}(t_n + h) - \widetilde{y}(t_n)}{h} \\ \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n + h) &\approx \frac{\widetilde{y}(t_n + 2h) - \widetilde{y}(t_n + h)}{h} \end{aligned}$$

puis

$$\widetilde{\frac{\mathrm{d}^2y}{\mathrm{d}t^2}}(t_n) \approx \frac{\widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n + h) - \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n)}{h}$$

et ainsi de suite jusqu'à obtenir des estimations des dérivés de y en t_n jusqu'à l'ordre p.

Il faut être vigilant lorsque l'on change la précision de calcul flottant de `mpmath` comme cela est fait dans l'exemple ci-dessus, car on a changé la précision à laquelle seront calculées les évaluations de $y(t)$ mais le calcul des approximations des $(y^{(i)}(t_n))_i$ ne change pas et a été effectué dans la précision précédente. Pour illustrer ce problème, reprenons la résolution de l'équation différentielle $y' = y$ vérifiée par la fonction exp donnée plus haut :

```
sage: mpmath.mp.prec=10
sage: sol = mpmath.odefun(lambda t, y: y, 0, 1)
sage: sol(1)
mpf('2.7148')
sage: mpmath.mp.prec=100
sage: sol(1)
mpf('2.7135204235459511323824699502438')
```

L'approximation de exp(1) est très mauvaise, et pourtant elle est calculée avec 100 bits de précision ! La fonction solution `sol` (un « interpolant » dans le jargon `mpmath`) a été calculée avec seulement 10 bits de précision et ses coefficients ne sont pas recalculés en cas de changement de précision, ce qui explique le résultat.

7

Équations non linéaires

Ce chapitre explique comment *résoudre* une équation non linéaire avec Sage. Dans un premier temps, on décrit le fonctionnement de quelques méthodes classiques, qu'elles soient exactes ou approchées. Ensuite on indique les algorithmes implémentés dans Sage.

7.1 Équations algébriques

Par équation algébrique on entend une équation de la forme $p(x) = 0$ où p désigne un polynôme à une indéterminée dont les coefficients appartiennent à un anneau intègre A. On dit qu'un élément $\alpha \in A$ est une *racine* du polynôme p si $p(\alpha) = 0$.

Soit α un élément de A. La division euclidienne de p par $x - \alpha$ assure l'existence d'un polynôme constant r tel que :

$$p = (x - \alpha)q + r.$$

En évaluant cette équation en α, on obtient $r = p(\alpha)$. Donc le polynôme $x - \alpha$ divise p si, et seulement si, α est une racine de p. Ce point de vue permet d'introduire la notion de *multiplicité* d'une racine α du polynôme p : il s'agit du plus grand entier m tel que $(x - \alpha)^m$ divise p. On observe que la somme des multiplicités des racines de p est inférieur ou égal à n si p est de degré n.

Méthode `Polynomial.roots()`

Résoudre l'équation algébrique $p(x) = 0$ consiste à identifier les racines du polynôme p avec leurs multiplicités. La méthode `Polynomial.roots()` donne les racines d'un polynôme. Elle prend jusqu'à trois paramètres, tous optionnels. Le paramètre `ring` permet de préciser l'anneau où chercher les racines. Si on ne précise pas de valeur pour ce paramètre, il s'agira de l'anneau des coefficients du polynôme. Le booléen `multiplicities` indique la nature des informations renvoyées par `Polynomial.roots()` : chaque racine peut être accompagnée de sa multiplicité. Le paramètre `algorithm` sert à préciser quel algorithme utiliser.

```
sage: R.<x> = PolynomialRing(RealField(prec=10))
sage: p = 2*x^7 - 21*x^6 + 64*x^5 - 67*x^4 + 90*x^3 + 265*x^2 \
....: - 900*x + 375
sage: p.roots()
[(-1.7, 1), (0.50, 1), (1.7, 1), (5.0, 2)]
sage: p.roots(ring=ComplexField(10), multiplicities=False)
[-1.7, 0.50, 1.7, 5.0, -2.2*I, 2.2*I]
sage: p.roots(ring=RationalField())
[(1/2, 1), (5, 2)]
```

Représentation des nombres

Rappelons comment désigner les anneaux usuels avec Sage. Les entiers relatifs sont représentés par des objets de la classe `Integer` et, pour effectuer des conversions, on utilise le *parent* `ZZ` ou la fonction `IntegerRing()` qui renvoie l'objet `ZZ`. De la même façon, les nombres rationnels sont représentés par des objets de la classe `Rational` ; le parent commun à ces objets est l'objet `QQ` que renvoit la fonction `RationalField()`. Dans les deux cas Sage utilise la bibliothèque de calcul en précision arbitraire GMP. Sans rentrer dans le détail de la réalisation de cette bibliothèque, les entiers manipulés avec Sage sont de taille arbitraire, la seule limitation provenant de la quantité de mémoire disponible sur la machine sur laquelle est exécuté le logiciel.

Plusieurs représentations approchées des nombres réels sont disponibles. Il y a `RealField()` pour la représentation utilisant les nombres à virgule flottante avec une précision donnée et, en particulier, `RR` pour une précision de 53 bits. Mais il y a aussi `RDF` et la fonction `RealDoubleField()` pour les nombres machine double précision. Et encore `RIF` et `RealIntervalField()` avec lesquelles un nombre réel est représenté par un intervalle le contenant, les extrémités de cet intervalle étant des nombres à virgule flottante.

Les représentations analogues pour les nombres complexes se nomment : `CC`, `CDF` et `CIF`. Là aussi, à chaque objet est associé une fonction ; ce sont `ComplexField()`, `ComplexIntervalField()` et `ComplexDoubleField()`.

Les calculs effectués par `Polynomial.roots()` sont exacts ou approchés selon le type de représentation des coefficients du polynôme et une éventuelle

valeur du paramètre `ring` : par exemple avec `ZZ` ou `QQ`, les calculs sont exacts ; avec `RR` les calculs sont approchés. Dans la seconde partie de ce chapitre on précisera l'algorithme utilisé pour le calcul des racines et le rôle des paramètres `ring` et `algorithm` (cf. §7.2).

La résolution des équations algébriques est intimement liée à la notion de nombre. Le *corps de décomposition* du polynôme p (supposé non constant) est la plus petite extension du corps des coefficients de p dans laquelle p est un produit de polynômes de degré 1 ; on montre qu'une telle extension existe toujours. Avec Sage, on construit cette extension avec la méthode `Polynomial.root_field()`. On peut alors calculer avec les racines *implicites* contenues dans le corps de décomposition.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = x^4 + x^3 + x^2 + x + 1
sage: K.<alpha> = p.root_field()
sage: p.roots(ring=K, multiplicities=None)
[alpha, alpha^2, alpha^3, -alpha^3 - alpha^2 - alpha - 1]
sage: alpha^5
1
```

Théorème de d'Alembert

Le corps de décomposition du polynôme à coefficients réels $x^2 + 1$ n'est autre que le corps des nombres complexes. Il est remarquable que tout polynôme non constant à coefficients complexes possède au moins une racine complexe : c'est ce qu'affirme le *théorème de d'Alembert*. En conséquence tout polynôme complexe non constant est un produit de polynômes de degré 1.

Voyons comment la méthode `Polynomial.roots()` permet d'illustrer ce théorème. Dans l'exemple qui suit, on construit l'anneau des polynômes à coefficients réels (on se contente d'une représentation utilisant les nombres à virgule flottante avec une précision de 53 bits). Puis un polynôme de degré inférieur ou égal à 15 est choisi aléatoirement dans cet anneau. Enfin on additionne les multiplicités des racines complexes calculées avec la méthode `Polynomial.roots()` et on compare cette somme au degré du polynôme.

```
sage: R.<x> = PolynomialRing(RR, 'x')
sage: d = ZZ.random_element(1, 15)
sage: p = R.random_element(d)
sage: p.degree() == sum(r[1] for r in p.roots(CC))
True
```

Distribution des racines

On poursuit avec une curieuse illustration de la puissance de la méthode `Polynomial.roots()` : on trace tous les points du plan complexe dont l'affixe

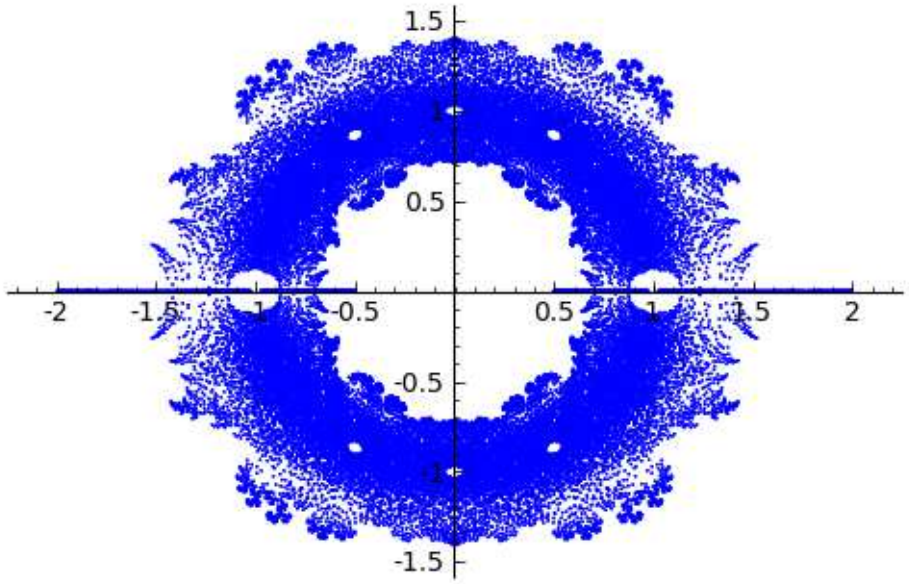


FIG. 7.1 – Distribution des racines.

est une racine d'un polynôme de degré 12 et à coefficients égaux à 1 ou -1. Le choix du degré est un compromis plus ou moins arbitraire ; il permet d'obtenir un tracé précis en peu de temps. L'utilisation de valeurs approchées pour les nombres complexes s'est également imposée pour des raisons de performance (cf. §5).

```
sage: def build_complex_roots(degree):
....:     R.<x> = PolynomialRing(CDF, 'x')
....:     v = []
....:     coeffs = [[-1, 1]] * (degree + 1)
....:     iter = cartesian_product_iterator(coeffs)
....:     for c in iter:
....:         v.extend(R(list(c)).roots(multiplicities=False))
....:     return v
....:
sage: data = build_complex_roots(12)
sage: g = plot(points(data, pointsize=1), aspect_ratio=1)
```

Résolution par radicaux

On a mentionné la possibilité dans certains cas de calculer les valeurs exactes des racines d'un polynôme. C'est par exemple possible lorsqu'on sait exprimer les racines en fonction des coefficients du polynôme et au moyen de radicaux (carrés, cubiques, etc). Dans ce cas on parle de *résolution par radicaux*.

Pour effectuer ce type de résolution avec Sage, il faut travailler avec les objets de la classe `Expression` qui représentent les expressions symboliques. On a vu que les entiers représentés par des objets de la classe `Integer` ont un même *parent*, l'objet `ZZ`. De la même façon, les objets de la classe `Expression` ont un même parent : il s'agit de l'objet `SR` (acronyme de *Symbolic Ring*) ; il offre notamment des possibilités de conversion vers la classe `Expression`.

Équations quadratiques

```
sage: a, b, c, x = var('a b c x')
sage: p = a * x^2 + b * x + c
sage: type(p)
<type 'sage.symbolic.expression.Expression'>
sage: p.parent()
Symbolic Ring
sage: p.roots(x)
[(-1/2*(b + sqrt(-4*a*c + b^2))/a, 1),
 (-1/2*(b - sqrt(-4*a*c + b^2))/a, 1)]
```

Degrés strictement supérieurs à 2

Il est possible de résoudre par radicaux les équations algébriques complexes de degré 3 et 4. En revanche il est impossible de résoudre par radicaux toutes les équations polynomiales de degré supérieur ou égal à 5. Cette impossibilité nous conduit à envisager des méthodes de résolution numérique (cf. §7.2).

```
sage: a, b, c, d, e, f, x = var('a b c d e f x')
sage: solve([a*x^5+b*x^4+c*x^3+d*x^2+e*x+f==0], x)
[0 == a*x^5 + b*x^4 + c*x^3 + d*x^2 + x*e + f]
```

Illustrons avec Sage une méthode de résolution des équations de degré 3. Pour commencer on montre que l'équation générale du troisième degré se ramène à la forme $x^3 + px + q = 0$.

```
sage: x, a, b, c, d = var('x a b c d')
sage: P = a * x^3 + b * x^2 + c * x + d
sage: alpha = var('alpha')
sage: P.subs(x=x + alpha).expand().coeff(x, 2)
3*a*alpha + b
sage: P.subs(x = x - b / (3 * a)).expand().collect(x)
a*x^3 - 1/3*b^2*x/a + c*x - 1/3*b*c/a + 2/27*b^3/a^2 + d
```

Pour obtenir les racines d'une équation de la forme $x^3 + px + q = 0$ on pose $x = u + v$.

```
sage: p, q, u, v = var('p q u v')
sage: P = x^3 + p * x + q
```

```
sage: P.subs(x = u + v).expand()
u^3 + 3*u^2*v + 3*u*v^2 + v^3 + p*u + p*v + q
```

On remarque que $u^3 + v^3 + q = 0$ implique $3uv + p = 0$, si bien que u^3 et v^3 sont les racines d'une équation du second degré.

```
sage: P.subs({x: u + v, q: -u^3 - v^3}).factor()
(u + v)*(3*u*v + p)
sage: P.subs({x: u + v, q: -u^3 - v^3, p: -3 * u * v})
0
sage: X = var('X')
sage: solve([X^2 + q * X - p^3/27 == 0], X, solution_dict=True)
[{X: -1/18*sqrt(4*p^3 + 27*q^2)*sqrt(3) - 1/2*q},
 {X: 1/18*sqrt(4*p^3 + 27*q^2)*sqrt(3) - 1/2*q}]
```

Les solutions de l'équation $x^3 + px + q = 0$ sont donc les sommes $u + v$ où u et v sont des racines cubiques de $-1/18\sqrt{4p^3 + 27q^2}\sqrt{3} - 1/2q$ et $1/18\sqrt{4p^3 + 27q^2}\sqrt{3} - 1/2q$ vérifiant $3uv + p = 0$. Bien entendu pour poursuivre, il faut discuter le signe de $\sqrt{4p^3 + 27q^2}$.

Méthode `Expression.roots()`

Les exemples précédents utilisent la méthode `Expression.roots()`. Cette méthode renvoie une liste de racines. Parmi les paramètres optionnels de cette méthode, on retrouve les paramètres `ring` et `multiplicities` déjà rencontrés avec la méthode `Polynomial.roots()`. Il est toutefois important de se rappeler que la méthode `Expression.roots()` ne s'applique pas uniquement à des expressions polynomiales.

Lorsque le paramètre `ring` n'est pas défini, la méthode `roots()` de la classe `Expression` délègue le calcul des racines au programme Maxima qui tente de factoriser l'expression puis effectue une résolution par radicaux sur chaque facteur de degré strictement inférieur à 5. Lorsque le paramètre `ring` est défini, l'expression est convertie en objet de la classe `Polynomial` dont les coefficients ont pour parent l'objet identifié par le paramètre `ring` ; ensuite le résultat de la méthode `Polynomial.roots()` est renvoyé. On décrit plus loin l'algorithme utilisé dans ce cas (cf. §7.2.2).

On verra plus loin des exemples de calculs avec des racines implicites, représentées notamment avec les objets `QQbar` et `AA` qui représentent des corps de nombres algébriques (cf. §9.1.5).

Élimination des racines multiples

Étant donné un polynôme p ayant des racines multiples, il est possible de construire un polynôme à racines simples, identiques à celles de p. Donc, lorsqu'on calcule les racines d'un polynôme, on peut toujours supposer que ces racines sont simples. Justifions l'existence du polynôme à racines simples

et voyons comment construire ce polynôme. Ceci nous permettra de donner une nouvelle illustration de la méthode `Expression.roots()`.

Soit α une racine du polynôme p dont la multiplicité est un entier m strictement supérieur à 1. C'est une racine du polynôme dérivé p' avec multiplicité $m-1$. En effet, si $p = (x-\alpha)^m\tilde{p}$ alors on a $p' = (x-\alpha)^{m-1}(m\tilde{p}+(x-\alpha)\tilde{p}')$.

```
sage: alpha, m, x = var('alpha m x')
sage: p = function('p', x)
sage: q = function('q', x)
sage: p = (x - alpha)^m * q
sage: p.derivative(x)
m*(-alpha + x)^(m - 1)*q(x) + (-alpha + x)^m*D[0](q)(x)
sage: simplify(p.derivative(x)(x=alpha))
0
```

En conséquence le pgcd de p et p' est le produit $\prod_{\alpha\in\Gamma}(x-\alpha)^{m_\alpha-1}$ avec Γ l'ensemble des racines de p de multiplicité strictement supérieure à 1 et m_α la multiplicité de la racine α. Si q désigne ce pgcd, alors le quotient de p par q a bien les propriétés attendues.

Notons que le degré du quotient de p par q est strictement inférieur au degré de p. En particulier, si ce degré est strictement inférieur à 5, il est possible d'exprimer les racines au moyen de radicaux. L'exemple qui suit en donne une illustration pour un polynôme à coefficients rationnels de degré 13.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = 128*x^13 - 1344*x^12 + 6048*x^11 - 15632*x^10 \
....: + 28056*x^9 - 44604*x^8 + 71198*x^7 - 98283*x^6 \
....: + 105840*x^5 - 101304*x^4 + 99468*x^3 - 81648*x^2 \
....: + 40824*x - 8748
sage: q = gcd(p, p.derivative())
sage: (p // q).degree()
4
sage: roots = SR(p // q).roots(multiplicities=False)
sage: roots
[1/2*I*2^(1/3)*sqrt(3) - 1/2*2^(1/3),
 -1/2*I*2^(1/3)*sqrt(3) - 1/2*2^(1/3),
 2^(1/3), 3/2]
sage: [QQbar(p(alpha)).is_zero() for alpha in roots]
[True, True, True, True]
```

7.2 Résolution numérique

En mathématiques, on oppose traditionnellement le *discret* et le *continu*. L'analyse numérique relativise en quelque sorte cette opposition : un des aspects majeurs de l'analyse numérique consiste en effet à aborder des questions

portant sur les nombres réels, par essence du domaine du continu, en adoptant un point de vue expérimental s'appuyant notamment sur l'ordinateur qui, lui, relève du discret.

S'agissant de la résolution d'équations non linéaires, de nombreuses questions se posent naturellement : combien de racines réelles, imaginaires, positives, négatives possède une équation donnée ?

Dans cette partie on commence par donner des éléments de réponse pour les équations algébriques. Puis on décrit quelques-unes des méthodes d'approximation qui permettent de calculer des valeurs approchées des solutions d'une équation non linéaire.

7.2.1 Localisation des solutions des équations algébriques

Règle de Descartes

La règle de Descartes s'énonce de la manière suivante : le nombre de racines positives d'un polynôme à coefficients réels est inférieur ou égal au nombre de changements de signe de la suite des coefficients du polynôme.

```
sage: R.<x> = PolynomialRing(RR, 'x')
sage: p = x^7 - 131/3*x^6 + 1070/3*x^5 - 2927/3*x^4 \
....: + 2435/3*x^3 - 806/3*x^2 + 3188/3*x - 680
sage: sign_changes = \
....: [p[i] * p[i + 1] < 0 \
....: for i in range(p.degree())].count(True)
sage: real_positive_roots = \
....: sum([alpha[1] \
....: if alpha[0] > 0 else 0 for alpha in p.roots()])
sage: sign_changes, real_positive_roots
(7, 5)
```

En effet, soient p un polynôme à coefficients réels de degré d et p' le polynôme dérivé. On note u et u' les suites des signes des coefficients des polynômes p et p' : on a $u_i = \pm 1$ selon que le coefficient de degré i de p est positif ou négatif. La suite u' se déduit de u par simple troncature : on a $u'_i = u_{i+1}$ pour $0 \leqslant i < d$. Il en résulte que le nombre de changements de signe de la suite u est au plus égal au nombre de changements de signe de la suite u' plus 1.

Par ailleurs, le nombre de racines positives de p est au plus égal au nombre de racines positives de p' plus un : un intervalle dont les extrémités sont des racines de p contient toujours une racine de p'.

Comme la règle de Descartes est vraie pour un polynôme de degré 1, les deux observations précédentes montrent qu'elle est encore vraie pour un polynôme de degré 2, etc.

Il est possible de préciser la relation entre le nombre de racines positives et le nombre de changements de signes de la suite des coefficients : la différence entre ces nombres est toujours paire.

Isolation de racines réelles de polynômes

On vient de voir qu'il est possible d'établir, pour les polynômes à coefficients réels, une majoration de nombre de racines contenues dans l'intervalle $[0, +\infty[$. Plus généralement, il existe des moyens de préciser le nombre de racines dans un intervalle donné.

Énonçons par exemple le théorème de Sturm. Soient un polynôme p à coefficients réels, d son degré et $[a, b]$ un intervalle. On construit une suite de polynômes par récurrence. Pour commencer $p_0 = p$ et $p_1 = p'$; ensuite, p_{i+2} est l'opposé du reste de la division euclidienne de p_i par p_{i+1}. En évaluant cette suite de polynômes aux points a et b on obtient deux suites réelles finies $(p_0(a), \ldots, p_d(a))$ et $(p_0(b), \ldots, p_d(b))$. Le théorème de Sturm s'énonce ainsi : le nombre de racines de p appartenant à l'intervalle $[a, b]$ est égal au nombre de changements de signe de la suite $(p_0(a), \ldots, p_d(a))$ diminué du nombre de changements de signes de la suite $(p_0(b), \ldots, p_d(b))$ en supposant que les racines de p sont simples, $p(a) \neq 0$ et $p(b) \neq 0$.

Montrons comment implémenter ce théorème avec Sage.

```
def count_sign_changes(l):
    changes = [l[i] * l[i + 1] < 0 for i in range(len(l) - 1)]
    return changes.count(True)

def sturm(p, a, b):
    assert p.degree() > 2
    assert not (p(a) == 0)
    assert not (p(b) == 0)
    if a > b:
        a, b = b, a
    remainders = [p, p.derivative()]
    for i in range(p.degree()):
        remainders.append(-(remainders[i] % remainders[i + 1]))
    evals = [[], []]
    for q in remainders:
        evals[0].append(q(a))
        evals[1].append(q(b))
    return count_sign_changes(evals[0]) \
           - count_sign_changes(evals[1])
```

Voici maintenant une illustration de cette fonction `sturm()`.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = (x - 34) * (x - 5) * (x - 3) * (x - 2) * (x - 2/3)
sage: sturm(p, 1, 4)
2
sage: sturm(p, 1, 10)
3
sage: sturm(p, 1, 200)
4
```

```
sage: p.roots(multiplicities=False)
[34, 5, 3, 2, 2/3]
sage: sturm(p, 1/2, 35)
5
```

7.2.2 Méthodes d'approximations successives

> *Approximation : Gén. au sing. Opération par laquelle on tend à se rapprocher de plus en plus de la valeur réelle d'une quantité ou d'une grandeur sans y parvenir rigoureusement.*
> Trésor de la Langue Française

Dans cette partie, on illustre différentes méthodes d'approximation des solutions d'une équation non linéaire $f(x) = 0$. Il y a essentiellement deux démarches pour calculer ces approximations. Nous verrons que l'algorithme le plus performant mêle les deux démarches.

La première démarche consiste à construire une suite d'intervalles emboîtés qui contiennent une solution de l'équation. On contrôle la précision et la convergence est assurée mais la vitesse de convergence n'est pas toujours bonne.

La seconde démarche suppose connue une valeur approchée d'une solution de l'équation. Si le comportement local de la fonction f est suffisamment régulier, on pourra calculer une nouvelle valeur approchée plus proche de la solution. Par récurrence, on obtient donc une suite de valeurs approchées. Cette démarche suppose donc connue une première approximation du nombre cherché. Par ailleurs, son efficacité repose sur le bon comportement local de la fonction f : *a priori*, on ne maîtrise pas la précision des valeurs approchées ; pire, la convergence de la suite de valeurs approchées n'est pas nécessairement garantie.

Dans toute cette partie, on considère une équation non linéaire $f(x) = 0$ où f désigne une fonction numérique définie sur un intervalle $[a, b]$ et continue sur cet intervalle. On suppose que les valeurs de f aux extrémités de l'intervalle $[a, b]$ sont non nulles et de signes opposés : autrement dit, le produit $f(a)f(b)$ est strictement négatif. La continuité de f assure donc l'existence dans l'intervalle $[a, b]$ d'au moins une solution à l'équation $f(x) = 0$.

Pour chaque méthode, on expérimentera avec la fonction suivante.

```
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: a, b = RR(-pi), RR(pi)
sage: bool(f(a) * f(b) < 0)
True
```

Il convient de noter qu'avec cet exemple la commande `solve` n'est d'aucune utilité.

```
sage: solve(f(x)==0, x)
```

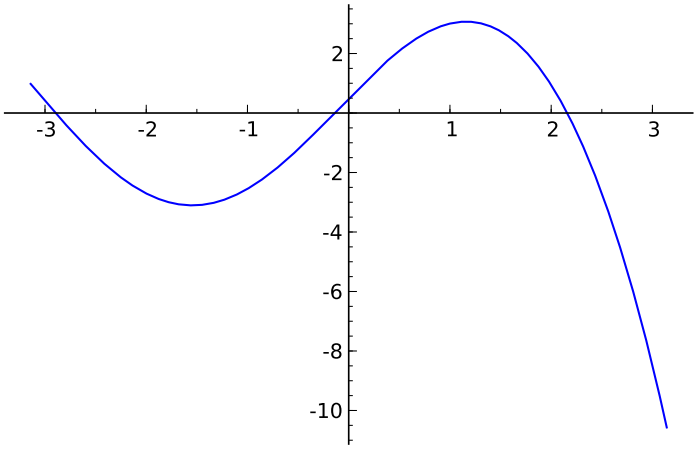


FIG. 7.2 – Courbe représentative de la fonction f.

```
[sin(x) == 1/8*e^x - 1/4]
sage: try:
....:     f.roots()
....: except RuntimeError:
....:     print('Root finding algorithm failed')
....:
Root finding algorithm failed
```

Les algorithmes de recherche de solutions d'équations non linéaires peuvent être coûteux : il convient de prendre quelques précautions avant d'en démarrer l'exécution. On s'assure notamment de l'existence de solutions en étudiant la continuité et la dérivabilité de la fonction à annuler ainsi que d'éventuels changements de signe ; à cet effet le tracé de graphe peut être utile (cf. §4).

Méthode de dichotomie

Cette méthode repose sur la première démarche : construire une suite d'intervalles emboîtés qui contiennent tous une solution de l'équation $f(x) = 0$.

On divise l'intervalle $[a, b]$ en son milieu, noté c. Supposons $f(c) \neq 0$. Soit $f(a)f(c)$ est strictement inférieur à zéro et l'intervalle $[a, c]$ contient nécessairement une solution de l'équation ; soit $f(c)f(b)$ est strictement supérieur à zéro et l'intervalle $[c, b]$ contient une solution de l'équation. Ainsi on sait construire un intervalle contenant une solution et dont la longueur est deux fois plus petite que celle de l'intervalle $[a, b]$. En répétant cette construction on obtient bien une suite d'intervalles aux propriétés attendues.

Pour mettre en œuvre cette démarche on définit la fonction Python `intervalgen`.

```
def intervalgen(f, phi, s, t):
    msg = 'Wrong arguments: f({0}) * f({1}) >= 0)'.format(s, t)
    assert (f(s) * f(t) < 0), msg
    yield s
    yield t
    while 1:
        u = phi(s, t)
        yield u
        if f(u) * f(s) < 0:
            t = u
        else:
            s = u

phi(s, t) = (s + t) / 2
```

La définition de cette fonction mérite quelques explications. La présence du mot clé `yield` dans la définition de `intervalgen` en fait un *générateur* (voir §12.2.4). Lors d'un appel à la méthode `next()` d'un générateur, si l'interpréteur rencontre le mot clé `yield`, toutes les données locales sont sauvegardées, l'exécution est interrompue et l'expression immédiatement à droite du mot clé rencontré est renvoyée. L'appel suivant à la méthode `next()` démarrera à l'instruction suivant le mot clé `yield` avec les données locales sauvegardées avant l'interruption. Utilisé dans une boucle infinie (`while True :`) le mot clé `yield` permet donc de programmer une suite récurrente avec une syntaxe proche de sa description mathématique. Il est possible de stopper définitivement l'exécution en utilisant l'habituel mot clé `return`.

Le paramètre `phi` représente une fonction ; elle caractérise la méthode d'approximation. Pour la méthode de la dichotomie, cette fonction calcule le milieu d'un intervalle. Pour tester une autre méthode d'approximations successives reposant également sur la construction d'intervalles emboîtés, on donnera une nouvelle définition de la fonction `phi` et on utilisera à nouveau la fonction `intervalgen` pour construire le générateur correspondant.

Les paramètres `s` et `t` de la fonction représentent les extrémités du premier intervalle. Un appel à `assert` permet de vérifier que la fonction f change de signe entre les extrémités de cet intervalle ; on a vu que cela garantit l'existence d'une solution.

Les deux premières valeurs du générateur correspondent aux paramètres `s` et `t`. La troisième valeur est le milieu de l'intervalle correspondant. Les paramètres `s` et `t` représentent ensuite les extrémités du dernier intervalle calculé. Après évaluation de f au milieu de cet intervalle, on change une des extrémités de l'intervalle en sorte que le nouvel intervalle contienne encore une solution. On convient de prendre pour valeur approchée de la solution cherchée le milieu du dernier intervalle calculé.

Expérimentons avec l'exemple choisi : suivent les trois approximations obtenues par la méthode de dichotomie appliquée sur l'intervalle $[-\pi, \pi]$.

```
sage: a, b
-3.14159265358979 3.14159265358979
sage: bisection = intervalgen(f, phi, a, b)
sage: bisection.next()
-3.14159265358979
sage: bisection.next()
3.14159265358979
sage: bisection.next()
0.00000000000000
```

Puisque nous nous apprêtons à comparer différentes méthodes d'approximation, il sera commode de disposer d'un mécanisme automatisant le calcul de la valeur approchée d'une solution de l'équation $f(x) = 0$ à partir des générateurs que nous définirons avec Sage pour chacune de ces méthodes. Ce mécanisme doit nous permettre de contrôler la précision du calcul et le nombre maximum d'itérations. C'est le rôle de la fonction `iterate` dont la définition suit.

```
from types import GeneratorType, FunctionType

def checklength(u, v, w, prec):
    return abs(v - u) < 2 * prec

def iterate(series, check=checklength, prec=10^-5, maxiter=100):
    assert isinstance(series, GeneratorType)
    assert isinstance(check, FunctionType)
    niter = 2
    v, w = series.next(), series.next()
    while (niter <= maxiter):
        niter += 1
        u, v, w = v, w, series.next()
        if check(u, v, w, prec):
            return 'After {0} iterations: {1}'.format(niter, w)
    msg = 'Failed after {0} iterations'.format(maxiter)
    raise RuntimeError, msg
```

Le paramètre `series` doit être un générateur. On conserve les trois dernières valeurs de ce générateur pour pouvoir effectuer un test de convergence. C'est le rôle du paramètre `check` : une fonction qui stoppera ou non les itérations. Par défaut la fonction `iterate` utilise la fonction `checklength` qui stoppe les itérations si le dernier intervalle calculé est de longueur strictement inférieure au double du paramètre `prec` ; cela garantit que la valeur calculée par la méthode de dichotomie est une valeur approchée avec une erreur strictement inférieure à `prec`.

Une exception est déclenchée dès que le nombre d'itérations dépasse le paramètre `maxiter`.

```
sage: bisection = intervalgen(f, phi, a, b)
sage: iterate(bisection)
After 20 iterations: 2.15847275559132
```

Exercice 20. Modifier la fonction `intervalgen` pour que le générateur stoppe si une des extrémités de l'intervalle est une solution.

Exercice 21. Utiliser les fonctions `intervalgen` et `iterate` pour programmer le calcul d'une valeur approchée d'une solution de l'équation $f(x) = 0$ à partir d'une suite d'intervalles emboîtés, chaque intervalle étant obtenu en divisant aléatoirement le précédent.

Méthode de la fausse position

Cette méthode repose encore sur la première démarche : construire une suite d'intervalles emboîtés qui contiennent tous une solution de l'équation $f(x) = 0$. Mais cette fois ci on utilise une interpolation linéaire de la fonction f pour diviser chaque intervalle.

Précisément, pour diviser l'intervalle $[a, b]$, on considère le segment joignant les deux points de la courbe représentative de f d'abscisses a et b. Comme $f(a)$ et $f(b)$ sont de signes opposés, ce segment coupe l'axe des abscisses : il divise donc l'intervalle $[a, b]$ en deux intervalles. Comme pour la méthode de dichotomie, on identifie un intervalle contenant une solution en calculant la valeur que prend la fonction f au point commun à ces deux intervalles.

La droite passant par des points de coordonnées $(a, f(a))$ et $(b, f(b))$ a pour équation :

$$y = \frac{f(b) - f(a)}{b - a}(x - a) + f(a). \tag{7.1}$$

Puisque $f(b) \neq f(a)$, cette droite coupe l'axe des abscisses au point d'abscisse :

$$a - f(a)\frac{b - a}{f(b) - f(a)}.$$

On peut donc tester cette méthode de la manière suivante.

```
sage: phi(s, t) = t - f(t) * (s - t) / (f(s) - f(t))
sage: falsepos = intervalgen(f, phi, a, b)
sage: iterate(falsepos)
After 8 iterations: -2.89603757331027
```

Il est important de remarquer que les suites construites avec les méthodes de dichotomie et de la fausse position ne convergent pas nécessairement vers les mêmes solutions. En réduisant l'intervalle d'étude on retrouve la solution positive obtenue avec la méthode de dichotomie.

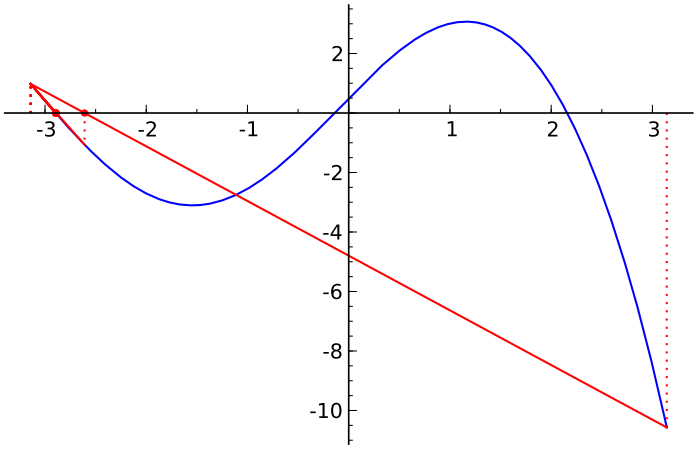


FIG. 7.3 – Méthode de la fausse position sur $[-\pi, \pi]$.

```
sage: a, b = RR(pi/2), RR(pi)
sage: phi(s, t) = t - f(t) * (s - t) / (f(s) - f(t))
sage: falsepos = intervalgen(f, phi, a, b)
sage: phi(s, t) = (s + t) / 2
sage: bisection = intervalgen(f, phi, a, b)
sage: iterate(falsepos)
After 15 iterations: 2.15846441170219
sage: iterate(bisection)
After 20 iterations: 2.15847275559132
```

Méthode de Newton

Comme la méthode de la fausse position, la méthode de Newton utilise une approximation linéaire de la fonction f. Du point de vue graphique, il s'agit de considérer une tangente à la courbe représentative de f comme approximation de cette courbe.

On suppose donc maintenant f dérivable et la fonction dérivée f' de signe constant dans l'intervalle $[a, b]$; ainsi f est monotone. On suppose aussi que f change de signe dans l'intervalle $[a, b]$. L'équation $f(x) = 0$ a donc une unique solution dans cet intervalle ; on note α ce nombre.

Soit $u_0 \in [a, b]$. La tangente à la courbe représentative de f au point d'abscisse u_0 a pour équation :

$$y = f'(u_0)(x - u_0) + f(u_0). \tag{7.2}$$

L'abscisse du point d'intersection de cette droite avec l'axe des abscisses est donc :

$$u_0 - f(u_0)/f'(u_0).$$

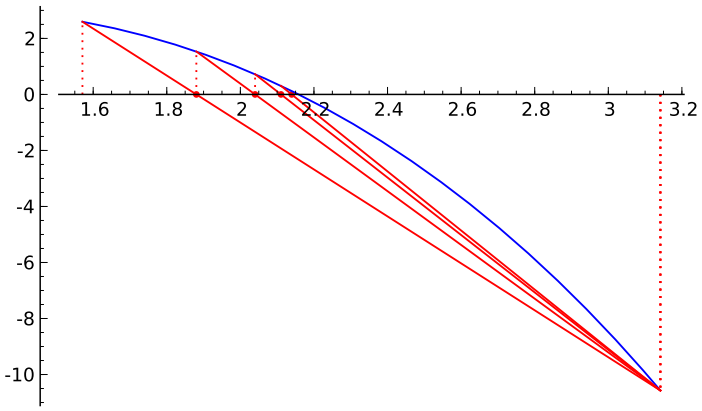


FIG. 7.4 – Méthode de la fausse position sur $[-\pi/2, \pi]$.

On note φ la fonction $x \mapsto x - f(x)/f'(x)$. Elle est définie à condition que f' ne s'annule pas dans l'intervalle $[a, b]$. On s'intéresse à la suite récurrente u définie par $u_{n+1} = \varphi(u_n)$.

Si la suite u est convergente[1], alors sa limite ℓ vérifie $\ell = \ell - f(\ell)/f'(\ell)$ dont résulte $f(\ell) = 0$: la limite est égale à α, la solution de l'équation $f(x) = 0$.

Pour que l'exemple respecte les hypothèses de monotonie, on est amené à réduire l'intervalle d'étude.

```
sage: f.derivative()
x |--> -1/2*e^x + 4*cos(x)
sage: a, b = RR(pi/2), RR(pi)
```

On définit un générateur Python `newtongen` représentant la suite récurrente que l'on vient de définir. Ensuite on définit un nouveau test de convergence `checkconv` qui stoppe les itérations si les deux derniers termes calculés sont suffisamment proches ; bien entendu ce test ne garantit pas la convergence de la suite des valeurs approchées.

```
def newtongen(f, u):
    while 1:
        yield u
        u -=  f(u) / (f.derivative()(u))

def checkconv(u, v, w, prec):
    return abs(w - v) / abs(w) <= prec
```

On peut maintenant tester la méthode de Newton sur notre exemple.

[1] Un théorème de L. Kantorovich donne une condition suffisante pour que la méthode de Newton converge.

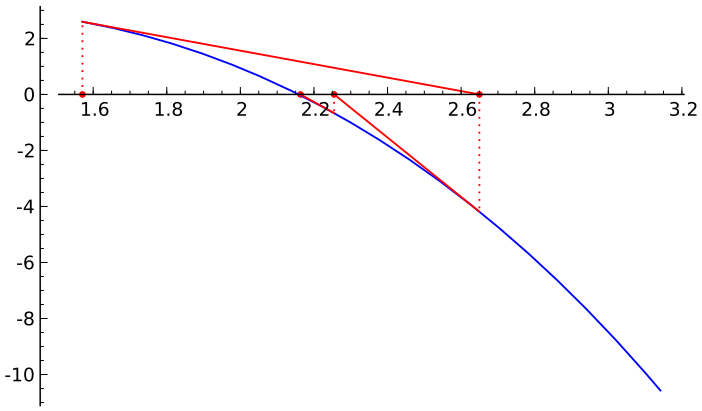


FIG. 7.5 – Méthode de Newton.

```
sage: iterate(newtongen(f, a), check=checkconv)
After 6 iterations: 2.15846852554764
```

Méthode de la sécante[2]

Dans la méthode de Newton le calcul de dérivée peut être coûteux. Il est possible de substituer à ce calcul de dérivée une interpolation linéaire : si on dispose de deux approximations de la solution, donc de deux points de la courbe représentative de f, et si la droite passant par ces deux points rencontre l'axe des abscisses, on considère l'abscisse du point d'intersection comme une nouvelle approximation. Pour démarrer la construction et lorsque les deux droites sont parallèles on convient de calculer une nouvelle approximation de la même façon que dans la méthode de Newton.

On reconnait donc le calcul effectué dans la méthode de la fausse position mais on n'identifie pas d'intervalle contenant une racine.

On définit un générateur Python mettant en œuvre cette méthode.

```
def secantgen(f, a):
    yield a

    estimate = f.derivative()(a)
    b = a - f(a) / estimate
    yield b

    while 1:
        fa, fb = f(a), f(b)
        if fa == fb:
```

[2]Note des auteurs : la différence entre la méthode de la fausse position et la méthode de la sécante sera mieux expliquée dans une version ultérieure du livre.

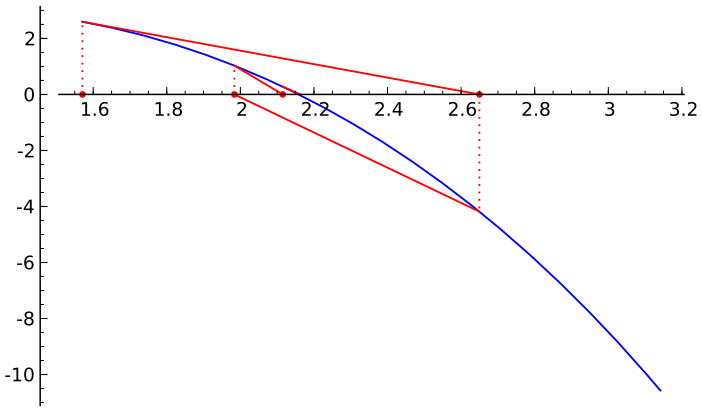


FIG. 7.6 – Méthode de la sécante.

```
        estimate = f.derivative()(a)
    else:
        estimate = (fb - fa) / (b - a)
    a = b
    b -= fb / estimate
    yield b
```

On peut maintenant tester la méthode de la sécante sur notre exemple.

```
sage: iterate(secantgen(f, a), check=checkconv)
After 8 iterations: 2.15846852557553
```

Méthode de Müller

Rien ne nous empêche d'étendre la méthode de la sécante en substituant à f des approximations polynomiales de degré quelconque. Par exemple la méthode de Müller utilise des approximations quadratiques.

Supposons construites trois approximations r, s et t de la solution de l'équation $f(x) = 0$. On considère le polynôme d'interpolation de Lagrange défini par les trois points de la courbe représentative de f d'abscisses r, s et t. C'est un polynôme du second degré. On convient de prendre pour nouvelle approximation la racine de ce polynôme qui est la plus proche de t. Par ailleurs, les trois premiers termes de la suite sont fixés de manière arbitraire : a, b puis $(a + b)/2$.

Il convient de remarquer que les racines des polynômes – et donc les approximations calculées – peuvent être des nombres complexes.

La programmation de cette méthode en Sage n'est pas difficile ; elle peut se faire sur le même modèle que la méthode de la sécante. Notre réalisation utilise toutefois une structure de donnée mieux adaptée à l'énumération des termes d'une suite récurrente.

```
from collections import deque

basering = PolynomialRing(CC, 'x')

def quadraticgen(f, r, s):
    t = (r + s) / 2
    yield t
    points = deque([(r, f(r)), (s, f(s)), (t, f(t))], maxlen=3)
    while 1:
        polynomial = basering.lagrange_polynomial(points)
        roots = polynomial.roots(ring=CC, multiplicities=False)
        u = min(roots, key=lambda x: abs(x - points[2][0]))
        points.append((u, f(u)))
        yield points[2][0]
```

Le module `collections` de la bibliothèque de référence Python implémente plusieurs structures de données. Dans `quadraticgen`, la classe `deque` est utilisée pour stocker les dernières approximations calculées. Un objet `deque` stocke des données dans la limite du nombre `maxlen` fixé lors de sa création ; ici le nombre maximal de données stockées est égal à l'ordre de récurrence de la suite des approximations. Lorsqu'un objet `deque` a atteint sa capacité maximale de stockage, la méthode `deque.append()` ajoute les nouvelles données sur le principe « premier entré, premier sorti ».

Notons que les itérations de cette méthode ne nécessitent pas le calcul de valeurs dérivées. De plus chaque itération ne nécessite qu'une évaluation de la fonction f.

```
sage: generator = quadraticgen(f, a, b)
sage: iterate(generator, check=checkconv)
After 4 iterations: 2.15846852554764
```

Retour aux polynômes

Revenons à la situation étudiée au début de ce chapitre. Il s'agissait de calculer les racines d'un polynôme à coefficients réels ; on notera P ce polynôme. Supposons P unitaire :

$$P = a_0 + a_1 x + \ldots + a_{d-1} x^{d-1} + x^d.$$

Il est facile de vérifier que P est le déterminant de la matrice *compagnon* :

$$A = \begin{pmatrix} 0 & 0 & 0 & \ldots & 0 & -a_0 \\ 1 & 0 & 0 & \ldots & 0 & -a_1 \\ 0 & 1 & 0 & \ldots & 0 & -a_2 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 1 & -a_{d-1} \end{pmatrix}.$$

En conséquence les racines du polynôme P sont les valeurs propres de la matrice A. Les méthodes du chapitre 5 s'appliquent donc.

On a vu que la méthode `Polynomial.roots()` prend jusqu'à trois paramètres, tous optionnels : `ring`, `multiplicities` et `algorithm`. Supposons qu'un objet Sage de la classe `Polynomial` est associé au nom `p` (donc `isinstance(p, 'Polynomial')` renvoit `True`). Alors l'algorithme utilisé par la commande `p.roots()` dépend des paramètres `ring` et `algorithm` ainsi que de l'anneau des coefficients du polynôme, c'est-à-dire `p.base_ring()`.

L'algorithme teste si `ring` et `p.base_ring()` représentent des nombres en virgule flottante. Le cas échéant, des valeurs approchées des racines sont calculées avec la bibliothèque NumPy si `p.base_ring()` est `RDF` ou `CDF`, ou avec la bibliothèque PARI autrement (le paramètre `algorithm` permet à l'utilisateur d'imposer son choix de bibliothèque pour ce calcul). En consultant le code source de NumPy on voit que la méthode d'approximation des racines utilisée par cette bibliothèque consiste à calculer les valeurs propres de la matrice compagnon.

La commande qui suit permet d'identifier les objets représentant des nombres en virgule flottante.

```
sage: rings = [ZZ, QQ, QQbar, RDF, RIF, RR, AA, CDF, CIF, CC]
sage: for ring in rings:
....: print("{0:50} {1}".format(ring, ring.is_exact()))
....:
Integer Ring                                       True
Rational Field                                     True
Algebraic Field                                    True
Real Double Field                                  False
Real Interval Field with 53 bits of precision      False
Real Field with 53 bits of precision               False
Algebraic Real Field                               True
Complex Double Field                               False
Complex Interval Field with 53 bits of precision   False
Complex Field with 53 bits of precision            False
```

Lorsque le paramètre `ring` vaut `AA`, ou `RIF`, et `p.base_ring()` vaut `ZZ`, `QQ` ou `AA`, l'algorithme appelle la fonction `real_roots()` du module `sage.rings.polynomial.real_roots`. Cette fonction convertit le polynôme dans la base de Bernstein, puis utilise l'algorithme de Casteljau (pour évaluer le polynôme exprimé dans la base de Bernstein) et la règle de Descartes (cf. §7.2.1) pour localiser les racines.

Lorsque le paramètre `ring` vaut `AA`, `QQbar`, ou `CIF`, et `p.base_ring()` vaut `ZZ`, `QQ`, `AA` ou représente des rationnels Gaussiens, l'algorithme délègue les calculs à NumPy et PARI dont les résultats sont convertis dans les anneaux attendus et avec un filtre pour ne conserver que les racines réelles dans le cas de `AA`.

On peut prendre connaissance de toutes les situations couvertes par la méthode `Polynomial.roots()` en consultant la documentation de cette méthode.

Vitesse de convergence

Considérons une suite numérique convergente u et notons ℓ sa limite. On dit que la vitesse de convergence de la suite u est *linéaire* s'il existe $K \in]0,1[$ tel que :

$$\lim_{n\to\infty} \frac{|u_{n+1}-\ell|}{|u_n-\ell|} = K.$$

La vitesse de convergence de la suite u est dite *quadratique* s'il existe $K > 0$ tel que :

$$\lim_{n\to\infty} \frac{|u_{n+1}-\ell|}{|u_n-\ell|^2} = K.$$

Revenons à la méthode de Newton. On a construit une suite récurrente u définie par $u_{n+1} = \varphi(u_n)$ avec φ la fonction $x \mapsto x - f(x)/f'(x)$. Sous l'hypothèse que f est deux fois dérivable, la formule de Taylor pour la fonction φ et x au voisinage de la racine α s'écrit :

$$\varphi(x) = \varphi(\alpha) + (x-\alpha)\varphi'(\alpha) + \frac{(x-\alpha)^2}{2}\varphi''(\alpha) + \mathcal{O}_\alpha((x-\alpha)^3).$$

Or $\varphi(\alpha) = \alpha$, $\varphi'(\alpha) = 0$ et $\varphi''(\alpha) = f''(\alpha)/f'(\alpha)$. En substituant dans la formule précédente et en revenant à la définition de la suite u, on obtient :

$$u_{n+1} - \alpha = \frac{(u_n-\alpha)^2}{2}\frac{f''(\alpha)}{f'(\alpha)} + \mathcal{O}_\infty((u_n-\alpha)^3).$$

Lorsque la méthode de Newton converge, la vitesse de convergence de la suite construite est donc quadratique.

Méthode `Expression.find_root()`

On s'intéresse maintenant à la situation la plus générale : le calcul d'une valeur approchée d'une solution d'une équation $f(x) = 0$. Avec Sage, ce calcul se fait avec la méthode `Expression.find_root()`.

Les paramètres de la méthode `Expression.find_root()` permettent de définir un intervalle où chercher une racine, la précision du calcul ou le nombre d'itérations. Le paramètre `full_output` permet d'obtenir des informations sur le calcul, notamment le nombre d'itérations et le nombre d'évaluations de la fonction.

```
sage: result = (f == 0).find_root(a, b, full_output=True)
sage: print(result[0], result[1].iterations)
2.1584685255476423, 9
```

En fait, la méthode `Expression.find_root()` n'implémente pas réellement d'algorithme de recherche de solutions d'équations : les calculs sont délégués au module SciPy.

La fonctionnalité de SciPy utilisée par Sage pour résoudre une équation implémente la méthode de Brent qui combine trois des méthodes vues précédemment : la méthode de dichotomie, la méthode de la sécante et l'interpolation quadratique. Les deux premières valeurs approchées sont les extrémités de l'intervalle où est cherchée la solution de l'équation. La valeur approchée suivante est obtenue par interpolation linéaire comme on l'a fait dans la méthode de la sécante. Avec les itérations suivantes la fonction est approchée par une interpolation quadratique et l'abscisse du point d'intersection de la courbe d'interpolation avec l'axe des abscisses est la nouvelle valeur approchée, à moins que cette abscisse ne soit pas comprise entre les deux précédentes valeurs approchées calculées auquel cas on poursuit avec la méthode de dichotomie.

La bibliothèque SciPy ne permet pas de calculer en précision arbitraire (à moins de se contenter de calculer avec des entiers) ; d'ailleurs le code source de la méthode `Expression.find_root()` commence par convertir les bornes en nombres machine double précision. À l'opposé, toutes les illustrations de méthodes de résolution d'équation que nous avons construites dans ce chapitre fonctionnent en précision arbitraire et même symbolique.

```
sage: a, b = pi/2, pi
sage: generator = newtongen(f, a)
sage: generator.next()
1/2*pi
sage: generator.next()
(1/2*pi, 1/2*pi-(e^(1/2*pi)-10)*e^(-1/2*pi))
```

Exercice 22. Écrire un générateur pour la méthode de Brent qui fonctionne en précision arbitraire.

Troisième partie

Algèbre et calcul formel

Dieu a créé les nombres entiers, tout le reste est fabriqué par l'homme.

Leopold Kronecker (1823 - 1891)

8 Corps finis et théorie des nombres

Ce chapitre décrit l'utilisation de Sage en théorie des nombres, pour manipuler des objets sur des anneaux ou corps finis (§8.1), pour tester la primalité (§8.2) ou factoriser un entier (§8.3) ; enfin nous discutons quelques applications (§8.4).

8.1 Anneaux et corps finis

Les anneaux et corps finis sont un objet fondamental en théorie des nombres, et en calcul symbolique en général. En effet, de nombreux algorithmes de calcul formel se ramènent à des calculs sur des corps finis, puis on exploite l'information obtenue via des techniques comme la remontée de Hensel ou la reconstruction par les restes chinois. Citons par exemple l'algorithme de Cantor-Zassenhaus pour la factorisation de polynôme univarié à coefficients entiers, qui commence par factoriser le polynôme donné sur un corps fini.

8.1.1 Anneau des entiers modulo n

En Sage, l'anneau $\mathbb{Z}/n\mathbb{Z}$ des entiers modulo n se définit à l'aide du constructeur `IntegerModRing` (ou plus simplement `Integers`). Tous les objets construits à partir de ce constructeur et leurs dérivés sont systématiquement réduits modulo n, et ont donc une forme canonique, c'est-à-dire que deux variables représentant la même valeur modulo n ont la même représentation interne. Dans certains cas bien particuliers, il est plus efficace de retarder les réductions modulo n, par exemple si on multiplie des matrices avec de tels coefficients ; on préférera alors travailler avec des entiers, et

effectuer les réductions modulo n « à la main » via `a % n`. Attention, le module n n'apparaît pas explicitement dans la valeur affichée :

```
sage: a=IntegerModRing(15)(3); b=IntegerModRing(17)(3);
      print a, b
3 3
sage: a == b
False
```

Une conséquence est que si l'on « copie-colle » des entiers modulo n, on perd l'information sur n. Étant donnée une variable contenant un entier modulo n, on retrouve l'information sur n via les méthodes `base_ring` ou `parent`, et la valeur de n via la méthode `characteristic` :

```
sage: R=a.base_ring(); R
Ring of integers modulo 15
sage: R.characteristic()
15
```

Les opérateurs de base (addition, soustraction, multiplication) sont surchargés pour les entiers modulo n, et appellent les fonctions correspondantes, de même que les entiers sont automatiquement convertis, dès lors qu'un des opérandes est un entier modulo n :

```
sage: print a+a, a-17, a*a+1, a^3
6 1 10 12
```

Quant à l'inversion $1/a \bmod n$ ou la division $b/a \bmod n$, Sage l'effectue quand elle est possible, sinon il renvoie une erreur `ZeroDivisionError`, i.e., quand a et n ont un pgcd non-trivial :

```
sage: 1/(a+1)
4
sage: 1/a
ZeroDivisionError: Inverse does not exist.
```

Pour obtenir la valeur de a — en tant qu'entier — à partir du résidu $a \bmod n$, on peut utiliser la méthode `lift` ou bien `ZZ` :

```
sage: z=lift(a); y=ZZ(a); print y, type(y), y==z
3 <type 'sage.rings.integer.Integer'> True
```

L'*ordre additif* de a modulo n est le plus petit entier $k > 0$ tel que $ka = 0 \bmod n$. Il vaut $k = n/g$ où $g = \text{pgcd}(a, n)$, et est donné par la méthode `additive_order` (on voit au passage qu'on peut aussi utiliser `Mod` ou `mod` pour définir les entiers modulo n) :

```
sage: [Mod(x,15).additive_order() for x in range(0,15)]
[1, 15, 15, 5, 15, 3, 5, 15, 15, 5, 3, 15, 5, 15, 15]
```

L'*ordre multiplicatif* de a modulo n, pour a premier avec n, est le plus petit entier $k > 0$ tel que $a^k = 1 \bmod n$. (Si a a un diviseur commun p avec n, alors $a^k \bmod n$ est un multiple de p quel que soit k.) Si cet ordre multiplicatif égale $\varphi(n)$, à savoir l'ordre du groupe multiplicatif modulo n, on dit que a est un *générateur* de ce groupe. Ainsi pour $n = 15$, il n'y a pas de générateur, puisque l'ordre maximal est $4 < 8 = \varphi(15)$:

```
sage: [[x,Mod(x,15).multiplicative_order()]
        for x in range(1,15) if gcd(x,15)==1]
[[1, 1], [2, 4], [4, 2], [7, 4], [8, 4], [11, 2], [13, 4], [14, 2]]
```

Voici un exemple avec $n = p$ premier, où 3 est générateur :

```
sage: p=10^20+39; mod(2,p).multiplicative_order()
50000000000000000019
sage: mod(3,p).multiplicative_order()
100000000000000000038
```

Une opération importante sur $\mathbb{Z}/n\mathbb{Z}$ est l'*exponentiation modulaire*, qui consiste à calculer $a^e \bmod n$. Le crypto-système RSA repose sur cette opération. Pour calculer efficacement $a^e \bmod n$, les algorithmes les plus efficaces nécessitent de l'ordre de $\log e$ multiplications ou carrés modulo n. Il est crucial de réduire systématiquement tous les calculs modulo n, au lieu de calculer d'abord a^e en tant qu'entier, comme le montre l'exemple suivant :

```
sage: n=3^100000; a=n-1; e=100
sage: %timeit (a^e) % n
5 loops, best of 3: 387 ms per loop
sage: %timeit power_mod(a,e,n)
125 loops, best of 3: 3.46 ms per loop
```

8.1.2 Corps finis

Les corps finis[1] en Sage se définissent à l'aide du constructeur `FiniteField` (ou plus simplement `GF`). On peut aussi bien construire les *corps premiers* `GF(p)` avec p premier que les corps composés `GF(q)` avec $q = p^k$, p premier et $k > 1$ un entier. Comme pour les anneaux, les objets créés dans un tel corps ont une forme canonique, par conséquent une réduction est effectuée à chaque opération. Les corps finis jouissent des mêmes propriétés que les anneaux (§8.1.1), avec en plus la possibilité d'inverser un élément non nul :

[1] En français, le corps fini à q éléments est noté usuellement $\mathbb{F}_q$, alors qu'en anglais on utilise plutôt GF(q). On utilise ici la notation française pour désigner le concept mathématique, et la notation anglaise pour désigner du code Sage.

```
sage: R = GF(17); [1/R(x) for x in range(1,17)]
[1, 9, 6, 13, 7, 3, 5, 15, 2, 12, 14, 10, 4, 11, 8, 16]
```

Un corps non premier $\mathbb{F}_{p^k}$ avec p premier et $k > 1$ est isomorphe à l'anneau quotient des polynômes de $\mathbb{F}_p[x]$ modulo un polynôme f unitaire et irréductible de degré k. Dans ce cas, Sage demande un nom pour le *générateur* du corps, c'est-à-dire la variable x :

```
sage: R = GF(9,name='x'); R
Finite Field in x of size 3^2
```

Ici, Sage a choisi automatiquement le polynôme f :

```
sage: R.polynomial()
x^2 + 2*x + 2
```

Les éléments du corps sont alors représentés par des polynômes $a_{k-1}x^{k-1} + \cdots + a_1x + a_0$, où les a_i sont des éléments de $\mathbb{F}_p$:

```
sage: [r for r in R]
[0, 2*x, x + 1, x + 2, 2, x, 2*x + 2, 2*x + 1, 1]
```

On peut aussi imposer à Sage le polynôme irréductible f :

```
sage: Q.<x> = PolynomialRing(GF(3))
sage: R2 = GF(9,name='x',modulus=x^2+1); R2
Finite Field in x of size 3^2
```

Attention cependant, car si les deux instances `R` et `R2` créées ci-dessus sont isomorphes à $\mathbb{F}_9$, l'isomorphisme n'est pas explicite :

```
sage: p = R(x+1); R2(p)
TypeError: unable to coerce from a finite field other than the
prime subfield
```

8.1.3 Reconstruction rationnelle

Le problème de la *reconstruction rationnelle* constitue une jolie application des calculs modulaires. Étant donné un résidu a modulo m, il s'agit de trouver un « petit » rationnel x/y tel que $x/y \equiv a \bmod m$. Si on sait qu'un tel petit rationnel existe, au lieu de calculer directement x/y en tant que rationnel, on calcule x/y modulo m, ce qui donne le résidu a, puis on retrouve x/y par reconstruction rationnelle. Cette seconde approche est souvent plus efficace, car on remplace des calculs rationnels — faisant intervenir de coûteux pgcds — par des calculs modulaires.

Lemme 1. *Soient $a, m \in \mathbb{N}$, avec $0 < a < m$. Il existe au plus une paire d'entiers $x, y \in \mathbb{Z}$ premiers entre eux tels que $x/y \equiv a \bmod m$ avec $0 < |x|, y \leqslant \sqrt{m/2}$.*

Il n'existe pas toujours de telle paire x, y, par exemple pour $a = 2$ et $m = 5$. L'algorithme de reconstruction rationnelle est basé sur l'algorithme de pgcd étendu. Le pgcd étendu de m et a calcule une suite d'entiers $a_i = \alpha_i m + \beta_i a$, où les entiers a_i décroissent, et les coefficients α_i, β_i croissent en valeur absolue. Il suffit donc de s'arrêter dès que $|a_i|, |\beta_i| < \sqrt{m/2}$, et la solution est alors $x/y = a_i/\beta_i$. Cet algorithme est disponible via la fonction `rational_reconstruction` de Sage, qui renvoie x/y lorsqu'une telle solution existe, et une erreur sinon :

```
sage: rational_reconstruction(411,1000)
-13/17
sage: rational_reconstruction(409,1000)
Traceback (most recent call last):
...
ValueError: Rational reconstruction of 409 (mod 1000) does not
exist.
```

Pour illustrer la reconstruction rationnelle, considérons le calcul du nombre harmonique $H_n = 1 + 1/2 + \cdots + 1/n$. Le calcul naïf avec des nombres rationnels est le suivant :

```
sage: def harmonic(n):
          return add([1/x for x in range(1,n+1)])
```

Or nous savons que H_n peut s'écrire sous la forme p_n/q_n avec p_n, q_n entiers, où q_n est le ppcm de $1, 2, \ldots, n$. On sait par ailleurs que $H_n \leqslant \log n + 1$, ce qui permet de borner p_n. On en déduit la fonction suivante qui détermine H_n par calcul modulaire et reconstruction rationnelle :

```
def harmonic_mod(n,m):
   return add([1/x % m for x in range(1,n+1)])
def harmonic2(n):
   q = lcm(range(1,n+1))
   pmax = RR(q*(log(n)+1))
   m = ZZ(2*pmax^2)
   m = ceil(m/q)*q + 1
   a = harmonic_mod(n,m)
   return rational_reconstruction(a,m)
```

La ligne `m = ZZ(2*pmax^2)` garantit que la reconstruction rationnelle va trouver $p \leqslant \sqrt{m/2}$, tandis que la ligne suivante garantit que m est premier avec $x = 1, 2, \ldots, n$, sinon $1/x \bmod n$ provoquerait une erreur.

```
sage: harmonic(100) == harmonic2(100)
True
```

Sur cet exemple, la fonction `harmonic2` n'est pas plus efficace que la fonction `harmonic`, mais elle illustre bien notre propos. Dans certains cas, il n'est pas nécessaire de connaître une borne rigoureuse sur x et y, une estimation « à la louche » suffit, car on peut vérifier facilement par ailleurs que x/y est la solution cherchée.

On peut généraliser la reconstruction rationnelle avec un numérateur x et un dénominateur y de tailles différentes (voir par exemple la section 5.10 du livre [vzGG03]).

8.1.4 Restes chinois

Une autre application utile des calculs modulaires est ce qu'on appelle communément les « restes chinois ». Étant donnés deux modules m et n premiers entre eux, soit x un entier inconnu tel que $x \equiv a \bmod m$ et $x \equiv b \bmod n$. Alors le *théorème des restes chinois* permet de reconstruire de façon unique la valeur de x modulo le produit mn. En effet, on déduit de $x \equiv a \bmod m$ que x s'écrit sous la forme $x = a + \lambda m$ avec $\lambda \in \mathbb{Z}$. En remplaçant cette valeur dans $x \equiv b \bmod n$, on obtient $\lambda \equiv \lambda_0 \bmod n$, où $\lambda_0 = (b-a)/m \bmod n$. Il en résulte $x = x_0 + \mu nm$, où $x_0 = a + \lambda_0 m$.

On a décrit ici la variante la plus simple des « restes chinois ». On peut également considérer le cas où m et n ne sont pas premiers entre eux, ou bien le cas de plusieurs moduli $m_1, m_2, \ldots, m_k$. La commande Sage pour trouver x_0 à partir de a, b, m, n est `crt(a,b,m,n)` :

```
sage: a=2; b=3; m=5; n=7; lambda0=(b-a)/m % n; a+lambda0*m
17
sage: crt(2,3,5,7)
17
```

Reprenons l'exemple du calcul de H_n. Calculons d'abord $H_n \bmod m_i$ pour $i = 1, 2, \ldots, k$, ensuite nous déduisons $H_n \bmod (m_1 \cdots m_k)$ par restes chinois, enfin nous retrouvons H_n par reconstruction rationnelle :

```
def harmonic3(n):
   q = lcm(range(1,n+1))
   pmax = RR(q*(log(n)+1))
   B = ZZ(2*pmax^2)
   m = 1; a = 0; p = 2^63
   while m<B:
      p = next_prime(p)
      b = harmonic_mod(n,p)
      a = crt(a,b,m,p)
```

```
      m = m*p
   return rational_reconstruction(a,m)
sage: harmonic(100) == harmonic3(100)
True
```

La fonction `crt` de Sage fonctionne aussi quand les moduli m et n ne sont pas premiers entre eux. Soit $g = \gcd(m, n)$, il y a une solution si et seulement si $a \bmod g \equiv b \bmod g$:

```
sage: crt(15,1,30,4)
45
sage: crt(15,2,30,4)
...
ValueError: No solution to crt problem since gcd(30,4) does not
divide 15-2
```

Une application plus complexe des restes chinois est présentée dans l'exercice 26.

8.2 Primalité

Tester si un entier est premier est une des opérations fondamentales d'un logiciel de calcul symbolique. Même si l'utilisateur ne s'en rend pas compte, de tels tests sont effectués plusieurs milliers de fois par seconde par le logiciel. Par exemple pour factoriser un polynôme de $\mathbb{Z}[x]$ on commence par le factoriser dans $\mathbb{F}_p[x]$ pour un nombre premier p, il faut donc trouver un tel p.

Deux grandes classes de tests de primalité existent. Les plus efficaces sont des tests de *pseudo-primalité*, et sont en général basés sur des variantes du petit théorème de Fermat, qui dit que si p est premier, alors tout entier $0 < a < p$ est un générateur du groupe multiplicatif $(\mathbb{Z}/p\mathbb{Z})^*$, donc $a^{p-1} \equiv 1 \bmod p$. On utilise en général une petite valeur de a $(2, 3, \ldots)$ pour accélérer le calcul de $a^{p-1} \bmod p$. Si $a^{p-1} \neq 1 \bmod p$, p n'est certainement pas premier. Si $a^{p-1} \equiv 1 \bmod p$, on ne peut rien conclure : on dit alors que p est pseudo-premier en base a. L'intuition est qu'un entier p qui est pseudo-premier pour plusieurs bases a de grandes chances d'être premier (voir cependant ci-dessous). Les tests de pseudo-primalité ont en commun que quand ils renvoient `False`, le nombre est certainement composé, par contre quand ils renvoient `True`, on ne peut rien conclure.

La seconde classe est constituée des tests de *vraie primalité*. Ces tests renvoient toujours une réponse correcte, mais peuvent être moins efficaces que les tests de pseudo-primalité, notamment pour les nombres qui sont pseudo-premiers en de nombreuses bases, et en particulier pour les nombres vraiment

premiers. De nombreux logiciels ne fournissent qu'un test de pseudo-primalité, voire pire le nom de la fonction correspondante (`isprime` par exemple) laisse croire à l'utilisateur que c'est un test de (vraie) primalité. Sage fournit deux fonctions distinctes : `is_pseudoprime` pour la pseudo-primalité, et `is_prime` pour la primalité :

```
sage: p=previous_prime(2^400)
sage: %timeit is_pseudoprime(p)
625 loops, best of 3: 1.07 ms per loop
sage: %timeit is_prime(p)
5 loops, best of 3: 485 ms per loop
```

Nous voyons sur cet exemple que le test de primalité est bien plus coûteux ; quand c'est possible, on préfèrera `is_pseudoprime`.

Certains algorithmes de primalité fournissent un *certificat*, qui peut être vérifié indépendamment, souvent de manière plus efficace que le test lui-même. Sage ne fournit pas de tel certificat dans la version actuelle, mais on peut en fabriquer un avec le théorème de Pocklington :

Théorème. *Soit $n > 1$ un entier tel que $n-1 = FR$, avec $F \geqslant \sqrt{n}$. Si pour tout facteur premier p de F, il existe a tel que $a^{n-1} \equiv 1 \bmod n$ et $a^{(n-1)/p} - 1$ est premier avec n, alors n est premier.*

Soit par exemple $n = 2^{31} - 1$. La factorisation de $n-1$ est $2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$. On peut prendre $F = 151 \cdot 331$; $a = 3$ convient pour les deux facteurs $p = 151$ et $p = 331$. Il suffit ensuite de prouver la primalité de 151 et 331. Ce test utilise de manière intensive l'exponentiation modulaire.

Les *nombres de Carmichael* sont des entiers composés qui sont pseudo-premiers dans toutes les bases. Le petit théorème de Fermat ne permet donc pas de les distinguer des nombres premiers, quel que soit le nombre de bases essayées. Le plus petit nombre de Carmichael est $561 = 3 \cdot 11 \cdot 17$. Un nombre de Carmichael a au moins trois facteurs premiers. En effet, supposons que $n = pq$ soit un nombre de Carmichael, avec p, q premiers, $p < q$; par définition des nombres de Carmichael, on a pour tout $1 \leqslant a < q$ l'égalité $a^{n-1} \equiv 1$ modulo n, et par suite modulo q, ce qui implique que $n-1$ est multiple de $q-1$. L'entier n est nécessairement de la forme $q + \lambda q(q-1)$, puisqu'il est multiple de q et $n-1$ multiple de $q-1$, ce qui est incompatible avec $n = pq$ puisque $p < q$. Si $n = pqr$, alors il suffit que $a^{n-1} \equiv 1 \bmod p$ — et de même pour q et r, puisque par restes chinois on aura alors $a^{n-1} \equiv 1 \bmod n$. Une condition suffisante est que $n-1$ soit multiple de $p-1$, $q-1$ et $r-1$:

```
sage: [560 % (x-1) for x in [3,11,17]]
[0, 0, 0]
```

Exercice 23. Écrire une fonction Sage comptant les nombres de Carmichael $pqr \leqslant n$, avec p, q, r premiers impairs distincts. Combien trouvez-vous

pour $n = 10^4, 10^5, 10^6, 10^7$? (Richard Pinch a compté 20138200 nombres de Carmichael inférieurs à 10^{21}.)

Enfin, pour itérer une opération sur des nombres premiers dans un intervalle, il vaut mieux utiliser la construction `prime_range`, qui construit une table via un crible, plutôt qu'une boucle avec `next_probable_prime` ou `next_prime` :

```
def count_primes1(n):
   return add([1 for p in range(n+1) if is_prime(p)])
def count_primes2(n):
   return add([1 for p in range(n+1) if is_pseudoprime(p)])
def count_primes3(n):
   s=0; p=2
   while p <= n: s+=1; p=next_prime(p)
   return s
def count_primes4(n):
   s=0; p=2
   while p <= n: s+=1; p=next_probable_prime(p)
   return s
def count_primes5(n):
   s=0
   for p in prime_range(n): s+=1
   return s
sage: %timeit count_primes1(10^5)
5 loops, best of 3: 674 ms per loop
sage: %timeit count_primes2(10^5)
5 loops, best of 3: 256 ms per loop
sage: %timeit count_primes3(10^5)
5 loops, best of 3: 49.2 ms per loop
sage: %timeit count_primes4(10^5)
5 loops, best of 3: 48.6 ms per loop
sage: %timeit count_primes5(10^5)
125 loops, best of 3: 2.67 ms per loop
```

8.3 Factorisation et logarithme discret

On dit qu'un entier a est un carré — ou résidu quadratique — modulo n s'il existe x, $0 \leqslant x < n$, tel que $a \equiv x^2 \bmod n$. Sinon, on dit que a est un non-résidu quadratique modulo n. Lorsque $n = p$ est premier, ce test peut se décider efficacement grâce au calcul du symbole de Jacobi de a et p, noté $(a|p)$, qui peut prendre les valeurs $\{-1, 0, 1\}$, où $(a|p) = 0$ signifie que a est multiple de p, et $(a|p) = 1$ (respectivement $(a|p) = -1$) signifie que a est (respectivement n'est pas) un carré modulo p. La complexité du calcul

du symbole de Jacobi $(a|n)$ est essentiellement la même que celle du calcul du pgcd de a et n, à savoir $O(M(\ell)\log\ell)$ où ℓ est la taille de n. Cependant toutes les implantations du symbole de Jacobi — voire du pgcd — n'ont pas cette complexité (`a.jacobi(n)` calcule $(a|n)$) :

```
sage: p=(2^42737+1)//3; a=3^42737
sage: %timeit a.gcd(p)
125 loops, best of 3: 4.3 ms per loop
sage: %timeit a.jacobi(p)
25 loops, best of 3: 26.1 ms per loop
```

Lorsque n est composé, trouver les solutions de $x^2 \equiv a \bmod n$ est aussi difficile que factoriser n. Toutefois le symbole de Jacobi, qui est relativement facile à calculer, donne une information partielle. En effet, si $(a|n) = -1$, il n'y a pas de solution, car une solution vérifie nécessairement $(a|p) = 1$ pour tous les facteurs premiers p de n, donc $(a|n) = 1$.

Le logarithme discret. Soit n un entier positif, g un *générateur* du groupe multiplicatif modulo n et a premier avec n, $0 < a < n$. Par définition du fait que g est un générateur, il existe un entier x tel que $g^x = a \bmod n$. Le problème du *logarithme discret* consiste à trouver un tel entier x. La méthode `log` permet de résoudre ce problème :

```
sage: p=10^10+19; a=mod(17,p); a.log(2)
6954104378
sage: mod(2,p)^6954104378
17
```

Les meilleurs algorithmes connus pour calculer un logarithme discret sont de même ordre de complexité — en fonction de la taille de n — que ceux pour factoriser n. Cependant l'implantation actuelle en Sage du logarithme discret est peu efficace :

```
sage: p=10^20+39; a=mod(17,p)
sage: time r=a.log(3)
CPU times: user 89.63 s, sys: 1.70 s, total: 91.33 s
```

Suites aliquotes. La *suite aliquote* associée à un entier positif n est la suite (s_k) définie par récurrence : $s_0 = n$ et $s_{k+1} = \sigma(s_k) - s_k$, où $\sigma(s_k)$ est la somme des diviseurs de s_k, i.e., s_{k+1} est la suite des diviseurs *propres* de s_k, c'est-à-dire sans s_k lui-même. On arrête l'itération lorsque $s_k = 1$ — alors s_{k-1} est premier — ou lorsque la suite (s_k) décrit un cycle. Par exemple en partant de $n = 30$ on obtient :

$$30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1.$$

Lorsque le cycle est de longueur un, on dit que l'entier correspondant est *parfait*, par exemple $6 = 1 + 2 + 3$ et $28 = 1 + 2 + 4 + 7 + 14$ sont parfaits. Lorsque le cycle est de longueur deux, on dit que les deux entiers en question sont *amicaux*, comme 220 et 284. Lorsque le cycle est de longueur trois ou plus, les entiers formant ce cycle sont dits *sociables*.

Exercice 24. Calculer la suite aliquote commençant par 840, afficher les 5 premiers et 5 derniers éléments, et tracer le graphe de $\log_{10} s_k$ en fonction de k. (On pourra utiliser la fonction `sigma`.)

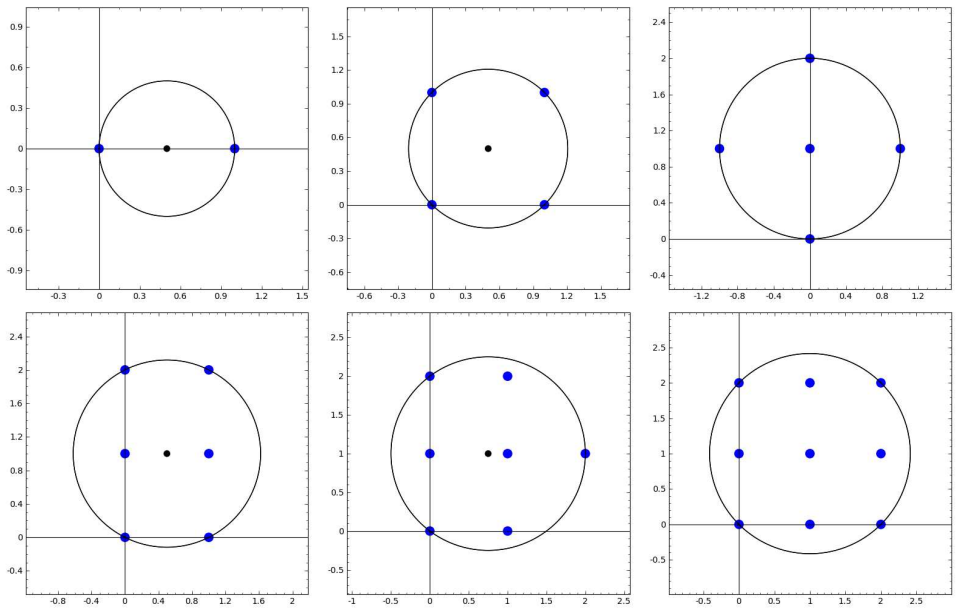
8.4 Applications

8.4.1 La constante δ

La constante δ est une généralisation en dimension deux de la constante γ d'Euler. Elle est définie comme suit :

$$\delta = \lim_{n\to\infty} \left(\sum_{k=2}^{n} \frac{1}{\pi r_k^2} - \log n \right), \tag{8.1}$$

où r_k est le rayon du plus petit disque du plan affine $\mathbb{R}^2$ contenant au moins k points de $\mathbb{Z}^2$. Par exemple $r_2 = 1/2$, $r_3 = r_4 = \sqrt{2}/2$, $r_5 = 1$, $r_6 = \sqrt{5}/2$, $r_7 = 5/4$, $r_8 = r_9 = \sqrt{2}$, ...



Exercice 25. 1. Écrire une fonction qui prend en entrée un entier positif k, et renvoie le rayon r_k et le centre (x, y) d'un plus petit disque contenant au moins k points de $\mathbb{Z}^2$.

2. Écrire une fonction dessinant le cercle correspondant à r_k avec les $m \geqslant k$ points de $\mathbb{Z}^2$ inclus.

3. En utilisant l'encadrement

$$\frac{\sqrt{\pi(k-6)+2}-\sqrt{2}}{\pi} < r_k < \sqrt{\frac{k-1}{\pi}}, \tag{8.2}$$

calculer une approximation de δ avec une erreur bornée par 0.3.

8.4.2 Calcul d'intégrale multiple via reconstruction rationnelle

Cette application est inspirée de l'article *Robust Mathematical Methods for Extremely Rare Events* de Bernard Beauzamy. Soient k et $n_1, n_2, \ldots, n_k$ des entiers positifs ou nuls. On veut calculer l'intégrale

$$I = \int_V x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k} \mathrm{d}x_1 \mathrm{d}x_2 \ldots \mathrm{d}x_k,$$

où le domaine d'intégration est défini par $V = \{x_1 \geqslant x_2 \geqslant \cdots \geqslant x_k \geqslant 0, x_1 + \cdots + x_k \leqslant 1\}$.

Exercice 26. Sachant que I est un nombre rationnel, mettre au point un algorithme utilisant la reconstruction rationnelle et/ou les restes chinois pour calculer I. On implantera cet algorithme en Sage et on l'appliquera au cas où $[n_1, \ldots, n_{31}] =$

$$[9, 7, 8, 11, 6, 3, 7, 6, 6, 4, 3, 4, 1, 2, 2, 1, 1, 1, 2, 0, 0, 0, 3, 0, 0, 0, 0, 1, 0, 0, 0].$$

9 Polynômes

Ce chapitre est consacré aux polynômes et aux objets apparentés, comme les fractions rationnelles et les séries formelles.

Nous avons vu au chapitre 2 comment effectuer des calculs sur des *expressions formelles*, éléments de « l'anneau symbolique » `SR`. Quelques-unes des méthodes applicables à ces expressions, par exemple `degree`, sont destinées aux polynômes :

```
sage: x = var('x'); p = (2*x+1)*(x+2)*(x^4-1)
sage: print p, "est de degré", p.degree(x)
(x + 2)*(2*x + 1)*(x^4 - 1) est de degré 6
```

Dans certains systèmes de calcul formel, dont Maple et Maxima, représenter les polynômes comme des expressions formelles particulières est la manière habituelle de les manipuler. À l'image d'Axiom, Magma ou Mupad, Sage permet aussi de traiter les polynômes de façon plus algébrique, et « sait calculer » dans des anneaux comme $\mathbb{Q}[x]$ ou $\mathbb{Z}/4\mathbb{Z}\,[x, y, z]$.

Ainsi, pour reproduire l'exemple précédent en travaillant dans un anneau de polynômes bien déterminé, on affecte à la variable Python `x` l'*indéterminée de l'anneau des polynômes en x à coefficients rationnels*, donnée par `polygen(QQ, 'x')`, au lieu de la *variable symbolique x* renvoyée[1] par `var('x')` :

```
sage: x = polygen(QQ, 'x'); p = (2*x+1)*(x+2)*(x^4-1)
sage: print p, "est de degré", p.degree()
2*x^6 + 5*x^5 + 2*x^4 - 2*x^2 - 5*x - 2 est de degré 6
```

[1] Une petite différence : alors que `var('x')` a le même effet que `x = var('x')` en utilisation interactive, `polygen(QQ, 'x')` sans affectation ne change pas la valeur de la variable Python `x`.

Observons que le polynôme est automatiquement développé. Les polynômes « algébriques » sont toujours représentés sous forme normale. C'est une différence cruciale par rapport aux polynômes de `SR`. En particulier, lorsque deux polynômes sont mathématiquement égaux, leur représentation informatique est la même, et une comparaison coefficient par coefficient suffit à tester l'égalité.

La première moitié du chapitre est consacrée aux objets à une indéterminée. Nous y explorons l'utilisation de Sage pour calculer dans les anneaux de polynômes $A[x]$, leurs quotients $A[x]/\langle P(x)\rangle$, les corps de fractions rationnelles $K(x)$ ou encore les anneaux de séries formelles $A[[x]]$. La seconde partie porte sur les polynômes à plusieurs indéterminées, et en particulier sur les systèmes polynomiaux — un vaste domaine dont ce livre n'aborde que quelques bases.

9.1 Polynômes à une indéterminée

9.1.1 Anneaux de polynômes

Construction. La première étape pour mener un calcul dans une structure algébrique R est souvent de construire R elle-même. On construit $\mathbb{Q}[x]$ par

```
sage: R = PolynomialRing(QQ, 'x')
sage: x = R.gen()
```

Le `'x'` qui apparaît sur la première ligne est une chaîne de caractères, le nom de l'indéterminée, ou *générateur* de l'anneau. Le `x` de la deuxième ligne est une variable Python dans laquelle on récupère le générateur ; employer le même nom[2] facilite la lecture du code. L'objet ainsi stocké dans la variable `x` représente le polynôme $x \in \mathbb{Q}[x]$. Il a pour parent (le *parent* d'un objet Sage est la structure algébrique « d'où il est issu », voir §1.7) l'anneau `QQ['x']` :

```
sage: x.parent()
Univariate Polynomial Ring in x over Rational Field
```

Le polynôme $x \in \mathbb{Q}[x]$ est considéré comme différent à la fois des polynômes identité $x \in A[x]$ d'anneau de base $A \neq \mathbb{Q}$ et de ceux comme $t \in \mathbb{Q}[t]$ dont l'indéterminée porte un autre nom.

L'expression `PolynomialRing(QQ, 't')` s'écrit aussi `QQ['t']`. En combinant cette abréviation avec la construction `S.<g> =`, qui affecte simultanément une structure à la variable `S` et son générateur à `g`, la construction de l'anneau $\mathbb{Q}[x]$ et de son indéterminée se réduit à `R.<x> = QQ['x']`, ou même simplement `R.<x> = QQ[]` en sous-entendant la variable `x`. La forme `x = polygen(QQ, 'x')` vue en introduction équivaut à

```
x = PolynomialRing(QQ, 'x').gen()
```

[2] Rappelons tout de même que techniquement, rien n'y oblige : les variables mathématiques, dont font partie les indéterminées de polynômes, sont d'une nature complètement différente des variables Python qui servent à la programmation.

Manipulation des anneaux de polynômes, $R = A[x]$	
construction (repr. dense)	`R.<x> = A[]` *ou* `R.<x> = PolynomialRing(A, 'x')`
ex. $\mathbb{Z}[x]$, $\mathbb{Q}[x]$, $\mathbb{R}[x]$, $\mathbb{Z}/n\mathbb{Z}[x]$	`ZZ['x']`, `QQ['x']`, `RR['x']`, `Integers(n)['x']`
construction (repr. creuse)	`R.<x> = PolynomialRing(A, 'x', sparse=True)`
accès à l'anneau de base A	`R.base_ring()`
accès à la variable x	`R.gen()` *ou* `R.0`
tests (intègre, nœthérien, ...)	`R.is_integral_domain()`, `R.is_noetherian()`, ...

Tableau 9.1 – Anneaux de polynômes.

Polynômes. Après l'instruction `R.<x> = QQ[]`, les expressions construites à partir de `x` et des constantes rationnelles par les opérations `+` et `*` sont des éléments de $\mathbb{Q}[x]$. Une autre façon de créer un polynôme consiste à énumérer ses coefficients :

```
sage: def rook_polynomial(n, var='x'):
....:     return ZZ[var]([binomial(n, k)^2 * factorial(k)
....:                                         for k in (0..n) ])
```

Ici, la conversion d'une liste d'entiers $[a_0, a_1, \dots]$ en un élément de `ZZ['x']` renvoie le polynôme $a_0 + a_1\,x + \cdots \in \mathbb{Z}[x]$. Les polynômes que construit cette fonction interviennent en combinatoire, leur nom anglais provient de l'interprétation du coefficient de x^k dans `rook_polynomial(n)` comme le nombre de façons de placer k tours sur l'échiquier $n \times n$ sans qu'elles se menacent.

Les éléments d'un anneau de polynômes sont représentés par des objets Python de la classe `Polynomial` ou de classes dérivées. Les principales opérations[3] disponibles sur ces objets sont résumées dans les tableaux 9.2 et 9.3. Ainsi, on récupère le degré d'un polyôme en appelant sa méthode `degree`. De même, `p.subs(a)` ou simplement `p(a)` donne la valeur de p au point a, mais sert aussi à calculer la composée $p \circ a$ lorsque a est lui-même un polynôme, et plus généralement à évaluer un polynôme de $A[x]$ en un élément d'une A-algèbre :

```
sage: p = R.random_element(degree=4); p  # un polynome au hasard
-4*x^4 - 52*x^3 - 1/6*x^2 - 4/23*x + 1
sage: p.subs(x^2)
-4*x^8 - 52*x^6 - 1/6*x^4 - 4/23*x^2 + 1
sage: p.subs(matrix([[1,2],[3,4]]))
[-375407/138  -273931/69]
[ -273931/46  -598600/69]
```

[3] Il y en a beaucoup d'autres. Ces tableaux omettent les fonctionnalités trop pointues, les variantes plus spécialisées de méthodes mentionnées, et de nombreuses méthodes communes à tous les « éléments d'anneaux », voire à tous les objets Sage, qui ne présentent pas d'intérêt particulier sur les polynômes. Notons cependant que les méthodes spécialisées (par exemple `p.rescale(a)`, équivalent à `p(a*x)`) sont souvent plus efficaces que les méthodes plus générales qui peuvent les remplacer.

La liste exacte des opérations disponibles, leur effet et leur efficacité dépendent fortement de l'anneau de base. Les polynômes de `ZZ['x']` possèdent une méthode `content` qui renvoie leur contenu, c'est-à-dire le pgcd de leurs coefficients ; ceux de `QQ['x']` non, l'opération étant triviale. La méthode `factor` existe quant à elle pour tous les polynômes mais déclenche une exception `NotImplementedError` pour un polynôme à coefficients dans `SR` ou dans $\mathbb{Z}/4\mathbb{Z}$. Cette exception signifie que l'opération n'est pas disponible dans Sage pour ce type d'objet — le plus souvent, bien qu'elle ait un sens mathématiquement.

Il est donc très utile de pouvoir jongler avec les différents anneaux de coefficients sur lesquels on peut considérer un « même » polynôme. Appliquée à un polynôme de $A[x]$, la méthode `change_ring` renvoie son image dans $B[x]$, quand il y a une façon naturelle de convertir les coefficients. La conversion est souvent donnée par un morphisme canonique de A dans B : notamment, `change_ring` sert à étendre l'anneau de base pour disposer de propriétés algébriques supplémentaires :

```
sage: x = polygen(ZZ); p = chebyshev_T(5, x); p
16*x^5 - 20*x^3 + 5*x
sage: p % (3*x^2)
x^5 + x^3 + 5*x
sage: p.change_ring(QQ) % (3*x^2)
5*x
```

Elle permet aussi de réduire un polynôme de $\mathbb{Z}[x]$ modulo un nombre premier :

```
sage: p.change_ring(GF(3))
x^5 + x^3 + 2*x
```

Inversement, si $B \subset A$ et si les coefficients de p sont en fait dans A, c'est aussi `change_ring` que l'on utilise pour ramener p dans $B[x]$.

Opérations sur les anneaux de polynômes. Les anneaux de polynômes sont des objets Sage à part entière. Le système « connaît » par exemple quelques propriétés algébriques de base des anneaux de polynômes :

```
sage: A = QQ['x']
sage: A.is_ring() and A.is_noetherian()  # A anneau nœthérien ?
True
sage: ZZ.is_subring(A)  # Z sous-anneau de A ?
True
sage: [n for n in range(20)                              # Z/nZ[x]
....:     if Integers(n)['x'].is_integral_domain()]  #   intègre ?
[0, 2, 3, 5, 7, 11, 13, 17, 19]
```

D'autres méthodes permettent de construire des polynômes remarquables, d'en tirer au hasard, ou encore d'énumérer des familles de polynômes, ici ceux de degré exactement 2 sur $\mathbb{F}_2$:

Accès aux données, opérations syntaxiques	
indéterminée x	`p.variable_name()`
coefficient de x^k	`p[k]`
coefficient dominant	`p.leading_coefficient()`
degré $\deg p$	`p.degree()`
liste des coefficients	`p.coeffs()`
coefficients *non nuls*	`p.coefficients()`
dictionnaire degré $\mapsto$ coefficient	`p.dict()`
tests (unitaire, constant, ...)	`p.is_monic()`, `p.is_constant()`, ...
Arithmétique de base	
opérations $p+q$, $p-q$, $p\times q$, p^k	`p + q`, `p - q`, `p * q`, `p^k`
substitution $x := a$	`p(a)` *ou* `p.subs(a)`
dérivée	`p.derivative()` *ou* `diff(p)`
Transformations	
changement d'anneau de base $A[x] \to B[x]$	`p.change_ring(B)` *ou* `B['x'](p)`
polynôme réciproque	`p.reverse()`

TABLEAU 9.2 – Opérations de base sur les polynômes $p, q \in A[x]$.

```
sage: list(GF(2)['x'].polynomials(of_degree=2))
[x^2, x^2 + 1, x^2 + x, x^2 + x + 1]
```

Nous nous limitons à ces quelques exemples et renvoyons à la documentation pour les autres opérations.

9.1.2 Représentation dense et représentation creuse

Un même objet mathématique — le polynôme p, à coefficients dans A — peut avoir différentes représentations informatiques. Si le résultat mathématique d'une opération sur p est bien sûr indépendant de la représentation, il en va autrement du comportement des objets Sage correspondants. Le choix de la représentation influe sur les opérations possibles, la forme exacte de leurs résultats, et particulièrement l'efficacité des calculs.

Il existe deux façons principales de représenter les polynômes. En représentation *dense*, les coefficients de $p = \sum_{i=0}^{n} p_i\, x^i$ sont stockés dans un tableau $[p_0, \ldots, p_n]$ indexé par les exposants. Une représentation *creuse* ne stocke que les coefficients non nuls : le polynôme est codé par un ensemble de paires exposant-coefficient (i, p_i), regroupées dans une liste, ou mieux, dans un dictionnaire indexé par les exposants (voir §3.2.9).

Pour des polynômes qui sont effectivement denses, c'est-à-dire dont la plupart des coefficients sont non nuls, la représentation dense occupe moins de mémoire et permet des calculs plus rapides. Elle économise le stockage des exposants et des structures de données internes du dictionnaire : ne reste que le strict nécessaire, les coefficients. De plus, l'accès à un élément ou l'itération

sur les éléments sont plus rapides dans un tableau que dans un dictionnaire. Inversement, la représentation creuse permet de calculer efficacement sur des polynômes qui ne tiendraient même pas en mémoire en représentation dense :

```
sage: R = PolynomialRing(ZZ, 'x', sparse=True)
sage: p = R.cyclotomic_polynomial(2^50); p, p.derivative()
(x^562949953421312 + 1, 562949953421312*x^562949953421311)
```

En Sage, la représentation est une caractéristique de l'anneau de polynômes, que l'on choisit à sa construction. Le polynôme « dense » $x \in \mathbb{Q}[x]$ et le polynôme « creux » $x \in \mathbb{Q}[x]$ n'ont donc pas le même[4] parent. La représentation par défaut des polynômes à une indéterminée est dense. L'option `sparse=True` de `PolynomialRing` sert à construire un anneau de polynômes creux.

De plus, les détails de représentation dépassant la distinction dense-creux ainsi que le code utilisé pour les calculs varient suivant la nature des coefficients des polynômes. Sage offre en effet, outre une implémentation *générique* qui fonctionne sur tout anneau commutatif, plusieurs implémentations des polynômes optimisées pour un type particulier de coefficients. Celles-ci apportent quelques fonctionnalités supplémentaires, et surtout sont considérablement plus efficaces que la version générique. Elles s'appuient pour cela sur des bibliothèques externes spécialisées, par exemple FLINT ou NTL dans le cas de $\mathbb{Z}[x]$.

Pour mener à bien de gros calculs, il est important de travailler autant que possible dans des anneaux de polynômes disposant d'implémentations efficaces. La page d'aide affichée par `p?` pour un polynôme `p` indique quelle implémentation il utilise. Le choix de l'implémentation découle le plus souvent de ceux de l'anneau de base et de la représentation. L'option `implementation` de `PolynomialRing` permet de préciser une implémentation quand il reste plusieurs possibilités.

Remarquons finalement que les expressions symboliques décrites dans les chapitres 1 et 2 (c'est-à-dire les éléments de `SR`) fournissent une troisième représentation des polynômes. Elles constituent un choix naturel quand un calcul mêle polynômes et expressions plus complexes, comme c'est souvent le cas en analyse. Mais la souplesse de représentation qu'elles offrent est parfois utile même dans un contexte plus algébrique. Par exemple, le polynôme $(x+1)^{10^{10}}$, une fois développé, est dense, mais il n'est pas nécessaire (ni souhaitable !) de le développer pour le dériver ou l'évaluer numériquement. Attention cependant : contrairement aux polynômes algébriques, les polynômes symboliques ne sont pas rattachés à un anneau de coefficients particulier et ne sont pas manipulés sous forme canonique. Un même polynôme a un grand

[4] Pourtant, `QQ['x'] == PolynomialRing(QQ, 'x', sparse=True)` renvoie vrai : les deux parents sont *égaux*, car ils représentent le même objet mathématique. Naturellement, le test correspondant avec `is` renvoie faux.

Divisibilité et division euclidienne	
test de divisibilité $p \mid q$	`p.divides(q)`
multiplicité d'un diviseur $q^k \mid p$	`k = p.valuation(q)`
division euclidienne $p = qd + r$	`q, r = p.quo_rem(d)`, `q = p//d`, `r = p%d`
pseudo-division $a^k p = qd + r$	`q, r, k = p.pseudo_divrem(d)`
plus grand commun diviseur	`p.gcd(q)`, `gcd([p1, p2, p3])`
plus petit commun multiple	`p.lcm(q)`, `lcm([p1, p2, p3])`
pgcd étendu $g = up + vq$	`g, u, v = p.xgcd(q)` *ou* `xgcd(p, q)`
Divers	
interpolation $p(x_i) = y_i$	`p = R.lagrange_polynomial([(x1,y1), ...])`
contenu de $p \in \mathbb{Z}[x]$	`p.content()`

Tableau 9.3 – Arithmétique des polynômes.

nombre d'écritures différentes, entre lesquelles c'est à l'utilisateur d'expliciter les conversions nécessaires. Dans le même ordre d'idées, le domaine `SR` regroupe toutes les expressions symboliques, sans distinction entre les polynômes et les autres, mais on peut tester explicitement si une expression symbolique `f` est polynomiale en une variable `x` par `f.is_polynomial(x)`.

9.1.3 Arithmétique euclidienne

Après la somme et le produit, les opérations les plus élémentaires sur les polynômes sont la division euclidienne et le calcul de plus grand commun diviseur. Les opérateurs et méthodes correspondants sont analogues à ceux sur les entiers (voir tableau 9.3).

La division euclidienne fonctionne sur un corps, et plus généralement sur un anneau commutatif, lorsque le coefficient dominant du diviseur est inversible, puisque ce coefficient est le seul élément de l'anneau de base par lequel il est nécessaire de diviser lors du calcul :

```
sage: R.<t> = Integers(42)[]; (t^20-1) % (t^5+8*t+7)
22*t^4 + 14*t^3 + 14*t + 6
```

Pour la même raison, si le coefficient dominant n'est pas inversible, on peut encore définir une *pseudo-division euclidienne* : soient A un anneau commutatif, $p, d \in A[x]$, et a le coefficient dominant de d. Alors il existe deux polynômes $q, r \in A[x]$, avec $\deg r < \deg d$, et un entier $k \leqslant \deg p - \deg d + 1$ tels que

$$a^k p = qd + r.$$

La pseudo-division euclidienne est donnée par la méthode `pseudo_divrem`.

Pour effectuer une division exacte, on utilise également l'opérateur de quotient euclidien `//`. En effet, diviser par un polynôme non constant avec `/` renvoie un résultat de type fraction rationnelle (voir §9.1.7), ou échoue si ce n'est pas possible :

```
sage: ((t^2+t)//t).parent()
Univariate Polynomial Ring in t over Ring of integers modulo 42
sage: (t^2+t)/t
Traceback (most recent call last):
...
TypeError: self must be an integral domain.
```

Exercice 27. Sage représente les polynômes de $\mathbb{Q}[x]$ sur la base monomiale $(x^n)_{n\in\mathbb{N}}$. Les polynômes de Tchebycheff T_n, définis par $T_n(\cos\theta) = \cos(n\theta)$, constituent une famille de polynômes orthogonaux et donc une base de $\mathbb{Q}[x]$. Les premiers polynômes de Tchebycheff sont

```
sage: x = polygen(QQ); [chebyshev_T(n, x) for n in (0..4)]
[1, x, 2*x^2 - 1, 4*x^3 - 3*x, 8*x^4 - 8*x^2 + 1]
```

Écrire une fonction qui prend en entrée un élément de $\mathbb{Q}[x]$ et renvoie les coefficients de sa décomposition sur la base $(T_n)_{n\in\mathbb{N}}$.

Exercice 28 (Division suivant les puissances croissantes)**.** Soient $n \in \mathbb{N}$ et $u, v \in A[x]$, avec $v(0)$ inversible. Alors il existe un unique couple (q, r) de polynômes de $A[x]$ tel que $u = qv + x^{n+1}r$. Écrire une fonction qui calcule q et r par un analogue de l'algorithme de division euclidienne. Comment faire ce même calcul le plus simplement possible, à l'aide de fonctions existantes ?

De même, Sage sait calculer le pgcd de polynômes sur un corps, grâce à la structure euclidienne, mais aussi sur certains autres anneaux, dont les entiers :

```
sage: S.<x> = ZZ[]; p = 2*(x^10-1)*(x^8-1); p.gcd(p.derivative())
2*x^2 - 2
```

Le *pgcd étendu* (en anglais *extended gcd*), c'est-à-dire le calcul d'une relation de Bézout

$$g = \mathrm{pgcd}(p, q) = ap + bq, \qquad g, p, q, a, b \in K[x]$$

est fourni par `u.xgcd(v)` :

```
sage: R.<x> = QQ[]; p = (x^5-1); q = (x^3-1)
sage: print "le pgcd est %s = (%s)*p + (%s)*q" % p.xgcd(q)
le pgcd est x - 1 = (-x)*p + (x^3 + 1)*q
```

(La méthode `xgcd` existe aussi pour les polynômes de `ZZ['x']`, mais attention : l'anneau $\mathbb{Z}[x]$ n'étant pas factoriel, son résultat n'est pas en général une relation de Bézout !)

En plus de la division euclidienne usuelle et de l'algorithme d'Euclide, on dispose pour ces opérations d'algorithmes rapides (division par la méthode de Newton, « demi-pgcd »), de complexité quasi-linéaire en les degrés des polynômes en jeu. Sage les utilise dans certaines circonstances. Un calcul de pgcd prend cependant beaucoup plus de temps qu'une multiplication ou une division, et éviter les calculs de pgcd superflus accélère considérablement certains « gros » calculs sur les polynômes.

9.1.4 Polynômes irréductibles et factorisation

Factorisation. La similitude algorithmique entre certains anneaux de polynômes et celui des entiers s'arrête avec la décomposition en produit de facteurs irréductibles, ou factorisation. Alors que décomposer un entier en facteurs premiers est un problème difficile, comme on l'a vu au chapitre précédent, factoriser un polynôme de degré 1000 sur $\mathbb{Q}$ ou $\mathbb{F}_p$ prend quelques secondes[5].

```
sage: p = ZZ['x'].random_element(degree=1000)
sage: %timeit p.factor()
5 loops, best of 3: 4.63 s per loop
```

La factorisation a lieu sur l'anneau de base du polynôme. Sage permet de factoriser sur des anneaux variés :

```
sage: x = polygen(ZZ); p = 54*x^4+36*x^3-102*x^2-72*x-12
sage: p.factor()
2 * 3 * (3*x + 1)^2 * (x^2 - 2)

sage: for A in [QQ, ComplexField(16), GF(5), QQ[sqrt(2)]]:
....:     print A, ":"; print A['x'](p).factor()
Rational Field :
(54) * (x + 1/3)^2 * (x^2 - 2)
Complex Field with 16 bits of precision :
(54.00) * (x - 1.414) * (x + 0.3333)^2 * (x + 1.414)
Finite Field of size 5 :
(4) * (x + 2)^2 * (x^2 + 3)
Number Field in sqrt2 with defining polynomial x^2 - 2 :
(54) * (x - sqrt2) * (x + sqrt2) * (x + 1/3)^2
```

Le résultat d'une décomposition en facteurs irréductibles n'est pas un polynôme (puisque les polynômes sont toujours sous forme normale, c'est-à-dire développés !), mais un objet `f` de type `Factorization`. On peut récupérer le `i`-ème facteur avec `f[i]`, et on retrouve le polynôme par `f.expand()`. Les objets `Factorization` disposent aussi de méthodes comme `gcd` et `lcm` qui ont le même sens que pour les polynômes mais opèrent sur les formes factorisées.

Irréductibilité. Le simple test d'irréductibilité demeure plus facile que la factorisation. Par exemple, un polynôme $p \in \mathbb{F}_q[x]$ de degré n est irréductible si et seulement s'il divise $x^{q^n} - x$ et est premier avec les $x^{q^{n/d}} - x$ où d parcourt les diviseurs premiers de n.

Vérifions que les cent premiers polynômes cyclotomiques Φ_n (Φ_n est le produit des $x-\zeta$ pour ζ racine n-ième primitive de l'unité) sont irréductibles :

[5] D'un point de vue théorique, on sait factoriser dans $\mathbb{Q}[x]$ en temps polynomial, et dans $\mathbb{F}_p[x]$ en temps polynomial probabiliste, alors que l'on ignore s'il est possible de factoriser les entiers en temps polynomial.

```
sage: all(QQ['x'].cyclotomic_polynomial(j).is_irreducible()
....:     for j in sxrange(1,100))
True
```

Exercice 29. Expliquer les résultats suivants :

```
sage: p = ZZ['x']([ZZ.random_element(10) * (1 - n%2)
....:              for n in range(42)])
sage: p.is_irreducible()
True
sage: p.derivative().is_irreducible()
False
```

Décomposition sans carré. Malgré sa bonne complexité théorique et pratique, la factorisation complète d'un polynôme est une opération complexe. La décomposition sans carré constitue une forme plus faible de factorisation, beaucoup plus facile à obtenir — quelques calculs de pgcd suffisent — et qui apporte déjà beaucoup d'information.

Soit $p = \prod_{i=1}^{r} p_i^{m_i} \in K[x]$ un polynôme décomposé en produit de facteurs irréductibles sur un corps K de caractéristique nulle. On dit que p est sans carré (en anglais *squarefree*) si tous les p_i sont de multiplicité $m_i = 1$, c'est-à-dire si les racines de p dans une clôture algébrique de K sont simples. Une *décomposition sans carré* est une factorisation en produit de facteurs sans carré deux à deux premiers entre eux :

$$p = f_1 f_2^2 \dots f_s^s \qquad \text{où} \qquad f_m = \prod_{m_i = m} p_i.$$

La décomposition sans carré sépare donc les facteurs irréductibles de p par multiplicité. En particulier, la *partie sans carré* $f_1 \dots f_s = p_1 \dots p_r$ de p est le polynôme à racines simples qui a les même racines que p aux multiplicités près.

9.1.5 Racines

Recherche de racines. Le calcul des racines d'un polynôme admet de nombreuses variantes, suivant que l'on cherche des racines réelles, complexes, ou dans un autre domaine, exactes ou approchées, avec ou sans multiplicités, de façon garantie ou heuristique... La méthode `roots` d'un polynôme renvoie par défaut les racines du polynôme dans son anneau de base, sous la forme d'une liste de couples (racine, multiplicité) :

```
sage: R.<x> = ZZ[]; p = (2*x^2-5*x+2)^2 * (x^4-7); p.roots()
[(2, 2)]
```

Avec un paramètre, `roots(D)` renvoie les racines dans le domaine D, ici les racines rationnelles, puis des approximations des racines ℓ-adiques pour $\ell = 19$:

Factorisation	
test d'irréductibilité	`p.is_irreducible()`
factorisation	`p.factor()`
factorisation sans carré	`p.squarefree_decomposition()`
partie sans carré $p/\operatorname{pgcd}(p,p')$	`p.radical()`

Racines	
racines dans A, dans D	`p.roots()` ; `p.roots(D)`
racines réelles	`p.roots(RR)` ; `p.real_roots()`
racines complexes	`p.roots(CC)` ; `p.complex_roots()`
isolation des racines réelles	`p.roots(RIF)` ; `p.real_root_intervals()`
isolation des racines complexes	`p.roots(CIF)`
résultant	`p.resultant(q)`
discriminant	`p.discriminant()`
groupe de Galois (p irréductible)	`p.galois_group()`

Tableau 9.4 – Factorisation et racines.

```
sage: p.roots(QQ)
[(2, 2), (1/2, 2)]
sage: p.roots(Zp(19, print_max_terms=3))
[(2 + 6*19^10 + 9*19^11 + ... + O(19^20), 1),
(7 + 16*19 + 17*19^2 + ... + O(19^20), 1),
(10 + 9*19 + 9*19^2 + ... + O(19^20), 1),
(10 + 9*19 + 9*19^2 + ... + O(19^20), 1),
(12 + 2*19 + 19^2 + ... + O(19^20), 1),
(2 + 13*19^10 + 9*19^11 + ... + O(19^20), 1)]
```

Cela fonctionne pour une grande variété de domaines, avec une efficacité variable.

En particulier, choisir pour `D` le corps des nombres algébriques `QQbar` ou celui des algébriques réels `AA` permet de calculer de façon exacte les racines complexes ou (respectivement) les racines réelles d'un polynôme à coefficients rationnels :

```
sage: l = p.roots(AA); l
[(-1.626576561697786?, 1), (0.500000000000000?, 2),
(1.626576561697786?, 1), (2.000000000000000?, 2)]
```

Malgré leur affichage, les racines renvoyées ne sont pas de simples valeurs approchées, comme on peut le voir en forçant Sage à simplifier autant que possible la puissance quatrième de la première :

```
sage: a = l[0][0]^4; a.simplify(); a
7
```

Une variante consiste à simplement *isoler* les racines, c'est-à-dire calculer des intervalles garantis contenir chacun exactement une racine, en passant comme domaine `D` celui des intervalles réels `RIF` ou complexes `CIF`.

Parmi les autres domaines utiles dans le cas d'un polynôme à coefficients rationnels, citons `RR`, `CC`, `RDF`, `CDF`, qui correspondent tous à des racines approchées, cherchées numériquement, ainsi que les corps de nombres `QQ[alpha]`. Les méthodes spécifiques `real_roots`, `complex_roots` et (pour certains anneaux de base) `real_interval_roots` offrent des options supplémentaires ou donnent des résultats légèrement différents des appels correspondants à `roots`. La recherche et l'isolation de racines numériques sont traitées plus en détail en §7.2.

Résultant. Sur tout anneau factoriel, l'existence d'un facteur commun non constant à deux polynômes se caractérise par l'annulation de leur *résultant* $\mathrm{Res}(p,q)$, qui est un polynôme en leurs coefficients. Un intérêt majeur du résultant par rapport au pgcd est qu'il se spécialise bien. Par exemple, les polynômes $x-12$ et $x-20$ sont premiers entre eux dans $\mathbb{Z}[x]$, mais calculer leur résultant montre qu'ils ont une racine commune dans $\mathbb{Z}/n\mathbb{Z}$ pour n divisant 8 :

```
sage: x = polygen(ZZ); (x-12).resultant(x-20)
-8
```

Soient $p=\sum_{i=0}^{m} p_i x^i$ et $q=\sum_{i=0}^{n} q_i x^i$ deux polynômes non constants de $A[x]$, avec $p_m, q_n \neq 0$. Le résultant de p et q est défini par

$$\mathrm{Res}(p,q)=\begin{vmatrix} p_m & \cdots & & \cdots & p_0 & & \\ & \ddots & & & & \ddots & \\ & & p_m & & \cdots & \cdots & p_0 \\ q_n & \cdots & & q_0 & & & \\ & \ddots & & & \ddots & & \\ & & \ddots & & & \ddots & \\ & & & & q_n & \cdots & q_0 \end{vmatrix},$$

(C'est le déterminant, dans des bases convenables, de l'application linéaire

$$\begin{array}{rcl} A_{n-1}[x]\times A_{m-1}[x] & \to & A_{m+n-1}[x] \\ u,v & \mapsto & up+vq \end{array}$$

où $A_k[x]\subset A[x]$ désigne le sous-module des polynômes de degré au plus k.) Si p et q sont scindés, leur résultant s'écrit aussi en fonction des différences de leurs racines :

$$\mathrm{Res}(p,q)=p_m^m q_n^n \prod_{i,j}(\alpha_i-\beta_j), \quad \begin{cases} p=p_m(x-\alpha_1)\dots(x-\alpha_m) \\ q=q_n(x-\beta_1)\dots(x-\beta_n). \end{cases}$$

La propriété de spécialisation mentionnée plus haut se déduit de la définition : si $\varphi : A\to A'$ est un morphisme d'anneau dont l'application à p ne fait pas chuter son degré (autrement dit, $\varphi(p_m)\neq 0$), alors

$$\mathrm{Res}(\varphi(p),\varphi(q))=\varphi(\mathrm{Res}(p,q)).$$

L'utilisation la plus commune du résultant concerne le cas où l'anneau de base est lui-même un anneau de polynômes. Par exemple, le discriminant de p est à une normalisation près le résultant de p et de sa dérivée :

$$\mathrm{disc}(p) = (-1)^{d(d-1)/2} \operatorname{Res}(p, p')/p_m.$$

Il s'annule donc quand p a une racine multiple. Les discriminants des polynômes généraux de degré 2 ou 3 sont classiques :

```
sage: R.<a,b,c,d> = QQ[]; x = polygen(R); p = a*x^2+b*x+c
sage: p.resultant(p.derivative())
-a*b^2 + 4*a^2*c
sage: p.discriminant()
b^2 - 4*a*c
sage: (a*x^3 + b*x^2 + c*x + d).discriminant()
b^2*c^2 - 4*a*c^3 - 4*b^3*d + 18*a*b*c*d - 27*a^2*d^2
```

Groupe de Galois. Le groupe de Galois d'un polynôme irréductible $p \in \mathbb{Q}[x]$ est un objet algébrique qui décrit certaines « symétries » des racines de p. Il s'agit d'un objet central de la théorie des équations algébriques. Notamment, l'équation $p(x) = 0$ est résoluble par radicaux, c'est-à-dire que ses solutions s'expriment à partir des coefficients de p au moyen des quatre opérations et de l'extraction de racine n-ième, si et seulement si le groupe de Galois de p est *résoluble*.

Sage permet de calculer les groupes de Galois des polynômes à coefficients rationnels de degré modéré, et d'effectuer toutes sortes d'opérations sur les groupes obtenus. Tant la théorie de Galois que les fonctionnalités de théorie des groupes de Sage dépassent le cadre de ce livre. Bornons-nous à appliquer sans plus d'explications le théorème de Galois sur la résolubilité par radicaux. Le calcul suivant[6] montre que les racines de $x^5 - x - 1$ ne s'expriment pas par radicaux :

```
sage: x = polygen(QQ); G = (x^5 - x -1).galois_group(); G
Transitive group number 5 of degree 5
sage: G.is_solvable()
False
```

Il s'agit d'un des exemples les plus simples dans ce cas, puisque les polynômes de degré inférieur ou égal à 4 sont toujours résolubles par radicaux, de même évidemment que ceux de la forme $x^5 - a$. En examinant les générateurs de G vu comme un groupe de permutations, on reconnaît que $G \simeq \mathfrak{S}_5$, ce que l'on vérifie facilement :

[6] Ce calcul nécessite une table de groupes finis qui ne fait pas partie de l'installation de base de Sage, mais que l'on peut télécharger et installer automatiquement par la commande `install_package("database_gap")`.

```
sage: G.gens()
[(1,2,3,4,5), (1,2)]
sage: G.is_isomorphic(SymmetricGroup(5))
True
```

9.1.6 Idéaux et quotients de $A[x]$

Les idéaux des anneaux de polynômes, et les quotients par ces idéaux, sont aussi des objets Sage, construits à partir de l'anneau de polynômes par les méthodes `ideal` et `quo`. Le produit d'une liste de polynômes par un anneau de polynômes est aussi interprété comme un idéal :

```
sage: R.<x> = QQ[]
sage: J1 = (x^2 - 2*x + 1, 2*x^2 + x - 3)*R; J1
Principal ideal (x - 1) of Univariate Polynomial Ring in x
over Rational Field
```

On peut multiplier les idéaux, et réduire un polynôme modulo un idéal :

```
sage: J2 = R.ideal(x^5 + 2)
sage: ((3*x+5)*J1*J2).reduce(x^10)
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
```

Le polynôme réduit demeure dans ce cas un élément de `QQ['x']`. Une autre possibilité consiste à construire le quotient par un idéal et y projeter des éléments. Le parent du projeté est alors l'anneau quotient. La méthode `lift` des éléments du quotient sert à les relever dans l'anneau de départ.

```
sage: B = R.quo((3*x+5)*J1*J2)  # quo nomme automatiquement 'xbar'
sage: B(x^10)                   #   le générateur de B image de x
421/81*xbar^6 - 502/81*xbar^5 + 842/81*xbar - 680/81
sage: B(x^10).lift()
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
```

Si K est un corps, l'anneau $K[x]$ est principal : les idéaux sont représentés dans les calculs par un générateur, et tout ceci n'est guère qu'un langage plus algébrique pour les opérations vues en §9.1.3. Son principal intérêt est que les anneaux quotients peuvent être utilisés dans de nouvelles constructions, ici $(\mathbb{F}_5[x]/\langle x^2+3\rangle)[y]$:

```
sage: R.<t> = GF(5)[]; R.quo(x^2+3)['x'].random_element()
(3*tbar + 1)*x^2 + (2*tbar + 3)*x + 3*tbar + 4
```

Sage permet aussi de construire des idéaux d'anneaux non principaux comme $\mathbb{Z}[x]$, mais les opérations disponibles sont alors limitées — sauf dans le cas des polynômes à plusieurs indéterminées sur un corps, nous y reviendrons longuement à la fin de ce chapitre.

Construction d'idéaux et d'anneaux quotients $Q = R/I$	
idéal $\langle u, v, w\rangle$	`R.ideal(u, v, w)` *ou* `(u, v, w)*A['x']`
réduction de p modulo J	`J.reduce(p)` *ou* `p.mod(J)`
anneau quotient R/I, $R/\langle p\rangle$	`R.quo(J)`, `R.quo(p)`
anneau quotienté pour obtenir Q	`Q.cover_ring()`
corps de nombres isomorphe	`Q.number_field()`

Éléments de $K[x]/\langle p\rangle$	
relevé (section de $R \twoheadrightarrow R/J$)	`u.lift()`
polynôme minimal	`u.minpoly()`
polynôme caractéristique	`u.charpoly()`
matrice	`u.matrix()`
trace	`u.trace()`

TABLEAU 9.5 – Idéaux et quotients.

Exercice 30. On définit la suite $(u_n)_{n\in\mathbb{N}}$ par les conditions initiales $u_n = n + 7$ pour $0 \leqslant n < 1000$ et la relation de récurrence linéaire

$$u_{n+1000} = 23u_{n+729} - 5u_{n+2} + 12u_{n+1} + 7 \qquad (n \geqslant 0).$$

Calculer les cinq derniers chiffres de $u_{10^{100}}$. *Indication : on pourra s'inspirer de l'algorithme de la §3.1.4. Mais celui-ci prend trop de temps quand l'ordre de la récurrence est élevé. Introduire un quotient d'anneau de polynômes judicieux pour éviter ce problème.*

Un cas particulier important est le quotient de $K[x]$ par un polynôme irréductible pour réaliser une extension algébrique de K. Les corps de nombres, extensions finies de $\mathbb{Q}$, sont représentés par des objets `NumberField` distincts des quotients de `QQ['x']`. Quand cela a un sens, la méthode `number_field` d'un quotient d'anneau de polynômes renvoie le corps de nombres correspondant. L'interface des corps de nombres, plus riche que celle des anneaux quotients, dépasse le cadre de ce livre. Les corps finis composés $\mathbb{F}_{p^k}$, réalisés comme extensions algébriques des corps finis premiers $\mathbb{F}_p$, sont quant à eux décrits en §8.1.

9.1.7 Fractions rationnelles

Diviser deux polynômes (sur un anneau intègre) produit une fraction rationnelle. Son parent est le corps des fractions de l'anneau de polynômes, qui peut s'obtenir par `Frac(R)` :

```
sage: x = polygen(RR); r = (1 + x)/(1 - x^2); r.parent()
Fraction Field of Univariate Polynomial Ring in x over Real
Field with 53 bits of precision
sage: r
(x + 1.00000000000000)/(-x^2 + 1.00000000000000)
```

Fractions rationnelles	
corps $K(x)$	`Frac(K['x'])`
numérateur	`r.numerator()`
dénominateur	`r.denominator()`
simplification (modifie `r`)	`r.reduce()`
décomposition en éléments simples	`r.partial_fraction_decomposition()`
reconstruction rationnelle	`p.rational_reconstruct(m)`

Séries tronquées	
anneau $A[[t]]$	`PowerSeriesRing(A, 'x', default_prec=10)`
anneau $A((t))$	`R.<x> = LaurentSeriesRing(A, 'x')`
coefficient $[x^k]\, f(x)$	`f[k]`
troncature	`x + O(x^n)`
précision	`f.prec()`
dérivée, primitive (nulle en 0)	`f.derivative()`, `f.integral()`
opérations usuelles $\sqrt{f}$, $\exp f$, ...	`f.sqrt()`, `f.exp()`, ...
inverse fonctionnel ($g \circ f = x$)	`g = f.reversion()`
solution de $y' + ay = b$	`a.solve_linear_de(precision, b)`

TABLEAU 9.6 – Objets construits à partir des polynômes.

On observe que la simplification n'est pas automatique. C'est parce que `RR` est un anneau *inexact*, c'est-à-dire dont les éléments s'interprètent comme des approximations d'objets mathématiques. La méthode `reduce` met la fraction sous forme réduite. Elle ne renvoie pas de nouvel objet, mais modifie la fraction rationnelle existante :

```
sage: r.reduce(); r
1.00000000000000/(-x + 1.00000000000000)
```

Sur un anneau exact, en revanche, les fractions rationnelles sont automatiquement réduites.

Les opérations sur les fractions rationnelles sont analogues à celles sur les polynômes. Celles qui ont un sens dans les deux cas (substitution, dérivée, factorisation, ...) s'utilisent de la même façon. Le tableau 9.6 énumère quelques autres méthodes utiles. La décomposition en éléments simples et surtout la reconstruction rationnelle méritent quelques explications.

Décomposition en éléments simples. Sage calcule la décomposition en éléments simples d'une fraction rationnelle a/b de `Frac(K['x'])` à partir de la factorisation de b dans `K['x']`. Il s'agit donc de la décomposition en éléments simples sur K. Le résultat est formé d'une partie polynomiale p et d'une liste de fractions rationnelles dont les dénominateurs sont des puissances de facteurs irréductibles de b :

```
sage: R.<x> = QQ[]; r = x^10/((x^2-1)^2*(x^2+3))
sage: poly, parts = r.partial_fraction_decomposition()
```

```
sage: print poly
x^4 - x^2 + 6
sage: for part in parts: print part.factor()
(17/32) * (x - 1)^-2 * (x - 15/17)
(-17/32) * (x + 1)^-2 * (x + 15/17)
(-243/16) * (x^2 + 3)^-1
```

On a ainsi obtenu la décomposition en éléments simples sur les rationnels

$$r = \frac{x^{10}}{(x^2-1)^2(x^2+3)} = x^4 - x^2 + 6 + \frac{\frac{17}{32}x - 15}{(x-1)^2} + \frac{-\frac{17}{32}x - 15}{(x+1)^2} + \frac{-\frac{243}{16}}{x^2+3}.$$

Il n'est pas difficile de voir que c'est aussi la décomposition de r sur les réels.

Sur les complexes en revanche, le dénominateur du dernier terme n'est pas irréductible, et donc la fraction rationnelle peut encore se décomposer. On peut calculer la décomposition en éléments simples sur les réels ou les complexes numériquement :

```
sage: C = ComplexField(15)
sage: Frac(C['x'])(r).partial_fraction_decomposition()
(x^4 - x^2 + 6.000, [(0.5312*x - 0.4688)/(x^2 - 2.000*x + 1.000),
4.384*I/(x - 1.732*I), (-4.384*I)/(x + 1.732*I),
(-0.5312*x - 0.4688)/(x^2 + 2.000*x + 1.000)])
```

Effectuer le même calcul exactement n'est pas immédiat, car Sage (en version 4.3) ne permet pas de calculer la décomposition en éléments simples sur `AA` ou `QQbar`.

Reconstruction rationnelle, approximation de Padé. Un analogue de la recontruction rationnelle présentée en §8.1.3 existe en Sage pour les polynômes à coefficients dans $A = \mathbb{Z}/n\mathbb{Z}$: la commande

```
s.rational_reconstruct(m, dp, dq)
```

calcule lorsque c'est possible des polynômes $p, q \in A[x]$ tels que

$$qs \equiv p \mod m, \qquad \deg p \leqslant d_p, \quad \deg q \leqslant d_q.$$

Restreignons-nous pour simplifier au cas où n est premier. Une telle relation avec q et m premiers entre eux entraîne $p/q = s$ dans $A[x]/\langle m\rangle$, d'où le nom de reconstruction rationnelle.

Le problème de reconstruction rationnelle se traduit par un système linéaire sur les coefficients de p et q, et un simple argument de dimension montre qu'il admet une solution non triviale dès que $d_p + d_q \geqslant \deg m - 1$. Il n'y a pas toujours de solution avec q et m premiers entre eux (par exemple, les solutions de $p \equiv qx \mod x^2$ avec $\deg p \leqslant 0$, $\deg q \leqslant 1$ sont les multiples constants de $(p, q) = (0, x)$), mais `rational_reconstruct` cherche en priorité les solutions qui satisfont cette condition supplémentaire.

Un cas particulier important est l'approximation de Padé, qui correspond à $m = x^n$. Un approximant de Padé de type $(k, n-k)$ d'une série formelle $f \in K[[x]]$ est une fraction rationnelle $p/q \in K(x)$ telle que $\deg p \leqslant k-1$, $\deg q \leqslant n-k$, $q(0) = 1$, et $p/q = f + O(x^n)$ (et donc $p/q \equiv f \mod x^n$).

Commençons par un exemple purement formel. Les commandes suivantes calculent un approximant de Padé de la série $f = \sum_{i=0}^{\infty} (i+1)^2 x^i$ à coefficients dans $\mathbb{Z}/101\mathbb{Z}$:

```
sage: R.<x> = Integers(101)[]
sage: f6 = sum( (i+1)^2 * x^i for i in (0..5) ); f6
36*x^5 + 25*x^4 + 16*x^3 + 9*x^2 + 4*x + 1
sage: num, den = f6.rational_reconstruct(x^6, 1, 3); num/den
(100*x + 100)/(x^3 + 98*x^2 + 3*x + 100)
```

En redéveloppant en série la fraction rationnelle trouvée, on observe que non seulement les développements coïncident jusqu'à l'ordre 6, mais le terme suivant aussi « est juste » !

```
sage: S = PowerSeriesRing(A, 'x', 7); S(num)/S(den)
1 + 4*x + 9*x^2 + 16*x^3 + 25*x^4 + 36*x^5 + 49*x^6 + O(x^7)
```

En effet, la série $f = (1+x)/(1-x)^3$ est elle-même une fraction rationnelle. Le développement tronqué `f6`, accompagné de bornes sur les degrés du numérateur et du dénominateur, suffit à la représenter sans ambiguïté. Cela a une grande importance dans certains algorithmes de calcul formel. De ce point de vue, le calcul d'approximants de Padé est l'inverse du développement en série des fractions rationnelles. Il permet de repasser de cette représentation alternative à la représentation habituelle comme quotient de deux polynômes.

Historiquement, pourtant, les approximants de Padé ne sont pas nés de ce genre de considérations formelles, mais de la théorie de l'approximation des fonctions analytiques. En effet, les approximants de Padé du développement en série d'une fonction analytique approchent souvent mieux la fonction que les troncatures de la série, voire, quand le degré du dénominateur est assez grand, peuvent fournir de bonnes approximations même en-dehors du disque de convergence de la série. On dit parfois qu'ils « avalent les pôles ». La figure 9.1, qui montre la convergence des approximants de Padé de type $(2k, k)$ de la fonction tangente au voisinage de 0, illustre ce phénomène.

Bien que `rational_reconstruct` soit limité aux polynômes sur $\mathbb{Z}/n\mathbb{Z}$, il est possible de s'en servir pour calculer des approximants de Padé à coefficients rationnels, et aboutir à cette figure. Le plus simple est de commencer par effectuer la reconstruction rationnelle modulo un nombre premier assez grand :

```
sage: x = var('x'); s = tan(x).taylor(x, 0, 20)
sage: p = previous_prime(2^30); ZpZx = Integers(p)['x']
sage: Qx = QQ['x']
```

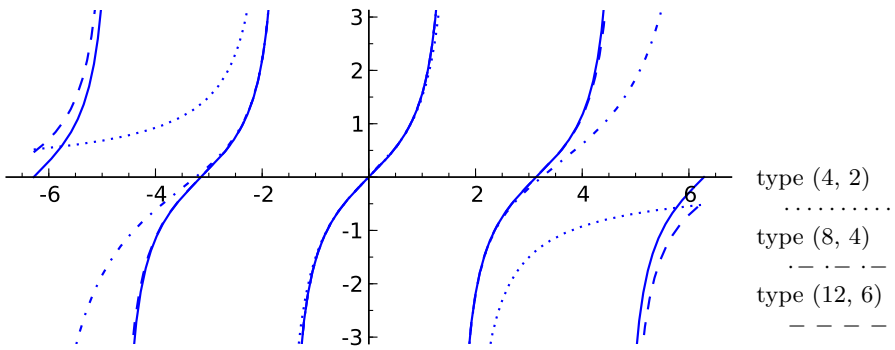


FIG. 9.1 – La fonction tangente et quelques approximants de Padé sur $[-2\pi, 2\pi]$.

```
sage: num, den = ZpZx(s).rational_reconstruct(ZpZx(x)^10, 4, 5)
sage: num/den
(1073741779*x^3 + 105*x)/(x^4 + 1073741744*x^2 + 105)
```

puis de relever la solution trouvée :

```
sage: def signed_lift(a):
....:     m = a.parent().defining_ideal().gen()
....:     n = a.lift()
....:     if n <= m // 2: return n
....:     else: return n - m

sage: Qx(map(signed_lift, num))/Qx(map(signed_lift, den))
(-10*x^3 + 105*x)/(x^4 - 45*x^2 + 105)
```

Lorsque les coefficients cherchés sont trop grands pour cette technique, on peut faire le calcul modulo plusieurs premiers, et appliquer le théorème chinois pour retrouver une solution à coefficients entiers, comme expliqué en §8.1.4. Une autre possibilité est de calculer une relation de récurrence à coefficients constants satisfaite par les coefficients de la série. Ce calcul est presque équivalent à celui d'un approximant de Padé (voir exercice 31), mais la fonction `berlekamp_massey` de Sage permet de le faire sur un corps quelconque.

Systématisons un peu le calcul précédent, en écrivant une fonction qui calcule directement l'approximant à coefficients rationnels, sous des hypothèses suffisamment favorables :

```
sage: def mypade(pol, n, k):
....:     x = ZpZx.gen();
....:     n, d = ZpZx(pol).rational_reconstruct(x^n, k-1, n-k)
....:     return Qx(map(signed_lift, n))/Qx(map(signed_lift, d))
```

Il n'y a plus alors qu'à appeler `plot` sur les résultats de cette fonction (convertis en éléments de `SR`, car `plot` ne permet pas de tracer directement le graphe d'une fraction rationnelle) pour obtenir le graphique de la figure 9.1 :

```
sage: add(
....:     plot(expr, -2*pi, 2*pi, ymin=-3, ymax=3, linestyle=style,
....:                         detect_poles=True, aspect_ratio=1)
....:     for (expr, style) in [
....:         (tan(x), '-'),
....:         (SR(mypade(tan_series, 4,  2)), ':' ),
....:         (SR(mypade(tan_series, 8,  4)), '-.'),
....:         (SR(mypade(tan_series, 12, 6)), '--') ])
```

Les exercices suivants présentent deux autres applications classiques de la reconstruction rationnelle.

Exercice 31.

1. Montrer que si $(u_n)_{n\in\mathbb{N}}$ satisfait une récurrence linéaire à coefficients constants, la série formelle $\sum_{n\in\mathbb{N}} u_n z^n$ est une fraction rationnelle. Comment s'interprètent le numérateur et le dénominateur ?
2. Deviner les termes suivants de la suite

$$1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339, 3056, 371, 14602, -4257, \ldots,$$

en utilisant `rational_reconstruct`. Retrouver le résultat à l'aide de la fonction `berlekamp_massey`.

Exercice 32 (Interpolation de Cauchy)**.** Trouver une fraction rationnelle $r = p/q \in \mathbb{F}_{17}(x)$ telle que $r(0) = -1$, $r(1) = 0$, $r(2) = 7$, $r(3) = 5$, avec p de degré minimal.

9.1.8 Séries

Nous étudions ici une dernière famille d'objets en une variable construits à partir des polynômes : les séries. Les séries du calcul formel sont le plus souvent des séries tronquées, c'est-à-dire des objets de la forme

$$f = f_0 + f_1\, x + \cdots + f_{n-1}\, x^{n-1} + O(x^n).$$

Elles sont utiles pour manipuler des fonctions qui n'admettent pas d'expression simple en forme close, notamment des solutions d'équations, et interviennent aussi fréquemment en combinatoire (voir chapitre 12) en tant que séries génératrices.

Arithmétique. L'anneau de séries formelles $\mathbb{Q}[[x]]$ s'obtient par

```
sage: R.<x> = PowerSeriesRing(QQ)
```

ou en abrégé `R.<x> = QQ[[]]`[7]. Les éléments de `A[['x']]` sont des séries tronquées, qui représentent des approximations des séries infinies du « véritable » $A[[x]]$, tout comme les éléments de `RR` sont des flottants considérés comme des approximations de réels. L'anneau `A[['x']]` est donc un anneau inexact.

Chaque série possède son propre ordre de troncature, et la précision est suivie au cours des calculs[8] :

```
sage: R.<x> = QQ[[]]
sage: f = 1 + x + O(x^2); g = x + 2*x^2 + O(x^4)
sage: f + g
1 + 2*x + O(x^2)
sage: f * g
x + 3*x^2 + O(x^3)
```

Il existe des séries de précision infinie, qui correspondent exactement aux polynômes :

```
sage: (1 + x^3).prec()
+Infinity
```

La précision par défaut, utilisée quand il est nécessaire de tronquer un résultat exact, se règle à la création de l'anneau, ou ensuite par sa méthode `set_default_prec` :

```
sage: R.<x> = PowerSeriesRing(Reals(24), default_prec=4)
sage:  1/(1 + RR.pi() * x)^2
1.00000 - 6.28319*x + 29.6088*x^2 - 124.025*x^3 + O(x^4)
```

Tout cela signifie qu'il n'est pas possible de tester l'égalité mathématique de deux séries. Cela constitue une différence conceptuelle importante entre celles-ci et les autres classes d'objets vues dans ce chapitre.

Attention, Sage considère cependant deux éléments de `A[['x']]` comme égaux dès qu'ils coïncident jusqu'à la plus faible de leurs précisions :

```
sage: R.<x> = QQ[[]]
sage: 1 + x + O(x^2) == 1 + x + x^2 + O(x^3)
True
```

En particulier, puisque `0` a une précision infinie, le test `O(x^2) == 0` renvoie vrai.

À nouveau, les opérations arithmétiques de base fonctionnent comme sur les polynômes. Le quotient par une série de valuation non nulle (sur un corps) crée une série de Laurent. On dispose aussi de quelques fonctions usuelles, par exemple `f.exp()` lorsque $f(0) = 0$, ainsi que des opérations de dérivation et d'intégration :

[7] Ou d'ailleurs à partir de $\mathbb{Q}[x]$, par `QQ['x'].completion('x')`.

[8] D'un certain point de vue, c'est la principale différence entre un polynôme modulo x^k et une série tronquée à l'ordre k : les opération sur ces deux sortes d'objets sont analogues, mais les éléments de $A[[x]]/\langle x^k \rangle$ ont, eux, tous la même « précision ».

```
sage: (1/(1+x)).sqrt().integral().exp()/x^2 + O(x^4)
x^-2 + x^-1 + 1/4 + 1/24*x - 1/192*x^2 + 11/1920*x^3 + O(x^4)
```

Développement de solutions d'équations. Face à une équation différentielle dont les solutions exactes sont trop compliquées à calculer ou à exploiter une fois calculées, ou tout simplement qui n'admet pas de solution en forme close, un recours fréquent consiste à chercher des solutions sous forme de séries. Plus précisément, on commence habituellement par déterminer les solutions de l'équation dans l'espace des séries formelles, et si nécessaire, on conclut ensuite par un argument de convergence que les solutions formelles construites ont un sens analytique. Le calcul formel peut être d'une aide précieuse pour la première étape.

Considérons par exemple l'équation différentielle

$$y'(x) = \sqrt{1+x^2}\, y(x) + \exp(x), \qquad y(0) = 1.$$

Cette équation admet une unique solution en série formelle, dont on peut calculer les premiers termes par

```
sage: (1+x^2).sqrt().solve_linear_de(prec=6, b=x.exp())
1 + 2*x + 3/2*x^2 + 5/6*x^3 + 1/2*x^4 + 7/30*x^5 + O(x^6)
```

De plus, le théorème de Cauchy d'existence de solutions d'équations différentielles linéaires à coefficients analytiques assure que cette série converge pour $|x| < 1$: sa somme fournit donc une solution analytique sur le disque unité complexe.

Cette approche n'est pas limitée aux équations différentielles. L'équation fonctionnelle $e^{xf(x)} = f(x)$ est plus compliquée, ne serait-ce que parce qu'elle n'est pas linéaire. Mais c'est une équation de point fixe, nous pouvons essayer de raffiner une solution (formelle) par itération :

```
sage: S.<x> = PowerSeriesRing(QQ, default_prec=5)
sage: f = S(1)
sage: for i in range(5):
....:     f = (x*f).exp()
....:     print f
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + O(x^5)
```

Que se passe-t-il ici ? Les solutions de $e^{xf(x)} = f(x)$ dans $\mathbb{Q}[[x]]$ sont les points fixes de la transformation $\Phi : f \mapsto e^{xf}$. Si une suite d'itérés de la forme $\Phi^n(a)$ converge, sa limite est nécessairement solution de l'équation. Inversement, posons $f(x) = \sum_{n=0}^{\infty} f_n\, x^n$, et développons en série les deux

membres : il vient

$$\begin{aligned}\sum_{n=0}^{\infty} f_n\, x^n &= \sum_{k=0}^{\infty} \frac{1}{k!}\Big(x \sum_{j=0}^{\infty} f_j\, x^j\Big)^n \\ &= \sum_{n=0}^{\infty}\Big(\sum_{k=0}^{\infty} \frac{1}{k!} \sum_{\substack{j_1,\dots,j_k\in\mathbb{N}\\ j_1+\cdots+j_k=n-k}} f_{j_1} f_{j_2} \dots f_{j_k}\Big)\, x^n.\end{aligned} \tag{9.1}$$

Peu importent les détails de la formule ; l'essentiel est que f_n se calcule en fonction des coefficients précédents $f_0, \dots, f_{n-1}$. Comme dans beaucoup de méthodes du point fixe, chaque itération de Φ fournit donc (au moins) un nouveau terme correct.

Exercice 33. Calculer le développement limité à l'ordre 15 de $\tan x$ au voisinage de zéro à partir de l'équation différentielle $\tan' = 1 + \tan^2$.

Séries paresseuses. Il existe en Sage une autre sorte de séries formelles appelées séries paresseuses. Ce ne sont pas des séries tronquées, mais bien des séries infinies ; l'adjectif paresseux signifie que les coefficients ne sont calculés que quand ils sont explicitement demandés. La contrepartie est que l'on ne peut représenter que des séries dont on sait calculer les coefficients : essentiellement, des combinaisons de séries de base et certaines solutions d'équations pour lesquelles existent des relations comme (9.1). Par exemple, la série paresseuse `lazy_exp` définie par[9]

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: lazy_exp = x.exponential(); lazy_exp
O(1)
```

est un objet qui contient le nécessaire pour calculer le développement en série de $\exp x$. Elle s'affiche initialement comme `O(1)` car aucun coefficient n'a encore été calculé. Tenter d'accéder au coefficient de x^5 déclenche le calcul, et les coefficients calculés sont alors mémorisés :

```
sage: lazy_exp[5]
1/120
sage: lazy_exp
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
```

Une présentation en bonne et due forme des séries paresseuses dépasserait le cadre de ce livre. Bornons-nous à reprendre l'exemple de l'équation $e^{xf(x)} = f(x)$ pour montrer comment il se simplifie. Nous pouvons d'abord essayer de reproduire le calcul fait plus haut dans l'anneau `QQ[['x']]` :

[9] Une future version de Sage devrait contenir un nouvel anneau de séries paresseuses appelé `FormalPowerSeriesRing`, plus puissant que `LazyPowerSeriesRing`.

```
sage: f = L(1)
sage: for i in range(5):
....:     f = (x*f).exponential()
....:     f.compute_coefficients(5)  # force le calcul des
....:     print f                    #   premiers coefficients
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + 49/30*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + 63/10*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 49/5*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 54/5*x^5 + O(x^6)
```

Les développements obtenus sont bien sûr les mêmes que précédemment[10]. Mais la valeur de `f` à chaque itération est maintenant une série complète, dont on peut calculer des coefficients à la demande. Toutes ces séries intermédiaires sont conservées en mémoire, et le calcul de chacune est automatiquement poussé à la précision requise si l'on accède, par exemple, au coefficient de x^7 dans le dernier itéré :

```
sage: f[7]
28673/630
```

Dans la première version de cet exemple, l'accès à `f[7]` aurait déclenché une erreur `IndexError`, l'indice 7 étant supérieur à l'ordre de troncature de la série `f`.

Cependant, la valeur renvoyée par `f[7]` n'est que le coefficient de x^7 dans l'itéré $\Phi^5(1)$, et non dans la solution. La force des séries paresseuses est la possibilité de passer directement à la limite, en codant f elle-même comme une série paresseuse :

```
sage: from sage.combinat.species.series import LazyPowerSeries
sage: f = LazyPowerSeries(L, name='f')
sage: f.define((x*f).exponential())
sage: f.coefficients(8)
[1, 1, 3/2, 8/3, 125/24, 54/5, 16807/720, 16384/315]
```

Ce qui « faisait marcher » le calcul itératif est la relation (9.1). En coulisses, Sage déduit de la définition récursive `f.define((x*f).exponential())` une formule du même genre, qui permet de calculer les coefficients par récurrence.

9.2 Polynômes à plusieurs indéterminées

Intéressons-nous maintenant aux polynômes à plusieurs indéterminées, ou — anglicisme commun dans le domaine du calcul formel — *multivariés*.

[10] On peut en revanche constater que Sage emploie parfois des conventions incohérentes : ce qui s'appelait `exp` s'appelle `exponential`, et `compute_coefficients(5)` calcule les coeffcients jusqu'à l'ordre 5 inclus tandis que `default_prec=5` donnait des séries tronquées après le coefficient de x^4.

Nous verrons apparaître des difficultés nouvelles mais aussi une plus grande richesse mathématique, liées en premier lieu au fait que l'anneau $K[x_1, \ldots, x_n]$ n'est pas principal. La théorie des bases de Gröbner fournit des outils pour contourner cette limitation. Au final, on dispose de méthodes puissantes, quoique coûteuses en temps de calcul, pour étudier les systèmes polynomiaux, avec d'innombrables applications qui couvrent des domaines variés.

9.2.1 Anneaux de polynômes à plusieurs indéterminées

Les anneaux $A[x_1, \ldots, x_n]$. Avant de pouvoir construire des polynômes, il nous faut définir une famille d'indéterminées vivant toutes dans un même anneau. Comme en une seule variable, on peut écrire :

```
sage: R = PolynomialRing(QQ, 'x,y,z')
sage: x, y, z = R.gens()  # renvoie le n-uplet des indéterminées
```

ou en abrégé `R.<x,y,z> = QQ[]`. Le constructeur `PolynomialRing` permet aussi de créer une famille d'indéterminées de même nom, avec des indices entiers. Placer le n-uplet renvoyé par `gens` lui-même dans la variable `x` permet alors d'accéder naturellement à l'indéterminée x_i par `x[i]` :

```
sage: R = PolynomialRing(QQ, 'x', 10)
sage: x = R.gens()
sage: sum(x[i] for i in xrange(5))
x0 + x1 + x2 + x3 + x4
```

Pour obtenir une famille d'indéterminées plus compliquée — ici, indexée par les nombres premiers — on passe à `PolynomialRing` une liste fabriquée par compréhension (voir §3.2.2) :

```
sage: R = PolynomialRing(QQ, ['x%d'%n for n in primes(40)])
sage: R.inject_variables()
Defining x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, x37
```

La méthode `inject_variables` initialise des variables Python `x2`, `x3`, ... contenant chacune le générateur correspondant de `R`.

Les anneaux $A[(x_n, y_n, \ldots)_{n\in\mathbb{N}}]$. Il arrive cependant que l'on ne sache pas, au début d'un calcul, combien de variables seront nécessaires. Cela rend l'utilisation de `PolynomialRing` assez pénible : il faut commencer à calculer dans un premier domaine, puis l'étendre et convertir tous les éléments à chaque fois que l'on souhaite introduire une nouvelle variable.

Les anneaux de polynômes en une infinité d'indéterminées sont une structure de données plus souple, conçue pour pallier ce problème. Leurs éléments peuvent contenir des variables prises dans une ou plusieurs familles infinies d'indéterminées : chaque générateur de l'anneau correspond non pas

Construction d'anneaux de polynômes	
anneau $A[x,y]$	`PolynomialRing(A, 'x,y')` *ou* `A['x,y']`
anneau $A[x_0,\ldots,x_{n-1}]$	`PolynomialRing(A, 'x', n)`
anneau $A[x_0,x_1,\ldots,y_0,y_1,\ldots]$	`InfinitePolynomialRing(A, ['x','y'])`
n-uplet des générateurs	`R.gens()`
1^{er}, 2^{e}, ... générateur	`R.0`, `R.1`, ...
indéterminées de $R = A[x,y][z][\ldots]$	`R.variable_names_recursive()`
conversion $A[x_1,x_2,y] \to A[x_1,x_2][y]$	`p.polynomial(y)`

Accès aux coefficients	
support, coefficients non nuls	`p.exponents()`, `p.coefficients()`
coefficient d'un monôme	`p[x^2*y]` *ou* `p[2,1]`
degré total	`p.degree()`
degré en x	`p.degree(x)`
degrés partiels	`p.degrees()`

Opérations de base	
transformation des coefficients	`p.map_coefficients(f)`
dérivée partielle $\mathrm{d}/\mathrm{d}x$	`p.derivative(x)`
évaluation $p(x,y)\vert_{x=a,y=b}$	`p.subs(x=a, y=b)`
homogénisation	`p.homogenize()`

TABLEAU 9.7 – Polynômes à plusieurs indéterminées.

à une seule variable, mais à une famille de variables indexées par les entiers naturels[11]. En voici un exemple d'utilisation :

```
sage: R.<x,y> = InfinitePolynomialRing(ZZ)
sage: p = mul(x[k] - y[k] for k in range(2)); p
x_1*x_0 - x_1*y_0 - x_0*y_1 + y_1*y_0
sage: p + x[100]
x_100 + x_1*x_0 - x_1*y_0 - x_0*y_1 + y_1*y_0
```

On peut revenir à un anneau usuel de polynômes grâce à la méthode `polynomial`, qui renvoie l'image d'un élément d'un `InfinitePolynomialRing` dans un anneau suffisamment grand pour contenir tous les éléments de l'anneau à une infinité de variables manipulés jusque-là.

Représentation récursive. Enfin, la *représentation récursive* consiste à coder les polynômes, disons, en x et y, comme des polynômes en y dont les coefficients sont des polynômes en x. Les conversions entre $A[x,y]$ et $A[x][y]$ fonctionnent comme on s'y attend :

```
sage: x = polygen(QQ); y = polygen(QQ[x], 'y')
sage: p = x^3 + x*y + y + y^2; p
```

[11] En contrepartie de cette souplesse, ces anneaux sont moins efficaces que les anneaux de polynômes habituels.

```
y^2 + (x + 1)*y + x^3
sage: q = QQ['x,y'](p); q
x^3 + x*y + y^2 + y
sage: QQ['x']['y'](q)
y^2 + (x + 1)*y + x^3
```

De plus, la méthode `polynomial` des polynômes multivariés permet d'isoler une variable, un peu comme la méthode `collect` des expressions :

```
sage: R.<x,y,z,t> = QQ[]; p = (x+y+z*t)^2
sage: p.polynomial(t).reverse() # le polynôme réciproque de p en t
(x^2 + 2*x*y + y^2)*t^2 + (2*x*z + 2*y*z)*t + z^2
```

Pourquoi introduire spécifiquement des anneaux de polynômes à plusieurs indéterminées, au lieu de se contenter de représenter ces derniers comme des éléments d'anneaux de polynômes en une indéterminée « emboîtés » ? Bien qu'utile à l'occasion, la représentation récursive n'est pas bien adaptée pour manipuler les systèmes polynomiaux, ce qui est une des raisons d'être des polynômes multivariés.

9.2.2 Polynômes à plusieurs indéterminées

Tout comme les polynômes en une variable sont de classe `Polynomial`, ceux en plusieurs variables sont de classe `MPolynomial`[12]. Pour les anneaux de base les plus usuels (comme $\mathbb{Z}$, $\mathbb{Q}$ ou $\mathbb{F}_q$), ils s'appuient sur le logiciel Singular, un système de calcul formel spécialisé dans les calculs rapides sur les polynômes. Dans les autres cas, Sage se rabat sur une implémentation générique beaucoup plus lente.

Les polynômes à plusieurs indéterminées sont toujours codés en représentation creuse[13]. Pourquoi ce choix ? Tout d'abord, un polynôme dense à n variables de degré total d compte $\binom{n+d}{d}$ monômes : pour $n = d = 10$, cela fait 184 756 coefficients à stocker ! Par ailleurs, même quand les polynômes sont denses, les supports (les positions des monômes non nuls) rencontrés en pratique ont des formes variées. Or, si par exemple un polynôme à n variables dense jusqu'au degré total $d-1$ est représenté par un tableau rectangulaire $d \times \cdots \times d$, pour d grand, seul un coefficient sur $n!$ environ est non nul. Au contraire, la représentation par dictionnaire s'adapte à la forme du support.

[12] Mais contrairement à `Polynomial`, cette classe n'est pas accessible directement depuis la ligne de commande : il faut utiliser son nom complet. Par exemple, on peut tester si un objet est de type polynôme multivarié par `isinstance(p, sage.rings.polynomial.multi_polynomial.MPolynomial)`.

[13] La représentation récursive fournit cependant une forme de polynômes multivariés partiellement denses. Dans la représentation en mémoire d'un polynôme de $A[x][y]$, chaque coefficient de y^k occupe un espace proportionnel à son degré en x, à quoi il faut ajouter une place proportionnelle au degré en y pour le polynôme lui-même.

Opérations de base. Les opérations arithmétiques `+`, `-` et `*`, de même que les méthodes `dict`, `coefficients`, et bien d'autres, s'utilisent comme leurs analogues en une seule variable. Parmi les petites différences, l'opérateur crochets `[]` d'extraction d'un coefficient accepte comme paramètre soit un monôme, soit son *exposant*, c'est-à-dire le n-uplet des puissances correspondantes des variables :

```
sage: R.<x,y,z> = QQ[]; p = y^2*x^2 + 3*y*x^2 + 2*y*z + x^3 + 6
sage: p[x^2*y] == p[(2,1,0)] == p[2,1,0] == 3
True
```

De même, l'évaluation nécessite de donner une valeur à chacune des variable ou de préciser celles à substituer :

```
sage: p.subs(x = 1, z = x^2+1)
2*x^2*y + y^2 + 5*y + 7
sage: p(0, 3, -1)
0
```

et le degré se décline en degré total et degrés partiels :

```
sage: print "total : %s ; en x : %s ; partiels : %s" % \
....:                 (p.degree(), p.degree(x), p.degrees())
total : 4 ; en x : 3 ; partiels : (3, 2, 1)
```

D'autres constructions subissent des adaptations évidentes, par exemple, la méthode `derivative` prend en paramètre la variable par rapport à laquelle dériver.

Enfin, la méthode `homogenize` calcule l'homogénisé d'un polynôme, soit en restant dans l'anneau parent, soit en l'étendant par une variable d'homogénisation :

```
sage: p.homogenize(x)
7*x^4 + 3*x^3*y + x^2*y^2 + 2*x^2*y*z
sage: p.homogenize()
x^2*y^2 + x^3*h + 3*x^2*y*h + 2*y*z*h^2 + 6*h^4
```

Itération. On a fréquemment besoin d'appliquer une transformation aux coefficients d'un polynôme, par exemple pour calculer le conjugué d'un polynôme à coefficients complexes. La méthode `map_coefficients` est là pour cela. Appliquée à un polynôme $p \in A[x_1, \ldots, x_n]$ avec comme paramètre une fonction $f : A \to A$, elle renvoie le polynôme obtenu en appliquant f à chacun des coefficients de p. Le plus souvent, f est une fonction anonyme introduite par la construction `lambda` (voir §3.2.2) :

```
sage: QQi.<i_> = QQ[i]  # i désigne le i de SR, i_ celui de QQi
sage: R.<x, y> = QQi[]; p = (x + i_*y)^3; p
x^3 + (3*I)*x^2*y - 3*x*y^2 + (-I)*y^3
sage: p.map_coefficients(lambda z: z.conjugate())
x^3 + (-3*I)*x^2*y - 3*x*y^2 + (I)*y^3
```

Sur cet exemple, on aurait aussi pu écrire `p.map_coefficients(conjugate)`, car `conjugate(z)` a le même effet que `z.conjugate` pour $z \in \mathbb{Q}[i]$. Appeler explicitement une méthode de l'objet `z` est plus sûr : ainsi, le code fonctionne avec tous les objets dotés d'une méthode `conjugate()`, et seulement ceux-là.

Arithmétique. Au-delà des opérations syntaxiques et arithmétiques élémentaires, les fonctions disponibles dans Sage sont en général limitées aux polynômes sur un corps, et parfois sur $\mathbb{Z}$ ou $\mathbb{Z}/n\mathbb{Z}$. Pour la suite de ce chapitre, et sauf mention explicite du contraire, nous nous placerons donc sur un corps.

La division euclidienne des polynômes n'a de sens qu'en une variable. En Sage, la méthode `quo_rem` et les opérateurs associés `//` et `%` restent pourtant définis pour les polynômes multivariés. La « division avec reste » qu'ils calculent vérifie

```
(p//q)*q + (p%q) == p
```

et coïncide avec la division euclidienne lorsque p et q ne dépendent que d'une variable, mais ce n'est pas elle-même une division euclidienne et elle n'a rien de canonique. Elle s'avère tout de même utile lorsque la division est exacte ou lorsque le diviseur est un monôme. Dans les autres cas, on préfèrera à `quo_rem` et ses variantes la méthode `mod`, qui réduit un polynôme modulo un idéal, en tenant compte du choix d'ordre monomial de l'anneau :

```
sage: R.<x,y> = QQ[]; p = x^2 + y^2; q = x + y
sage: print("(%s)*(%s) + (%s) == %s" % (p//q, q, p%q, p//q*q+p%q))
(-x + y)*(x + y) + (2*x^2) == x^2 + y^2
sage: p.mod(q)  # n'est PAS équivalent à p%q
2*y^2
```

Les méthodes `divides`, `gcd`, `lcm` ou encore `factor` ont le même sens qu'en une seule variable. Faute de division euclidienne, même les premières ne sont pas disponibles sur un corps quelconque ; mais elles fonctionnent sur un certain nombre de corps usuels, par exemple les corps de nombres :

```
sage: R.<x,y> = QQ[exp(2*I*pi/5)][]
sage: (x^10 + y^5).gcd(x^4 - y^2)
x^2 + y
sage: (x^10 + y^5).factor()
(x^2 + (a^2)*y) * (x^2 + (a)*y) * (x^2 + (a^3)*y) *
(x^2 + (-a^3 - a^2 - a - 1)*y) * (x^2 + y)
```

Notons que Sage refuse de factoriser sur les corps finis composés, à moins que l'on ne l'autorise à utiliser des méthodes non rigoureuses avec `proof=False` :

```
sage: k.<a> = GF(9); R.<x,y,z> = k[]
sage: (a*x^2*z^2 + x*y*z - y^2).factor(proof=False)
(a) * (x*z + (-a - 1)*y) * (x*z + (-a)*y)
```

Il s'agit là d'un mécanisme général, utilisé par plusieurs autres fonctions sur les polynômes en plusieurs variables. Dans la plupart des cas, les méthodes heuristiques ne servent qu'à gagner du temps (parfois beaucoup !) par rapport aux méthodes prouvées. On peut activer leur utilisation par défaut pour les opérations sur les polynômes avec `proof.polynomial(False)`, ou dans tout Sage par `proof.all(False)`.

Mathematics is the art of reducing any problem to linear algebra.

William Stein

10 Algèbre linéaire

Ce chapitre traite de l'algèbre linéaire exacte et symbolique, c'est-à-dire sur des anneaux propres au calcul formel, tels que $\mathbb{Z}$, des corps finis, des anneaux de polynômes, ... L'algèbre linéaire numérique est traitée quant à elle au chapitre 5. Nous présentons les constructions sur les matrices et leurs espaces ainsi que les opérations de base (§10.1), puis les différents calculs possibles sur ces matrices, regroupés en deux thèmes : ceux liés à l'élimination de Gauss et aux transformations par équivalence à gauche (§10.2.1 et §10.2.2), et ceux liés aux valeurs et espaces propres et aux transformations de similitude (§10.2.3). On peut trouver dans les ouvrages de Gantmacher [Gan90], de von zur Gathen et Gerhard [vzGG03], de Lombardi et Journaïdi [LA04] un traitement approfondi des notions abordées dans ce chapitre.

10.1 Constructions et manipulations élémentaires

10.1.1 Espace de vecteurs, de matrices

De la même façon que pour les polynômes, les vecteurs et les matrices sont manipulés comme des objets algébriques appartenant à un espace. Si les coefficients appartiennent à un corps K, c'est un espace vectoriel sur K ; s'ils appartiennent à un anneau, c'est un K-module libre.

On construit ainsi l'espace $\mathcal{M}_{2,3}(\mathbb{Z})$ et l'espace vectoriel $(\mathbb{F}_{3^2})^3$ par

```
sage: MS=MatrixSpace(ZZ,2,3); MS
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: VS=VectorSpace(GF(3^2,'x'),3); VS
```

```
Vector space of dimension 3 over Finite Field in x of size 3^2
```

Un système générateur pour ces espaces est donné par la base canonique ; elle peut être obtenue indifféremment par les méthodes `MS.gens()` ou `MS.basis()`.

```
sage: MS.basis()
[
[1 0 0]  [0 1 0]  [0 0 1]  [0 0 0]  [0 0 0]  [0 0 0]
[0 0 0], [0 0 0], [0 0 0], [1 0 0], [0 1 0], [0 0 1]
]
```

Groupes de matrices. On pourra par ailleurs définir des sous-groupes de l'espace total des matrices. Ainsi le constructeur `MatrixGroup([A,B,...])` retourne le groupe généré par les matrices passées en argument, qui doivent être inversibles.

```
sage: A=matrix(GF(11),2,2,[1,0,0,2])
sage: B=matrix(GF(11),2,2,[0,1,1,0])
sage: MG=MatrixGroup([A,B])
sage: MG.cardinality()
200
sage: identity_matrix(GF(11),2) in MG
True
```

Le groupe général linéaire de degré n sur un corps K, noté $\mathrm{GL}_n(K)$, est le groupe formé par les matrices $n \times n$ inversibles de $\mathcal{M}_{n,n}(K)$. Il se construit naturellement en Sage avec la commande `GL(n,K)`. Le groupe spécial linéaire $\mathrm{SL}_n(K)$ des éléments de $\mathrm{GL}_n(K)$ de déterminant 1 se construit avec la commande `SL(n,K)`.

10.1.2 Constructions des matrices et des vecteurs

Les matrices et les vecteurs peuvent naturellement être générés comme des éléments d'un espace en fournissant la liste des coefficients en arguments. Pour les matrices, ceux-ci seront lus *par ligne* :

```
sage: A=MS([1,2,3,4,5,6]);A
[1 2 3]
[4 5 6]
```

Le constructeur vide `MS()` retourne une matrice nulle, tout comme la méthode `MS.zero()`. Plusieurs constructeurs spécialisés permettent de produire les matrices usuelles, comme `random_matrix`, `identity_matrix`, `jordan_block` (voir tableau 10.1). En particulier on pourra construire des matrices et des vecteurs avec les constructeurs `matrix` et `vector`, sans devoir construire l'espace associé préalablement. Par défaut une matrice est construite sur l'anneau des entiers $\mathbb{Z}$ et a pour dimensions 0×0.

```
sage: a=matrix(); a.parent()
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
```

On peut naturellement spécifier un domaine pour les coefficients ainsi que les dimensions, pour créer ainsi une matrice nulle, ou remplie d'une liste de coefficients passée en argument.

```
sage: a=matrix(GF(8,'x'),3,4); a.parent()
Full MatrixSpace of 3 by 4 dense matrices over Finite Field in x
of size 2^3
```

Le constructeur `matrix` accepte aussi comme argument des objets admettant une transformation naturelle en matrice. Ainsi, il peut être utilisé pour obtenir la matrice d'adjacence d'un graphe, sous la forme d'une matrice à coefficients dans $\mathbb{Z}$.

```
sage: g = graphs.PetersenGraph()
sage: m = matrix(g); m; m.parent()
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring
```

Matrices par blocs. Pour construire une matrice par blocs à partir de sous-matrices, on peut utiliser la fonction `block_matrix`.

```
sage: A=matrix([[1,2],[3,4]])
sage: block_matrix([A,-A,2*A, A^2])
[ 1  2|-1 -2]
[ 3  4|-3 -4]
[-----+-----]
[ 2  4| 7 10]
[ 6  8|15 22]
```

La structure est carrée par blocs par défaut mais le nombre de blocs ligne ou colonne peut être spécifié par les arguments optionnels `ncols` et `nrows`. Lorsque cela est possible, un coefficient comme `0` ou `1` est interprété comme un block diagonal (ici zéro ou identité) de dimension appropriée.

Espaces de matrices	
construction	`MS=MatrixSpace(A, m, n)`
construction (creux)	`MS=MatrixSpace(A, m, n, sparse = True)`
anneau de base A	`MS.base_ring()`
extension de l'anneau	`MS.base_extend(B)`
changement de l'anneau	`MS.change_ring(B)`
groupe engendré	`MatrixGroup([M,N])`
base de l'espace	`MS.basis()` *ou* `MS.gens()`

Construction de matrices	
matrice nulle	`MS()` *ou* `MS.zero()` *ou* `matrix(A,nrows,ncols)`
matrice avec coefficients	`MS([1,2,3,4]` *ou* `matrix(ZZ,2,2,[1,2,3,4])` *ou* `matrix([[1,2][3,4]])`
matrice identité	`MS.one()` *ou* `MS.identity_matrix()` *ou* `identity_matrix(A,n)`
matrice aléatoire	`MS.random_element()` *ou* `random_matrix(A,nrows,ncols)`
bloc de Jordan	`jordan_block(x,n)`
matrice par blocs	`block_matrix([A,1,B,0])` , `block_diagonal_matrix`

Manipulations de base	
accès à un coefficient	`A[2,3]`
accès à une ligne, colonne	`A[:,2]`, `A[-1,:]`
accès aux colonnes paires	`A[:,0:8:2]`
sous-matrices	`A[3:4,2:5]`, `A[:,2:5]`, `A[:4,2:5]` `A.matrix_from_rows([1,2])`, `A.matrix_from_columns([2,3])`, `A.matrix_from_rows_and_columns([1,2],[2,3])` `A.submatrix(i,j,nrows,ncols)`
concaténation par lignes	`A.stack(B)`
concaténation par colonnes	`A.augment(B)`

TABLEAU 10.1 – Constructeurs de matrices et de leurs espaces.

```
sage: A=matrix([[1,2,3],[4,5,6]])
sage: block_matrix([1,A,0,0,-A,2], ncols=3)
[ 1  0| 1  2  3| 0  0]
[ 0  1| 4  5  6| 0  0]
[-----+--------+-----]
[ 0  0|-1 -2 -3| 2  0]
[ 0  0|-4 -5 -6| 0  2]
```

Pour le cas particulier des matrices diagonales par blocs, on passe simplement la liste des blocs diagonaux au constructeur `block_diagonal_matrix`.

```
sage: A=matrix([[1,2,3],[0,1,0]])
sage: block_diagonal_matrix(A,A.transpose())
[1 2 3|0 0]
[0 1 0|0 0]
[-----+---]
[0 0 0|1 0]
[0 0 0|2 1]
[0 0 0|3 0]
```

La structure par blocs n'est qu'une commodité d'affichage, et Sage traite la matrice comme toute autre matrice. On peut par ailleurs désactiver cet affichage en ajoutant l'argument `subdivide=False` au constructeur `block_matrix`.

10.1.3 Manipulations de base et arithmétique sur les matrices

Indices et accès aux coefficients. L'accès aux coefficients ainsi qu'à des sous-matrices extraites se fait de façon unifiée par l'opérateur crochet `A[i,j]`, selon les conventions usuelles de Python. Les indices de ligne i et de colonne j peuvent être des entiers (pour l'accès à des coefficients) ou des intervalles sous la forme `1:3` ou `[1..3]` (on rappelle que par convention, en Python les indices commencent à 0, et les intervalles sont toujours inclusifs pour la borne inférieure et exclusifs pour la borne supérieure). L'intervalle `:` sans valeurs correspond à la totalité des indices possibles dans la dimension considérée. La notation `a:b:k` permet d'accéder aux indices compris entre a et $b-1$ par pas de k. Enfin, les indices négatifs sont aussi valides, et permettent de parcourir les indices à partir de la fin. Ainsi `A[-2,:]` correspond à l'avant dernière ligne. L'accès à ces sous-matrices se fait aussi bien en lecture qu'en écriture. On peut par exemple modifier une colonne donnée de la façon suivante :

```
sage: A=matrix(3,3,range(9))
sage: A[:,1]=vector([1,1,1]); A
[0 1 2]
[3 1 5]
[6 1 8]
```

L'indice de pas c peut aussi être négatif, indiquant un parcours par valeurs décroissantes.

```
sage:  A[::-1], A[:,::-1], A[::2,-1]
(
[6 1 8]  [2 1 0]
[3 1 5]  [5 1 3]  [2]
[0 1 2], [8 1 6], [8]
)
```

Extraction de sous-matrices. Pour extraire une matrice à partir d'une liste d'indices de ligne ou de colonnes non nécessairement contigus, on utilise les méthodes `A.matrix_from_rows`, ou `A.matrix_from_columns` ou dans le cas général, la méthode `A.matrix_from_rows_and_columns`.

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

Pour extraire une sous-matrice de lignes et de colonnes contiguës, on pourra aussi utiliser la méthode `submatrix(i,j,m,n)` formant la sous-matrice de dimension $m \times n$ dont le coefficient supérieur gauche se trouve en position (i, j).

Plongements et extensions. Lorsque l'on veut plonger un espace de matrices dans l'espace des matrices de même dimension mais à coefficients dans une extension de l'anneau de base, on pourra utiliser la méthode `base_extend` de l'espace de matrice. Cette opération n'est toutefois valide que pour les opérations d'extension de corps ou d'anneau. Pour effectuer le changement de l'anneau de base suivant un morphisme (lorsqu'il existe), on utilise plutôt la méthode `change_ring`.

```
sage: MS=MatrixSpace(GF(3),2,3)
sage: MS.base_extend(GF(9,'x'))
Full MatrixSpace of 2 by 3 dense matrices over Finite Field
in x of size 3^2
sage: MS=MatrixSpace(ZZ,2,3)
sage: MS.change_ring(GF(3))
Full MatrixSpace of 2 by 3 dense matrices over Finite Field
of size 3
```

Mutabilité et mise en cache. Les objets représentant les matrices sont par défaut mutables, c'est-à-dire que l'on peut librement en modifier leurs membres (ici leurs coefficients) après leur construction. Si on souhaite protéger la matrice contre les modifications, on utilise la fonction `A.set_immutable()`. Il est alors toujours possible d'en tirer des copies mutables par la fonction `copy(A)`. À noter que le mécanisme de mise en cache des résultats calculés (tels que le rang, le déterminant, ...) reste toujours fonctionnel, quel que soit l'état de mutabilité.

Opérations de base	
transposée, conjuguée	`A.tranpose()`, `A.conjugate()`
produit externe	`a*A`
somme, produit, puissance k-ième, inverse	`A + B`, `A * B`, `A^k`, `A^-1`, `~A`

Tableau 10.2 – Opérations de base et arithmétique sur les matrices.

10.1.4 Opérations de base sur les matrices

Les opérations arithmétiques sur les matrices se font avec les opérateurs usuels `+,-,*,^`. L'inverse d'une matrice `A` peut s'écrire aussi bien `A^-1` que `~A`. Lorsque a est un scalaire et A une matrice, l'opération `a*A` correspond à la multiplication externe de l'espace de matrices. Pour les autres opérations où un scalaire `a` est fourni en lieu et place d'une matrice (par exemple l'opération `a+A`), il est considéré comme la matrice scalaire correspondante aI_n si $a \neq 0$ et les dimensions le permettent. Le produit élément par élément de deux matrices s'effectue avec l'opération `elementwise_product`.

10.2 Calculs sur les matrices

En algèbre linéaire, les matrices peuvent être utilisées pour représenter aussi bien des familles de vecteurs, des systèmes d'équations linéaires ou des applications linéaires, des sous-espaces. Ainsi, le calcul d'une propriété comme le rang d'une famille, la solution d'un système, les espaces propres d'une application linéaire, ou la dimension d'un sous-espace se ramènent à des transformations sur ces matrices révélant cette propriété.

Ces transformations correspondent à des changements de base, vus au niveau matriciel comme des transformation d'équivalence : $B = PAQ^{-1}$, où P et Q sont des matrices inversibles. Deux matrices sont dites équivalentes s'il existe une telle transformation pour passer de l'une à l'autre. On peut ainsi former des classes d'équivalence pour cette relation, et l'on définit des formes normales, permettant de caractériser de manière unique chaque classe d'équivalence. Dans ce qui suit, nous présentons l'essentiel des calculs sur les matrices disponibles avec Sage, sous l'angle de deux cas particuliers de ces transformations :

– Les transformations d'équivalence à gauche, de la forme $B = UA$, qui révèlent les propriétés caractéristiques pour les familles de vecteurs, telles que le rang (nombre de vecteurs linéairement indépendants), le déterminant (volume du parallélépipède décrit par la famille de vecteurs), le profil de rang (premier sous ensemble de vecteurs formant une base), ... L'élimination de Gauss est l'outil central pour ces transformations, et la forme échelonnée réduite (forme de Gauss-Jordan dans un corps ou forme de Hermite dans $\mathbb{Z}$) est la forme normale. En outre,

ces transformations servent à la résolution des systèmes linéaires.
- Les transformations de similitude, de la forme $B = UAU^{-1}$, qui révèlent les propriétés caractéristiques des matrices représentant des endomorphismes, comme les valeurs propres, les espaces propres, les polynômes minimal et caractéristique, ... La forme de Jordan ou la forme de Frobenius, selon les domaines de calcul, seront les formes normales pour ces transformations.
- La forme de Gram-Schmidt est une autre décomposition basée sur les transformations d'équivalence à gauche, transformant une matrice en un ensemble de vecteurs orthogonaux.

10.2.1 Élimination de Gauss, forme échelonnée

Élimination de Gauss et équivalence à gauche. L'élimination de Gauss est l'une des opérations fondamentales en algèbre linéaire car elle permet d'accéder à une représentation de la matrice à la fois plus adaptée au calcul, comme la résolution de systèmes, et révélant certaines de ses propriétés fondamentales, comme le rang, le déterminant, le profil de rang, etc. Les opérations de base pour l'élimination sont les opérations élémentaires sur les lignes :
- permutation de deux lignes : $L_i \leftrightarrow L_j$,
- ajout d'un multiple d'une ligne à une autre : $L_i \leftarrow L_i + sL_j$.

Au niveau matriciel, ces transformations correspondent à la multiplication à gauche par les matrices de transposition $T_{i,j}$ et de transvection $C_{i,j,s}$ données par :

$$T_{i,j} = \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} \begin{array}{c} \begin{matrix} i & & j \end{matrix} \\ \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 0 & & 1 & & \\ & & & \ddots & & & \\ & & 1 & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \end{array}, C_{i,j,s} = \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} \begin{array}{c} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix} \\ \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & \ddots & & & \\ & & s & & 1 & & \\ & & & & & \ddots & \end{bmatrix} \end{array}.$$

Ces matrices ont toutes pour déterminant 1 ou -1. Ainsi toute multiplication à gauche par un produit de ces matrices peut être vu comme un changement de base préservant les volumes et donc en particulier le déterminant. En Sage, l'opération de transvection est effectuée par la méthode `add_multiple_of_row(i,j,s)`, et celle de transposition par la méthode `swap_rows(i,j)`.

Étant donné un vecteur colonne $x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ dont la k-ième composante x_k est inversible, on définit la transformation de Gauss comme la composition des transvections C_{i,k,ℓ_i} pour $i = k+1 \dots n$, avec $\ell_i = -\frac{x_i}{x_k}$ (peu importe l'ordre étant donné qu'elles commutent). La matrice correspondante est la suivante :

$$G_{x,k} = C_{k+1,k,\ell_{k+1}} \times \cdots \times C_{n,k,\ell_n} = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & \ell_{k+1} & \ddots & & & \\ & & \ell_{k+2} & & \ddots & & \\ & & \vdots & & & \ddots & \\ & & \ell_n & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ k \end{matrix}$$

Une transformation de Gauss $G_{x,k}$ a pour effet d'éliminer les coefficients du vecteur situés sous le coefficient pivot x_k :

$$G_{x,k} \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Pour une matrice $A = [a_{i,j}]$ de dimension $m \times n$, l'algorithme du pivot de Gauss procède alors itérativement, de la colonne gauche à la colonne droite. En supposant que les $k-1$ premières colonnes ont déjà été traitées, fournissant $p \leqslant k-1$ pivots, la k-ième est alors traitée de la façon suivante :

- trouver la première composante inversible $a_{i,k}$ de la colonne C_k sur une ligne $i > p$. On l'appelle le pivot.
- Si aucun pivot n'est trouvé, passer à la colonne suivante.
- Appliquer la transposition $T_{i,p+1}$ sur les lignes pour placer le pivot en position $(p+1, k)$.
- Appliquer la transformation de Gauss $G_{x,p+1}$.

Cet algorithme transforme la matrice A en une matrice triangulaire supérieure. Plus précisément elle aura une forme échelonnée, c'est-à-dire telle que le premier coefficient non nul de chaque ligne se trouve plus à droite que celui de la ligne précédente ; de plus, toutes les lignes nulles se trouvent regroupées en bas de la matrice. Voici un exemple de déroulement de cet algorithme.

```
sage: a=matrix(GF(7),4,3,[6,2,2,5,4,4,6,4,5,5,1,3]); a
```

```
[6 2 2]
[5 4 4]
[6 4 5]
[5 1 3]
sage: u=copy(identity_matrix(GF(7),4)); u[:,0]=-a[:,0]/a[0,0];
sage: u, u*a
(
[6 0 0 0]  [1 5 5]
[5 1 0 0]  [0 0 0]
[6 0 1 0]  [0 2 3]
[5 0 0 1], [0 4 6]
)
sage: v=copy(identity_matrix(GF(7),4)); v.swap_rows(1,2);
sage: b=v*u*a; v,b
(
[1 0 0 0]  [1 5 5]
[0 0 1 0]  [0 2 3]
[0 1 0 0]  [0 0 0]
[0 0 0 1], [0 4 6]
)
sage: w=copy(identity_matrix(GF(7),4));
sage: w[1:,1]=-b[1:,1]/b[1,1]; w, w*b
(
[1 0 0 0]  [1 5 5]
[0 6 0 0]  [0 5 4]
[0 0 1 0]  [0 0 0]
[0 5 0 1], [0 0 0]
)
```

Élimination de Gauss-Jordan. La transformation de Gauss-Jordan est similaire à celle de Gauss, en ajoutant à $G_{x,k}$ les transvections correspondant aux lignes d'indice $i < k$; cela revient à éliminer les coefficients d'une colonne au-dessus et au-dessous du pivot. Si de plus on divise chaque ligne par son pivot, on obtient alors une forme échelonnée dite *réduite* encore appelée forme de Gauss-Jordan. Pour toute classe d'équivalence de matrices, il n'existe qu'une unique matrice sous cette forme ; il s'agit donc d'une forme normale.

Définition. Une matrice est dite sous forme échelonnée réduite si
- toutes les lignes nulles sont en bas de la matrice,
- le premier coefficient non nul de chaque ligne non nulle, appelé le pivot, est un 1, et est situé à droite du pivot de la ligne précédente,
- les pivots sont les seuls coefficients non nuls au sein de leur colonne.

Théorème. *Pour toute matrice A de dimension $m \times n$ à coefficients dans un corps, il existe une unique matrice R de dimension $m \times n$ sous forme échelonnée réduite et une matrice inversible U de dimension $m \times m$ telles que $UA = R$. Il s'agit de la décomposition de Gauss-Jordan.*

En Sage, la forme échelonnée réduite est donnée par les méthodes `echelonize` et `echelon_form`. La première remplace la matrice initiale par sa forme échelonnée réduite alors que la deuxième renvoie une matrice immuable sans modifier la matrice initiale.

```
sage: A=matrix(GF(7),4,5,[4,4,0,2,4,5,1,6,5,4,1,1,0,1,0,
....:                     5,1,6,6,2]); A, A.echelon_form()
(
[4 4 0 2 4]  [1 0 5 0 3]
[5 1 6 5 4]  [0 1 2 0 6]
[1 1 0 1 0]  [0 0 0 1 5]
[5 1 6 6 2], [0 0 0 0 0]
)
```

Plusieurs variantes de l'élimination de Gauss s'interprètent sous la forme de différentes décompositions matricielles, parfois utiles pour le calcul : les décompositions $A = LU$ pour les matrices génériques, $A = LUP$, pour les matrices régulières, $A = LSP$, $A = LQUP$ ou $A = PLUQ$ pour les matrices de rang quelconque. Les matrices L sont triangulaires inférieures (en anglais *Lower triangular*), U triangulaires supérieures (*Upper triangular*), et les matrices P, Q sont des permutations. Si ces dernières sont algorithmiquement moins coûteuses que la forme échelonnée réduite, elles n'offrent pas l'avantage de fournir une forme normale.

Forme échelonnée dans les anneaux euclidiens. Dans un anneau euclidien, les coefficients non nuls ne sont pas nécessairement inversibles, et l'élimination de Gauss consisterait donc à choisir le premier élément inversible de la colonne courante pour pivot. Ainsi certaines colonnes non nulles peuvent ne pas contenir de pivot et l'élimination n'est alors plus possible.

Il est cependant toujours possible de définir une transformation unimodulaire éliminant le coefficient de tête d'une ligne avec celui d'une autre, grâce à l'algorithme d'Euclide étendu.

Soit $A = \begin{bmatrix} a & * \\ b & * \end{bmatrix}$ et soit $g = \mathrm{pgcd}(a, b)$. Soient u et v les coefficients de Bézout fournis par l'algorithme d'Euclide étendu appliqué à a et b (tels que $g = ua + vb$), et $s = b/g, t = -a/g$ tels que

$$\begin{bmatrix} u & v \\ s & t \end{bmatrix} \begin{bmatrix} a & * \\ b & * \end{bmatrix} = \begin{bmatrix} g & * \\ 0 & * \end{bmatrix}.$$

Cette transformation est unimodulaire car $\det\left(\begin{bmatrix} u & v \\ s & t \end{bmatrix}\right) = 1$.

Par ailleurs, comme pour Gauss-Jordan, on peut toujours ajouter des multiples de la ligne pivot aux lignes supérieures afin de réduire leurs coefficients dans la même colonne modulo le pivot g. Cette opération effectuée

itérativement sur toutes les colonnes de la matrice produit la forme normale de Hermite.

Définition. Une matrice est dite sous forme de Hermite si
- ses lignes nulles sont en bas,
- le premier coefficient non nul de chaque ligne, appelé le pivot, se trouve à droite de celui de la ligne supérieure,
- tous les coefficients au-dessus du pivot sont réduits modulo le pivot.

Théorème. *Pour toute matrice A de dimension $m \times n$ à coefficients dans un anneau euclidien, il existe une unique matrice H de dimension $m \times n$ sous forme de Hermite et une matrice U unimodulaire, de dimension $m \times m$ telles que $UA = H$.*

Dans le cas d'un corps, la forme de Hermite correspond à la forme échelonnée réduite, ou forme de Gauss-Jordan. En effet, dans ce cas, tous les pivots sont inversibles, chaque ligne peut être divisée par son pivot, et les coefficients au-dessus de celui-ci peuvent être à nouveau éliminés par des transformations de Gauss, produisant ainsi une forme échelonnée réduite. En Sage, il n'y a donc qu'une seule méthode : `echelon_form`, qui retourne soit la forme de Hermite, soit la forme échelonnée réduite, selon que la matrice est à coefficients dans un anneau ou dans un corps.

Par exemple, pour une matrice à coefficients dans $\mathbb{Z}$, on obtient les deux formes échelonnées différentes, selon que le domaine de base soit $\mathbb{Z}$ ou $\mathbb{Q}$:

```
sage: a=matrix(ZZ, 4, 6, [2,1,2,2,2,-1,1,2,-1,2,1,-1,2,1,-1,-1,\
....:                     2,2,2,1,1,-1,-1,-1]); a.echelon_form()
[ 1  2  0  5  4 -1]
[ 0  3  0  2 -6 -7]
[ 0  0  1  3  3  0]
[ 0  0  0  6  9  3]
sage: a.base_extend(QQ).echelon_form()
[   1    0    0    0  5/2 11/6]
[   0    1    0    0   -3 -8/3]
[   0    0    1    0 -3/2 -3/2]
[   0    0    0    1  3/2  1/2]
```

Pour les matrices sur $\mathbb{Z}$, la forme normale de Hermite est aussi accessible par la fonction `hermite_form`. Pour obtenir la matrice de passage U telle que $UA = H$, on peut utiliser l'option `transformation=True`.

```
sage: A=matrix(ZZ,4,5,[4,4,0,2,4,5,1,6,5,4,\
....:                  1,1,0,1,0,5,1,6,6,2])
sage: E,U=A.echelon_form(transformation=True); E,U
(
[ 1  1  0  0  2]  [ 0  1  1 -1]
[ 0  4 -6  0 -4]  [ 0 -1  5  0]
[ 0  0  0  1 -2]  [ 0 -1  0  1]
[ 0  0  0  0  0], [ 1 -2 -4  2]
)
```

Facteurs invariants et forme normale de Smith. Si l'on s'autorise à éliminer plus avant la forme de Hermite par des transformations unimodulaires à droite, on peut alors obtenir une forme diagonale canonique, appelée forme normale de Smith. Ses coefficients diagonaux sont appelés les *facteurs invariants* (en anglais *elementary divisors*) de la matrice. Ils sont totalement ordonnés pour la divisibilité (i.e. $s_i \mid s_{i+1}$).

Théorème. *Pour toute matrice A de dimension $m \times n$ et à coefficients dans un anneau principal, il existe des matrices unimodulaires U et V de dimension $m \times m$ et $n \times n$ et une unique matrice diagonale $S = diag(s_1, \ldots, s_{\min m,n})$ telles que $S = UAV$. Les coefficients s_i vérifient $s_i \mid s_{i+1}$ et sont appelés les facteurs invariants de A.*

En Sage, la méthode `elementary_divisors` renvoie la liste des facteurs invariants. On peut par ailleurs calculer la forme normale de Smith ainsi que les matrices de passages U et V par la commande `smith_form`.

```
sage: A=matrix(ZZ,4,5,[-1,-1,-1,-2,-2,-2,1,1,-1,2,2,2,2,2,-1,\
....:                  2,2,2,2,2])
sage: A.smith_form()
(
                              [ 0 -2 -1 -5  0]
[1 0 0 0 0]  [ 1  0  0  0]  [ 1  0  1 -1 -1]
[0 1 0 0 0]  [ 0  0  1  0]  [ 0  0  0  0  1]
[0 0 3 0 0]  [-2  1  0  0]  [-1  2  0  5  0]
[0 0 0 6 0], [ 0  0 -2 -1], [ 0 -1  0 -2  0]
)
sage: S,U,V=A.smith_form(); S
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 3 0 0]
[0 0 0 6 0]
sage: A.elementary_divisors()
[1, 1, 3, 6]
sage: S==U*A*V
True
```

Rang, profil de rang et pivots. L'élimination de Gauss révèle de nombreux invariants de la matrice, tels que son rang et son déterminant (qui peut être lu comme le produit des pivots). Ils sont accessibles par les fonctions `det` et `rank`. Ces valeurs seront mises en cache, et ne seront donc pas recalculées lors d'un deuxième appel.

Plus généralement, la notion de profil de rang est très utile, lorsque l'on considère la matrice comme une séquence de vecteurs.

Définition. Le profil de rang par colonne d'une matrice $m \times n$ A de rang r est la séquence de r indices lexicographiquement minimale, telle que les colonnes correspondantes dans A sont linéairement indépendantes.

Le profil de rang se lit directement sur la forme échelonnée réduite, comme la séquence des indices des pivots. Il est calculé par la fonction `pivots`. Lorsque la forme échelonnée réduite a déjà été calculée, le profil de rang est aussi mémorisé dans le cache, et peut être obtenu sans calcul supplémentaire.

Le profil de rang par ligne se définit de manière similaire, en considérant la matrice comme une séquence de m vecteurs ligne. Il s'obtient par la commande `pivot_rows` ou comme sous-produit de la forme échelonnée réduite de la matrice transposée.

```
sage: B=matrix(GF(7),5,4,[4,5,1,5,4,1,1,1,0,6,0,6,2,5,1,6,\
....:                     4,4,0,2])
sage: B.transpose().echelon_form()
[1 0 5 0 3]
[0 1 2 0 6]
[0 0 0 1 5]
[0 0 0 0 0]
sage: B.pivot_rows()
[0, 1, 3]
sage: B.transpose().pivots() == B.pivot_rows()
True
```

10.2.2 Résolution de systèmes ; image et base du noyau

Résolution de systèmes. Un système linéaire peut être représenté par une matrice A et un vecteur b soit à droite : $Ax = b$, soit à gauche : ${}^t x\, A = b$. Les fonctions `solve_right` et `solve_left` effectuent leur résolution. On peut aussi utiliser de façon équivalente les opérateurs `A\b` et `b/A`. Lorsque le système est donné par une matrice à coefficients dans un anneau, la résolution est systématiquement effectuée dans le corps des fractions de cet anneau (e.g., $\mathbb{Q}$ pour $\mathbb{Z}$ ou $K(X)$ pour $K[X]$). On verra plus loin comment la résolution peut être faite dans l'anneau lui-même. Le membre de droite dans l'égalité du système peut être aussi bien un vecteur qu'une matrice (ce qui correspond à résoudre plusieurs systèmes linéaires simultanément, avec la même matrice).

Les matrices des systèmes peuvent être rectangulaires, et admettre une unique, aucune ou une infinité de solutions. Dans ce dernier cas, les fonctions `solve` renvoient l'une de ces solutions, en mettant à zéro les composantes linéairement indépendantes.

```
sage: R.<x>=PolynomialRing(GF(5),'x')
sage: A=random_matrix(R,2,3); A
[      3*x^2 + x       x^2 + 2*x       2*x^2 + 2]
[    x^2 + x + 2 2*x^2 + 4*x + 3   x^2 + 4*x + 3]
sage: b=random_matrix(R,2,1); b
[  4*x^2 + 1]
[3*x^2 + 2*x]
sage: A.solve_right(b)
```

```
[(4*x^3 + 2*x + 4)/(3*x^3 + 2*x^2 + 2*x)]
[  (3*x^2 + 4*x + 3)/(x^3 + 4*x^2 + 4*x)]
[                                     0]
```

Image et noyau. Interprétée comme une application linéaire Φ, une matrice A de dimension $m \times n$ définit deux sous-espaces vectoriels de K^m et K^n, respectivement l'image et le noyau de Φ.

L'image est l'ensemble des vecteurs de K^m obtenus par combinaisons linéaires des colonnes de A. Il s'obtient par la fonction `image` qui renvoie un espace vectoriel dont la base est sous forme échelonnée réduite.

Le noyau est le sous-espace vectoriel de K^n des vecteurs x tels que $Ax = 0$. Obtenir une base de ce sous-espace sert en particulier à décrire l'ensemble des solutions d'un système linéaire, lorsque celui-ci en possède une infinité : si $\overline{x}$ est une solution de $Ax = b$ et V le noyau de A, alors l'ensemble des solutions s'écrit simplement $\overline{x} + V$. Il s'obtient par la fonction `right_kernel` qui renvoie l'espace vectoriel ainsi qu'une base mise sous forme échelonnée réduite. On peut naturellement aussi définir le noyau à gauche (l'ensemble des x dans K^m tels que ${}^t x\, A = 0$), qui correspond au noyau à droite de la transposée de A (i.e., l'adjoint de Φ). Il s'obtient avec la fonction `left_kernel`. Par convention, la fonction `kernel` retourne le noyau à gauche. De plus les bases sont données comme des matrices de vecteurs lignes dans les deux cas.

```
sage: a=matrix(QQ,3,5,[2,2,-1,-2,-1,2,-1,1,2,-1/2,2,-2,\
....:                  -1,2,-1/2])
sage: a.image()
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[     1      0      0    1/4 -11/32]
[     0      1      0     -1   -1/8]
[     0      0      1    1/2   1/16]
sage: a.right_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[    1     0     0  -1/3   8/3]
[    0     1  -1/2 11/12   2/3]
```

La notion de noyau se généralise naturellement au cas où les coefficients ne sont plus dans un corps ; il s'agit alors d'un module libre. En particulier, pour une matrice définie dans un corps de fraction, on obtiendra le noyau dans l'anneau de base par la commande `integer_kernel`. Par exemple, pour une matrice à coefficients dans $\mathbb{Z}$, plongée dans l'espace vectoriel des matrices à coefficients dans $\mathbb{Q}$, on pourra calculer aussi bien son noyau comme un sous-espace vectoriel de $\mathbb{Q}^m$ ou comme un module libre de $\mathbb{Z}^m$.

```
sage: a=matrix(ZZ,5,3,[1,1,122,-1,-2,1,-188,2,1,1,-10,1,\
....:                  -1,-1,-1])
sage: a.kernel()
Free module of degree 5 and rank 2 over Integer Ring
Echelon basis matrix:
[   1  979  -11 -279  811]
[   0 2079  -22 -569 1488]
sage: b=a.base_extend(QQ)
sage: b.kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[        1         0  -121/189 -2090/189   6949/63]
[        0         1    -2/189 -569/2079   496/693]
sage: b.integer_kernel()
Free module of degree 5 and rank 2 over Integer Ring
Echelon basis matrix:
[   1  979  -11 -279  811]
[   0 2079  -22 -569 1488]
sage: b.integer_kernel() == a.kernel()
True
```

10.2.3 Valeurs propres, forme de Jordan et transformations de similitude

Lorsque l'on interprète une matrice carrée comme un opérateur linéaire (un endomorphisme), elle n'en est que la représentation dans une base donnée. Tout changement de base correspond à une transformation de similitude $B = U^{-1}AU$ de la matrice. Les deux matrices A et B sont alors dites *semblables*. Ainsi les propriétés de l'opérateur linéaire, qui sont indépendantes de la base, sont révélées par l'étude des invariants de similitude de la matrice.

Parmi ces invariants, les plus simples sont le rang et le déterminant. En effet les matrices U et U^{-1} étant inversibles, le rang de $U^{-1}AU$ égale le rang de A. De plus $\det(U^{-1}AU) = \det(U^{-1})\det(A)\det(U) = \det(U^{-1}U)\det(A) = \det(A)$. De la même façon, le polynôme caractéristique de la matrice A, défini par $\chi_A(x) = \det(x\mathrm{Id} - A)$ est aussi invariant par transformation de similitude :

$$\det(x\mathrm{Id} - U^{-1}AU) = \det(U^{-1}(x\mathrm{Id} - A)U) = \det(x\mathrm{Id} - A).$$

Par conséquent, les valeurs caractéristiques d'une matrice, définies comme les racines du polynôme caractéristique dans son corps de décomposition, sont donc aussi des invariants de similitude. Par définition, un scalaire λ est une valeur propre d'une matrice A s'il existe un vecteur non nul u tel que $Au = \lambda u$. L'espace propre associé à une valeur propre λ est l'ensemble des vecteurs u tels que $Au = \lambda u$. C'est un sous-espace vectoriel défini par $E_\lambda = \mathrm{Ker}(\lambda\mathrm{Id} - A)$.

Élimination de Gauss et applications	
transvection sur les lignes	`add_multiple_of_row(i,j,s)`
transvection sur les colonnes	`add_multiple_of_column(i,j,s)`
transposition de lignes, colonnes	`swap_rows(i,j)`, `swap_columns(i,j)`
forme de Gauss-Jordan, immuable	`echelon_form()`
facteurs invariants	`elementary_divisors`
forme normale de Smith	`smith_form`
forme de Gauss-Jordan en place	`echelonize()`
déterminant, rang	`det`, `rank`
mineurs d'ordre k	`minors(k)`
profil de rang en colonne, en ligne	`pivots`, `pivot_rows`
résolution de système à gauche	`A\b` *ou* `A.solve_left(b)`
résolution de système à droite	`b/A` *ou* `A.solve_right(b)`
espace image	`image`
noyau à gauche	`kernel` *ou* `left_kernel`
noyau à droite	`right_kernel`
noyau dans l'anneau de base	`integer_kernel`

Décomposition spectrale	
polynôme minimal	`minimal_polynomial` *ou* `minpoly`
polynôme caractéristique	`characteristic_polynomial` *ou* `charpoly`
itérés de Krylov à gauche	`maxspin(v)`
valeurs propres	`eigenvalues`
vecteurs propres à gauche, à droite	`eigenvectors_left`, `eigenvectors_right`
espaces propres à gauche, à droite	`eigenspaces_left`, `eigenspaces_right`
diagonalisation	`eigenmatrix_left`, `eigenmatrix_right`
bloc de Jordan $J_{a,k}$	`jordan_block(a,k)`

Tableau 10.3 – Calculs sur les matrices.

Les valeurs propres coïncident avec les valeurs caractéristiques :

$$\det(\lambda \mathrm{Id} - A) = 0 \Leftrightarrow \dim(\mathrm{Ker}(\lambda \mathrm{Id} - A)) \geqslant 1 \Leftrightarrow \exists u \neq 0, \lambda u - Au = 0.$$

Ces deux points de vue correspondent respectivement à l'approche algébrique et géométrique des valeurs propres. Dans le point de vue géométrique, on s'intéresse à l'action de l'opérateur linéaire A sur les vecteurs de l'espace avec plus de précision que dans le point de vue algébrique. En particulier on distingue les notions de multiplicité algébrique, correspondant à l'ordre de la racine dans le polynôme caractéristique, de la multiplicité géométrique, correspondant à la dimension du sous-espace propre associé à la valeur propre. Pour les matrices diagonalisables, ces deux notions sont équivalentes. Dans le cas contraire, la multiplicité géométrique est toujours inférieure à la multiplicité algébrique.

Le point de vue géométrique permet de décrire plus en détail la structure de la matrice. Par ailleurs, il donne des algorithmes beaucoup plus rapides pour le calcul des valeurs et espaces propres, et des polynômes caractéristique et minimal.

Espaces invariants cycliques, et forme normale de Frobenius. Soit A une matrice $n \times n$ sur un corps K et u un vecteur de K^n. La famille de vecteurs $u, Au, A^2u, \ldots, A^nu$, appelée séquence de Krylov, est liée (comme famille de $n+1$ vecteurs en dimension n). Soit d tel que A^du soit le premier vecteur de la séquence linéairement dépendant avec ses prédécesseurs $u, Au, \ldots, A^{d-1}u$. On écrira

$$A^du = \sum_{i=0}^{d-1} \alpha_i A^iu$$

cette relation de dépendance linéaire. Le polynôme $\varphi_{A,u}(x) = x^d - \sum_{i=0}^{d-1} \alpha_i x^i$, qui vérifie $\varphi_{A,u}(A)u = 0$ est donc un polynôme unitaire annulateur de la séquence de Krylov et de degré minimal. On l'appelle le *polynôme minimal* du vecteur u (sous entendu, relativement à la matrice A). L'ensemble des polynômes annulateurs de u forme un idéal de $K[X]$, engendré par $\varphi_{A,u}$.

Le polynôme minimal de la matrice A est défini comme le polynôme unitaire $\varphi_A(x)$ de plus petit degré annulant la matrice A : $\varphi_A(A) = 0$. En particulier, en appliquant φ_A à droite sur le vecteur u, on constate que φ_A est un polynôme annulateur de la séquence de Krylov. Il est donc nécessairement un multiple du polynôme minimal de u. On peut en outre montrer (cf. exercice 34) qu'il existe un vecteur $\overline{u}$ tel que

$$\varphi_{A,\overline{u}} = \varphi_A. \tag{10.1}$$

Lorsque le vecteur u est choisi aléatoirement, la probabilité qu'il satisfasse l'équation (10.1) est d'autant plus grande que la taille du corps est grande (on peut montrer qu'elle est au moins de $1 - \frac{n}{|K|}$).

Exercice 34. Montrons qu'il existe toujours un vecteur $\overline{u}$ dont le polynôme minimal coïncide avec le polynôme minimal de la matrice.

1. Soit $(e_1, \ldots, e_n)$ une base de l'espace vectoriel. Montrer que φ_A coïncide avec le ppcm des φ_{A,e_i}.
2. Dans le cas particulier où φ_A est une puissance d'un polynôme irréductible, montrer qu'il existe un indice i_0 tel que $\varphi_A = \varphi_{A,e_{i_0}}$.
3. Montrer que si les polynômes minimaux $\varphi_i = \varphi_{A,e_i}$ et $\varphi_j = \varphi_{A,e_j}$ des vecteurs e_i et e_j sont premiers entre eux, alors $\varphi_{A,e_i+e_j} = \varphi_i\varphi_j$.
4. Montrer que si $\varphi_A = P_1P_2$ où P_1 et P_2 sont premiers entre eux, alors il existe un vecteur $x_i \neq 0$ tels que P_i soit le polynôme minimal de x_i.
5. Conclure en utilisant la factorisation en polynômes irréductibles de $\varphi_A = \varphi_1^{m_1} \ldots \varphi_k^{m_k}$.
6. Illustration : soit $A = \begin{bmatrix} 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 6 & 0 & 0 \\ 0 & 1 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 1 & 5 \end{bmatrix}$ une matrice dans GF(7). Calculer les degrés du polynôme minimal de A, et des polynômes minimaux

des vecteurs de la base canonique $u = e_1$ et $v = e_4$, ainsi que $u + v$. On peut se servir de la fonction `maxspin(u)` appliquée à la transposée de A, qui retourne la séquence maximale des itérés de Krylov d'un vecteur u.

Soit $P = x^k + \sum_{i=0}^{k-1} \alpha_i x^i$ un polynôme unitaire de degré k. La matrice compagnon associée au polynôme P est la matrice $k \times k$ définie par

$$C_P = \begin{bmatrix} 0 & & & -\alpha_0 \\ 1 & & & -\alpha_1 \\ & \ddots & & \vdots \\ & & 1 & -\alpha_{k-1} \end{bmatrix}.$$

Cette matrice a la propriété d'avoir P pour polynôme minimal et caractéristique. Elle joue ainsi un grand rôle dans le calcul des polynômes minimal et caractéristique.

Proposition. *Soit K_u la matrice formée par les d premiers itérés de Krylov d'un vecteur u. Alors*

$$AK_u = K_u C_{\varphi_{A,u}}$$

Ainsi, lorsque $d = n$, la matrice K_u est carrée d'ordre n et inversible. Elle définit une transformation de similitude $K_u^{-1} A K_u = C_{\varphi_{A,u}}$ réduisant la matrice A à une matrice compagnon. Or cette transformation préserve le déterminant, et donc le polynôme caractéristique ; on pourra ainsi lire directement les coefficients du polynôme minimal et caractéristique (ici identiques) sur la matrice compagnon.

```
sage: A=matrix(GF(97),4,4,[86,1,6,68,34,24,8,35,15,36,68,42,\
....:                      27,1,78,26])
sage: e1=identity_matrix(GF(97),4)[0]
sage: U=matrix(A.transpose().maxspin(e1)).transpose()
sage: F=U^-1*A*U; F
[ 0  0  0 83]
[ 1  0  0 77]
[ 0  1  0 20]
[ 0  0  1 10]
sage: K.<x>=GF(97)[]
sage: P=x^4-sum(F[i,3]*x^i for i in range(4)); P
x^4 + 87*x^3 + 77*x^2 + 20*x + 14
sage: P==A.charpoly()
True
```

Dans le cas général, $d \leqslant n$; les vecteurs itérés $u, \ldots, A^{d-1}u$ forment une base d'un sous-espace I invariant sous l'action de la matrice A (i.e., tel que $AI \subseteq I$). Comme chacun de ces vecteurs est obtenu cycliquement en appliquant la matrice A au vecteur précédent, on l'appelle aussi sous-espace cyclique. La dimension maximale d'un tel sous-espace est le degré du polynôme minimal

de la matrice. Il est engendré par les itérés de Krylov du vecteur construit dans l'exercice 34, qu'on notera u_1^*. On l'appelle le premier sous-espace invariant. Ce premier espace invariant admet un espace supplémentaire V. En calculant *modulo* le premier espace invariant, c'est-à-dire en considérant que deux vecteurs sont égaux si leur différence appartient au premier sous-espace invariant, on peut définir un second sous-espace invariant pour les vecteurs dans cet espace supplémentaire ainsi qu'un polynôme minimal qui est appelé le second invariant de similitude. On obtiendra alors une relation de la forme :

$$A \begin{bmatrix} K_{u_1^*} & K_{u_2^*} \end{bmatrix} = \begin{bmatrix} K_{u_1^*} & K_{u_2^*} \end{bmatrix} \begin{bmatrix} C_{\varphi_1} & \\ & C_{\varphi_2} \end{bmatrix},$$

où φ_1, φ_2 sont les deux premiers invariants de similitude, et $K_{u_1^*}, K_{u_2^*}$ sont les matrices de Krylov correspondant aux deux espaces cycliques engendrés par les vecteurs u_1^* et u_2^*.

Itérativement, on construit une matrice $U = \begin{bmatrix} K_{u_1^*} & \dots & K_{u_k^*} \end{bmatrix}$ carrée, inversible, telle que

$$K^{-1}AK = \begin{bmatrix} C_{\varphi_1} & & \\ & \ddots & \\ & & C_{\varphi_k} \end{bmatrix}. \tag{10.2}$$

Comme chaque u_i^* est annulé par les φ_j pour $j \leqslant i$ on en déduit que $\varphi_i \mid \varphi_{i-1}$ pour tout $2 \leqslant i \leqslant k$, autrement dit, la suite des φ_i est totalement ordonnée pour la division. On peut montrer que pour toute matrice, il n'existe qu'une unique séquence de polynômes invariants $\varphi_1, \dots, \varphi_k$. Ainsi la matrice diagonale par blocs $\mathrm{Diag}(C_{\varphi_1}, \dots, C_{\varphi_k})$, semblable à la matrice A et révélant ces polynômes, est une forme normale, appelée forme rationnelle canonique, ou forme normale de Frobenius.

Théorème (Forme normale de Frobenius)**.** *Toute matrice carrée A dans un corps est semblable à une unique matrice $F = \begin{bmatrix} C_{\varphi_1} & & \\ & \ddots & \\ & & C_{\varphi_k} \end{bmatrix}$, avec $\varphi_{i+1} \mid \varphi_i$ pour tout $i < k$.*

D'après l'équation (10.2), il apparaît qu'on peut lire les bases des sous-espaces invariants sur la matrice de passage K.

Remarque. Le théorème de Cayley-Hamilton énonce que le polynôme caractéristique annule sa matrice : $\chi_A(A) = 0$. Il se montre simplement, après l'introduction de cette forme normale de Frobenius. En effet,

$$\begin{aligned} \chi_A(x) &= \det(x\mathrm{Id} - A) = \det(U^{-1}) \det(x\mathrm{Id} - F) \det(U) \\ &= \prod_{i=1}^{k} \det(x\mathrm{Id} - C_{\varphi_i}) = \prod_{i=1}^{k} \varphi_i(x). \end{aligned}$$

Ainsi, le polynôme minimal φ_1 est un diviseur du polynôme caractéristique, qui est donc annulateur de la matrice A.

En Sage, on pourra calculer la forme normale de Frobenius dans $\mathbb{Q}$ de matrices à coefficients dans $\mathbb{Z}$ avec la méthode `frobenius`[1] :

```
sage: A=matrix(ZZ,8,[[6,0,-2,4,0,0,0,-2],[14,-1,0,6,0,-1,-1,1],\
....:               [2,2,0,1,0,0,1,0],[-12,0,5,-8,0,0,0,4],\
....:               [0,4,0,0,0,0,4,0],[0,0,0,0,1,0,0,0],\
....:               [-14,2,0,-6,0,2,2,-1],[-4,0,2,-4,0,0,0,4]])
sage: A.frobenius()
[0 0 0 4 0 0 0 0]
[1 0 0 4 0 0 0 0]
[0 1 0 1 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 4 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 1 0]
[0 0 0 0 0 0 0 2]
```

On peut obtenir par ailleurs la liste de polynômes invariants en passant `1` en argument. Pour obtenir l'information sur les espaces invariants associés, on passe l'argument `2` qui produira la matrice de passage K. Elle fournit une base de l'espace total, décomposée en la somme directe des espaces invariants.

```
sage: A.frobenius(1)
[x^4 - x^2 - 4*x - 4, x^3 - x^2 - 4, x - 2]
sage: F,K=A.frobenius(2)
sage: K
[     1    -1/2    1/16   15/64   3/128    7/64  -23/64  43/128]
[     0       0   -5/64 -13/128 -15/256  17/128  -7/128  53/256]
[     0       0   9/128 -11/128  -7/128   -1/32   5/128    5/32]
[     0       0  -5/128       0   7/256  -7/128   -1/64   9/256]
[     0       1    1/16    5/32  -17/64   -1/32   31/32  -21/64]
[     0       0    1/32    5/64  31/128  -17/64   -1/64 -21/128]
[     0       0    1/32    5/64  -1/128   15/64   -1/64 -21/128]
[     0       0       1     5/2    -1/4    -1/2    -1/2   -21/4]
sage: K^-1*F*K==A
True
```

Ces résultats sous-entendent que la matrice A à coefficients dans $\mathbb{Z}$ a été plongée dans son corps de fractions $\mathbb{Q}$. Pour étudier l'action de la matrice A sur le module libre $\mathbb{Z}^n$, et la décomposition du module qu'elle engendre, on utilise la fonction `decomposition` ; cependant son étude dépasse le cadre de cet ouvrage.

[1] C'est une légère aberration de l'interface actuelle du logiciel : alors que la forme de Frobenius est définie pour toute matrice dans un corps, Sage ne permet de la calculer que pour les matrices à coefficients dans $\mathbb{Z}$, en effectuant implicitement le plongement dans $\mathbb{Q}$.

Facteurs invariants et invariants de similitude. Une propriété importante relie les invariants de similitude et les facteurs invariants vus dans la section 10.2.1.

Théorème. *Les invariants de similitude d'une matrice à coefficients dans un corps correspondent aux facteurs invariants de sa matrice caractéristique.*

La preuve de ce résultat dépasse le cadre de cet ouvrage et nous nous contentons de l'illustrer sur l'exemple précédent.

```
sage: S.<x>=QQ[]
sage: B=x*identity_matrix(8)-A
sage: B.elementary_divisors()
[1, 1, 1, 1, 1, x - 2, x^3 - x^2 - 4, x^4 - x^2 - 4*x - 4]
sage: A.frobenius(1)
[x^4 - x^2 - 4*x - 4, x^3 - x^2 - 4, x - 2]
```

Valeurs propres, vecteurs propres. Si l'on décompose le polynôme minimal en facteurs irréductibles, $\varphi_1 = \psi_1^{m_1} \dots \psi_s^{m_s}$, alors tous les facteurs invariants s'écrivent sous la forme $\varphi_i = \psi_1^{m_{i,1}} \dots \psi_s^{m_{i,s}}$, avec des multiplicités $m_{i,j} \leqslant m_j$. On montre que l'on peut alors trouver une transformation de similitude qui change chaque bloc compagnon C_{φ_i} de la forme de Frobenius en un bloc diagonal $\mathrm{Diag}(C_{\psi_1^{m_{i,1}}}, \dots, C_{\psi_s^{m_{i,s}}})$. Cette variante de la forme de Frobenius, que l'on appelle forme intermédiaire, est toujours formée de blocs compagnons, mais cette fois correspondant chacun à une puissance d'un polynôme irréductible.

$$F = \begin{bmatrix} \boxed{\begin{matrix} C_{\psi_1^{m_{1,1}}} & & \\ & \ddots & \\ & & C_{\psi_s^{m_{1,s}}} \end{matrix}} & & & \\ & \boxed{\begin{matrix} C_{\psi_1^{m_{2,1}}} & & \\ & \ddots & \end{matrix}} & & \\ & & \ddots & \\ & & & \boxed{\begin{matrix} C_{\psi_1^{m_{k,1}}} & & \\ & \ddots & \end{matrix}} \end{bmatrix} \tag{10.3}$$

Lorsqu'un facteur irréductible ψ_i est de degré 1 et de multiplicité 1, son bloc compagnon est une matrice 1×1 sur la diagonale et correspond ainsi à une valeur propre. Lorsque le polynôme minimal est scindé et sans carré, la matrice est donc diagonalisable.

On obtiendra les valeurs propres par la méthode `eigenvalues`. La liste des vecteurs propres à droite (respectivement à gauche), associés à leur valeur propre et sa multiplicité est donnée par la méthode `eigenvectors_right` (respectivement `eigenvectors_left`). Enfin les espaces propres, ainsi que leur

base de vecteurs propres, sont fournis par les méthodes `eigenspaces_right` et `eigenspaces_left`.

```
sage: A=matrix(GF(7),4,[5,5,4,3,0,3,3,4,0,1,5,4,6,0,6,3])
sage: A.eigenvalues()
[4, 1, 2, 2]
sage: A.eigenvectors_right()
[(4, [
(1, 5, 5, 1)
], 1), (1, [
(0, 1, 1, 4)
], 1), (2, [
(1, 3, 0, 1),
(0, 0, 1, 1)
], 2)]
sage: A.eigenspaces_right()
[
(4, Vector space of degree 4 and dimension 1 over Finite Field
of size 7
User basis matrix:
[1 5 5 1]),
(1, Vector space of degree 4 and dimension 1 over Finite Field
of size 7
User basis matrix:
[0 1 1 4]),
(2, Vector space of degree 4 and dimension 2 over Finite Field
of size 7
User basis matrix:
[1 3 0 1]
[0 0 1 1])
]
sage: A.eigenmatrix_right()
(
[4 0 0 0]  [1 0 1 0]
[0 1 0 0]  [5 1 3 0]
[0 0 2 0]  [5 1 0 1]
[0 0 0 2], [1 4 1 1]
)
```

Forme de Jordan. Lorsque le polynôme minimal est scindé mais ayant des facteurs avec des multiplicités supérieures à 1, la forme intermédiaire (10.3) n'est pas diagonale. On montre alors qu'il n'existe pas de transformation de similitude la rendant diagonale, elle n'est donc pas diagonalisable. On peut en revanche la trigonaliser, c'est-à-dire la rendre triangulaire supérieure, telle que les valeurs propres apparaissent sur la diagonale. Parmi les différentes matrices triangulaires possibles, la plus réduite de toutes est la forme normale de Jordan.

Un bloc de Jordan $J_{\lambda,k}$, associé à la valeur propre λ et l'ordre k, est la

matrice $J_{\lambda,k}$ de dimensions $k \times k$ donnée par

$$J_{\lambda,k} = \begin{bmatrix} \lambda & 1 & & \\ & \ddots & \ddots & \\ & & \lambda & 1 \\ & & & \lambda \end{bmatrix}.$$

Cette matrice joue un rôle similaire à celui des blocs compagnons, en révélant plus précisément la multiplicité d'un polynôme. En effet, son polynôme caractéristique vaut $\chi_{J_{\lambda,k}} = (X-\lambda)^k$. De plus son polynôme minimal vaut aussi $\varphi_{J_{\lambda,k}} = (X-\lambda)^k$, en effet, il est nécessairement un multiple de $P = X-\lambda$. Or

$$P(J_{\lambda,k}) = \begin{bmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ & & & 0 \end{bmatrix}$$

est une matrice nilpotente d'ordre k d'où $\varphi_{J_{\lambda,k}} = \chi_{J_{\lambda,k}} = (X-\lambda)^k$. La forme normale de Jordan correspond ensuite à la forme intermédiaire (10.3), où les blocs compagnons des $\psi_j^{m_{i,j}}$ ont été remplacés par les blocs de Jordan $J_{\lambda_j,m_{i,j}}$ (on rappelle que, le polynôme minimal étant scindé, les ψ_j s'écrivent sous la forme $X-\lambda_j$).

Ainsi lorsque son polynôme minimal est scindé, toute matrice est semblable à une matrice de Jordan de la forme

$$J = \begin{bmatrix} \boxed{\begin{matrix} J_{\lambda_1,m_{1,1}} & & \\ & \ddots & \\ & & J_{\lambda_s,m_{1,s}} \end{matrix}} & & & \\ & \boxed{\begin{matrix} J_{\lambda_1,m_{2,1}} & \\ & \ddots \end{matrix}} & & \\ & & \ddots & \\ & & & \boxed{\begin{matrix} J_{\lambda_1,m_{k,1}} & \\ & \ddots \end{matrix}} \end{bmatrix}. \tag{10.4}$$

En particulier, dans tout corps algébriquement clos, comme $\mathbb{C}$, la forme de Jordan d'une matrice est toujours définie.

En Sage, le constructeur `jordan_block(a,k)` produit le bloc de Jordan $J_{a,k}$. On obtiendra la forme normale de Jordan, si elle existe, par la méthode `jordan_form`. L'option `transformation=True` permet d'obtenir la matrice de transformation U telle que $U^{-1}AU$ est sous forme de Jordan.

```
sage: A=matrix(ZZ,4,[3,-1,0,-1,0,2,0,-1,1,-1,2,0,1,-1,-1,3])
sage: A.jordan_form()
[3|0|0 0]
[-+-+---]
[0|3|0 0]
[-+-+---]
[0|0|2 1]
[0|0|0 2]
sage: J,U=A.jordan_form(transformation=True)
sage: U^-1*A*U==J
True
```

La forme de Jordan est unique à une permutation des blocs de Jordan près. Selon les ouvrages, on peut imposer ou non que leur ordre d'apparition sur la diagonale respecte l'ordre des polynômes invariants, comme dans l'équation (10.4). On remarque dans l'exemple ci-dessus que Sage ne respecte pas cet ordre, puisque le premier polynôme invariant (le polynôme minimal) est le polynôme $(X-3)(X-2)^2$.

Forme normale primaire. Pour être complet, il faut mentionner une dernière forme normale qui généralise la forme de Jordan dans le cas quelconque où le polynôme minimal n'est pas scindé. Pour un polynôme irréductible P de degré k, on définit le bloc de Jordan de multiplicité m comme la matrice $J_{P,m}$ de dimension $km \times km$ vérifiant

$$J_{P,m} = \begin{bmatrix} C_P & B & & \\ & \ddots & \ddots & \\ & & C_P & B \\ & & & C_P \end{bmatrix}$$

où B est la matrice $k \times k$ dont le seul coefficient non nul est $B_{k,1} = 1$. On note que si $P = X - \lambda$, on retrouve la notion de bloc de Jordan associé à la valeur propre λ. On montre de façon similaire que les polynômes minimal et caractéristique de cette matrice valent

$$\chi_{J_{P,m}} = \varphi_{J_{P,m}} = P^m.$$

Ainsi on montre qu'il existe une transformation de similitude remplaçant chaque bloc compagnon $C_{\psi_i^{m_{i,j}}}$ de la forme intermédiaire (10.3) en un bloc de Jordan $J_{\psi_i,m_{i,j}}$. La matrice ainsi formée est appelée la forme primaire ou encore la deuxième forme de Frobenius. Il s'agit là encore d'une forme normale, c'est-à-dire unique à une permutation des blocs diagonaux près.

L'unicité de ces formes normales permet en particulier de tester si deux matrices sont semblables, et par la même occasion de produire une matrice de passage entre l'une et l'autre.

Exercice 35. Écrire un programme qui détermine si deux matrices A et B sont semblables et renvoie la matrice U de passage telle que $A = U^{-1}BU$ (on pourra renvoyer `None` dans le cas où les matrices ne sont pas semblables).

Pour résoudre cette équation différentielle, regardez-la jusqu'à ce que la solution vienne d'elle-même.

George Pólya (1887 - 1985)

11 Équations différentielles

11.1 Introduction

Si la méthode de George Pólya semble peu efficace, on peut alors appeler Sage à la rescousse. On peut, si on le souhaite, l'invoquer afin d'obtenir une étude qualitative : en effet, ses outils numériques et graphiques guideront l'intuition. C'est l'objet de la section 6.2 du chapitre consacré au calcul numérique.

On peut préférer résoudre les équations différentielles exactement : Sage peut alors *parfois* y aider en donnant directement une réponse formelle comme nous le verrons dans ce chapitre.

Dans la plupart des cas, il faudra passer par une manipulation savante de ces équations pour aider Sage. Il faudra veiller dans ces deux derniers cas à garder en tête que la solution attendue d'une équation différentielle est une *fonction* dérivable sur un certain intervalle mais que Sage, lui, manipule des *expressions* sans domaine de définition. La machine aura donc besoin d'une intervention humaine pour aller vers une solution rigoureuse.

Nous étudierons dans ce chapitre les généralités sur les équations différentielles ordinaires d'ordre 1 puis quelques cas particuliers comme les équations linéaires, les équations à variables séparables, les équations homogènes, une équation dépendant d'un paramètre puis de manière plus sommaire les équations d'ordre 2 ainsi qu'un exemple d'équation aux dérivées partielles. Nous terminerons par l'utilisation de la transformée de Laplace.

11.2 Équations différentielles ordinaires d'ordre 1

11.2.1 Commandes de base

On commence par définir une variable t et une fonction x dépendant de cette variable :

```
sage: t = var('t')
sage: x = function('x', t)
```

On utilise ensuite :

```
desolve(equation,variable,ics=...,ivar=...,show_method=...,
        contrib_ode=...)
```

Où :
- `equation` est l'équation différentielle. L'égalité est symbolisée par `==`. Par exemple l'équation $x' = 2x + t$ s'écrit `diff(x,t)==2*x+t` ;
- `variable` est le nom de la variable dépendante, c'est-à-dire la fonction x dans $x' = 2x + t$;
- `ics` est un argument optionnel qui permet d'entrer des conditions initiales. Pour une équation du premier ordre, on entre une liste $[t_0,x_0]$, pour les équations du second ordre c'est $[t_0,x_0,t_1,x_1]$ ou $[t_0,x_0,x'_0]$;
- `ivar` est un argument optionnel qui permet de préciser la variable indépendante, c'est-à-dire t dans $x' = 2x + t$. Cet argument doit absolument être précisé en cas d'équations dépendant de paramètres comme par exemple $x' = ax + bt + c$;
- `show_method` est un argument optionnel fixé à `False` par défaut. Dans le cas contraire, il demande à Sage de préciser la méthode de résolution utilisée. Les termes anglais renvoyés sont `linear`, `separable`, `exact`, `homogeneous`, `bernoulli`, `generalized homogeneous`. Sage renvoie alors une liste dont le premier argument est la solution et le deuxième la méthode ;
- `contrib_ode` est un argument optionnel par défaut fixé à `False`. Dans le cas contraire, `desolve` pourra s'occuper des équations de Ricatti, Lagrange, Clairaut et d'autres cas pathologiques.

11.2.2 Équations du premier ordre pouvant être résolues directement par Sage

Voici un rapide aperçu des équations pouvant être résolues directement par Sage :

Équations linéaires Il s'agit d'équations du type :

$$y' + P(x)y = Q(x)$$

avec P et Q des fonctions continues sur des intervalles donnés.
Par exemple : $y' + 3y = \mathrm{e}^x$.

```
sage: desolve(diff(y,x)+3*y==exp(x),y,show_method=True)
[1/4*(4*c + e^(4*x))*e^(-3*x), 'linear']
```

Équations à variables séparables Il s'agit d'équations du type :

$$P(x) = y'Q(y)$$

avec P et Q des fonctions continues sur des intervalles donnés.
Par exemple : $yy' = x$.

```
sage: desolve(y*diff(y,x)==x,y,show_method=True)
[1/2*y(x)^2 == 1/2*x^2 + c, 'separable']
```

Attention ! Sage parfois ne reconnaît pas les équations à variables séparables et les traite comme des équations exactes.
Par exemple : $y' = \mathrm{e}^{x+y}$.

```
sage: desolve(diff(y,x)==exp(x+y),y,show_method=True)
[-(e^(x + y(x)) + 1)*e^(-y(x)) == c, 'exact']
```

Équations de Bernoulli Il s'agit d'équations du type :

$$y' + P(x)y = Q(y)y^{\alpha}$$

avec P et Q des fonctions continues sur des intervalles donnés et $\alpha \notin \{0; 1\}$. Par exemple : $y' - y = xy^4$.

```
sage: desolve(diff(y,x)-y==x*y^4,y,show_method=True)
[e^x/(-1/3*(3*x - 1)*e^(3*x) + c)^(1/3), 'bernoulli']
```

Équations homogènes Il s'agit d'équations du type :

$$y' = \frac{P(x,y)}{Q(x,y)}$$

avec P et Q des fonctions homogènes de même degré sur des intervalles donnés. Par exemple $x^2y' = y^2 + xy + x^2$.

```
sage: desolve(x^2*diff(y,x)==y^2+x*y+x^2,y,
              show_method=True)
[c*x == e^(arctan(y(x)/x)), 'homogeneous']
```

Les solutions ne sont pas données de manière explicite. Nous verrons plus loin comment se débrouiller dans certains cas.

Équations exactes Il s'agit d'équations du type :

$$\frac{\partial f}{\partial x}\mathrm{d}x + \frac{\partial f}{\partial y}\mathrm{d}y$$

avec f une fonction de deux variables différentiable.
Par exemple $y' = \frac{\cos(y)-2x}{y+x\sin(y)}$.

```
sage: desolve(diff(y,x)==(cos(y)-2*x)/(y+x*sin(y)),y,
              show_method=True)
[x^2 - x*cos(y(x)) + 1/2*y(x)^2 == c, 'exact']
```

Ici encore, les solutions ne sont pas données de manière explicite.

Équations de Ricatti Il s'agit d'équations du type :

$$y' = P(x)y^2 + Q(x)y + R(x)$$

avec P, Q et R des fonctions continues sur des intervalles donnés.
Par exemple : $y' = xy^2 + \frac{1}{x}y - \frac{1}{x^2}$.
Il faut dans ce cas fixer l'option `contrib_ode` à `True` pour que Sage cherche des solutions avec des méthodes plus complexes.

```
sage: desolve(diff(y,x)==x*y^2+y/x-1/x^2,y,contrib_ode=
              True,show_method=True)
[[y(x) == longue expression], 'riccati']
```

Équations de Lagrange et de Clairaut Lorsque l'équation est du type $y = xP(y') + Q(y')$ où P et Q sont de classe $\mathcal{C}^1$ sur un certain intervalle, on parle d'équation de Lagrange. Lorsque P est l'identité, on parle d'équation de Clairaut. Par exemple : $y = xy' - y'^2$.

```
sage: desolve(y==x*diff(y,x)-diff(y,x)^2,y,contrib_ode=
              True,show_method=True)
[[y(x) == -c^2 + c*x, y(x) == 1/4*x^2], 'clairault']
```

11.2.3 Équation linéaire

Résolvons par exemple $x' + 2x = t^2 - 2t + 3$:

```
sage: desolve(diff(x,t)+2*x==t**2-2*t+3,x)
-1/4*(2*(2*t - 1)*e^(2*t) - (2*t^2 - 2*t + 1)*e^(2*t) - 4*c
- 6*e^(2*t))*e^(-2*t)
```

Ordonnons un peu tout ça avec la commande `expand` :

```
sage: desolve(diff(x,t)+2*x==t**2-2*t+3,x).expand()
c*e^(-2*t) + 1/2*t^2 - 3/2*t + 9/4
```

C'est-à-dire $x(t) = c\ \mathrm{e}^{-2t} + \frac{t^2}{2} - \frac{3t}{2} + \frac{9}{4}$.
On prendra donc l'habitude d'utiliser `desolve(...).expand()`. Quelle est la méthode utilisée ?

```
sage:  desolve(diff(x,t)+2*x==t**2-2*t+3,x,show_method=True)[1]
'linear'
```

Ajoutons des conditions initiales, par exemple $x(0) = 1$:

```
sage: desolve(diff(x,t)+2*x==t**2-2*t+3,x,ics=[0,1]).expand()
1/2*t^2 - 3/2*t - 5/4*e^(-2*t) + 9/4
```

11.2.4 Équations à variables séparables

Étudions l'équation : $y' \ln(y) = y \sin(t)$

```
sage: t = var('t')
sage: y = function('y', t)
sage: desolve(diff(y,t)*ln(y)==y*sin(t),y,show_method=True)

[1/2*log(y(t))^2 == c - cos(t), 'separable']
```

Sage est d'accord avec nous : c'est bien une équation à variables séparables. Prenons l'habitude de nommer nos solutions pour pouvoir les réutiliser par la suite :

```
sage: ed=desolve(diff(y,t)*ln(y)==y*sin(t),y)
sage: ed
1/2*log(y(t))^2 == c - cos(t)
```

Ici, y n'est pas donné de façon explicite : $\frac{1}{2}\ln^2(y(t)) = c - \cos(t)$.
On peut demander une expression de $y(t)$ en utilisant `solve` :

```
solve(ed,y(t))
```

mais Sage n'est pas content :

```
TypeError: Computation failed since Maxima requested additional
constraints(try the command 'assume(c-cos(t)>0)'before integral
or limit evaluation, for example)
```

En effet, $c - \cos(t)$ doit être positif. Obéissons à Sage :

```
sage: assume(c-cos(t)>0)
NameError: name 'c' is not defined
```

En effet, nous n'avons pas défini c : c'est Sage qui l'a introduit. Pour accéder à c, nous allons utiliser `variables()` qui donne la liste des variables d'une expression :

```
sage: ed.variables()
(c, t)
```

Seuls c et t sont considérés comme des variables car y a été défini comme une fonction de la variable t. On accède donc à c avec `ed.variables()[0]` :

```
sage: c=ed.variables()[0]
sage: assume(c-cos(t)>0)
sage: solve(ed,y(t))
   [y(t) == e^(-sqrt(c - cos(t))*sqrt(2)),
    y(t) == e^(sqrt(c - cos(t))*sqrt(2))]
```

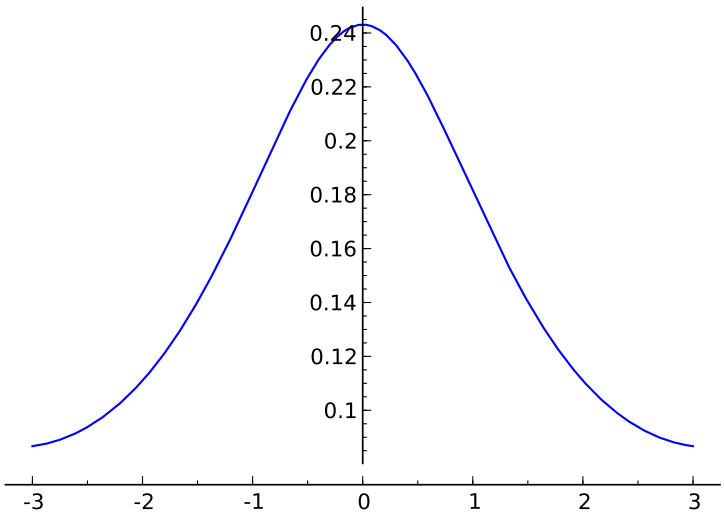
Pour avoir l'allure des courbes des solutions, il nous faut récupérer le membre de droite de chaque solution avec la commande `rhs()`. Par exemple, pour obtenir le membre de droite de la première solution en remplaçant c par 5 :

```
sage: solve(ed,y(t))[0].subs_expr(c==5).rhs()
   e^(-sqrt(-cos(t) + 5)*sqrt(2))
```

Autre exemple, pour avoir le tracé de la première solution avec $c = 2$:

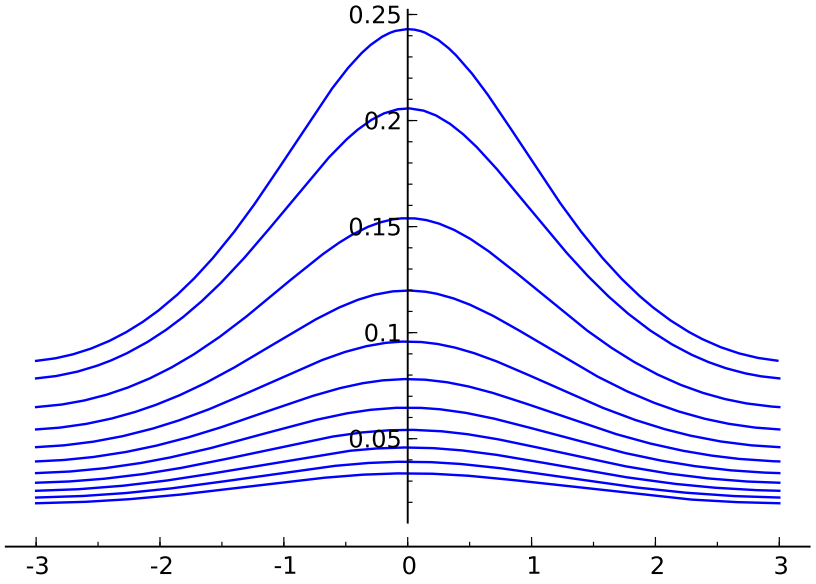
```
sage: plot(solve(ed,y(t))[0].subs_expr(c==2).rhs(),
           t,-3,3)
```

et on obtient :



Pour avoir plusieurs figures, on effectue une boucle :

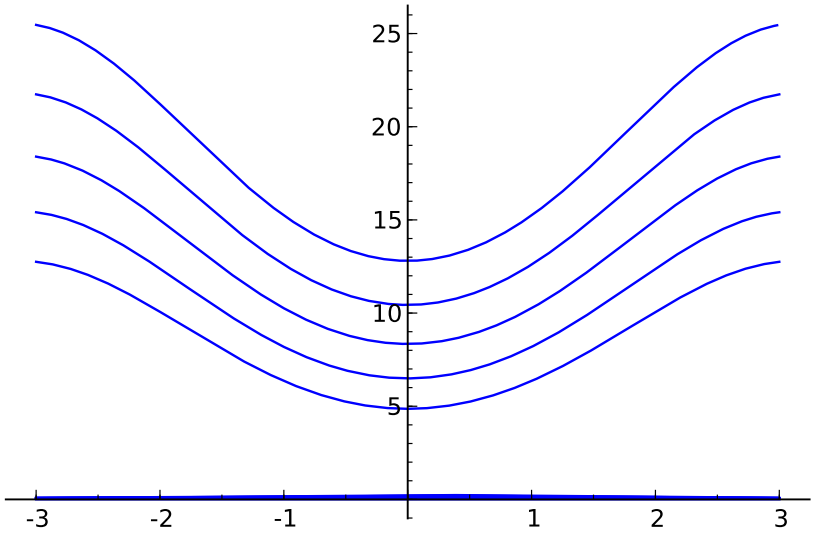
```
sage: P=Graphics()
sage: for k in range(1,20,2):
....: P+=plot(solve(ed,y(t))[0].subs_expr(c==1+0.25*k).rhs(),
              t,-3,3)
sage: P
```



On aurait pu avoir l'aspect correspondant aux deux solutions en effectuant une double boucle :

```
sage: P=Graphics()
sage: for j in [0,1]:
....: for k in range(1,10,2):
....: P+=plot(solve(ed,y(x))[j].subs_expr(c==2+0.25*k).rhs(),
              x,-3,3)
sage: P
```

mais la différence d'échelle entre les solutions ne permet plus de distinguer les courbes correspondant à la première solution :



Exercice 36 (Équations différentielles à variables séparables)**.** Étudiez les équations à variables séparables suivantes :

1. (E_1) : $\frac{yy'}{\sqrt{1+y^2}} = \sin(x)$;
2. (E_2) : $y' = \frac{\sin(x)}{\cos(y)}$.

11.2.5 Équations homogènes

On veut résoudre l'équation homogène $tx' = x + \sqrt{x^2+t^2}$. C'est bien une équation homogène car on peut l'écrire :

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{x+\sqrt{x^2+t^2}}{t} = \frac{N(x,t)}{M(x,t)}.$$

Or $N(kx,kt) = kN(x,t)$ et $M(kx,kt) = kM(x,t)$.
Il suffit donc d'effectuer le changement de variable vérifiant $x(t) = t \cdot u(t)$ pour tout réel t pour obtenir une équation à variables séparables.

```
sage: u=function('u',t)
sage: x(t) = function('x',t)
sage: x(t)=t*u(t)
sage: d=diff(x,t)
sage: DE=(t*d==x+sqrt(t**2+x**2))
```

On effectue le changement de variables dans l'équation différentielle de départ. L'équation n'étant pas définie en 0, on va la résoudre sur $]0;+\infty[$ et sur $]-\infty;0[$. Travaillons d'abord sur $]0;+\infty[$:

```
sage: assume(t>0)
sage: desolve(DE,u)
  arcsinh(u(t)) == c + integrate(abs(t)/t^2, t)
```

On n'obtient pas u de manière explicite. Pour y remédier, on va chercher une commande dans Maxima : `ev` comme *évaluer* avec l'option `logarc=true` qui indique que les fonctions trigonométriques hyperboliques inverses seront converties à l'aide de logarithmes. Ensuite on va pouvoir exprimer u à l'aide de la commande `solve` de Sage :

```
sage:S=desolve(DE,u)._maxima_().ev(logarc=true).sage().solve(u)
sage:S
   [u(t) == t*e^c - sqrt(u(t)^2 + 1)]
```

Ici, `S` est une liste constituée d'une équation ; `S[0]` sera donc l'équation elle-même.
L'équation n'est cependant toujours pas résolue explicitement. Nous allons aider un peu Sage en lui demandant de résoudre l'équation équivalente

$$(te^c - u)^2 = u^2 + 1$$

Nous allons donc soustraire `t*e^c` aux deux membres de l'équation et élever les deux membres résultant au carré.
Un problème subsiste car c n'est pas déclaré comme variable. Faisons-le en en profitant pour simplifier son écriture :

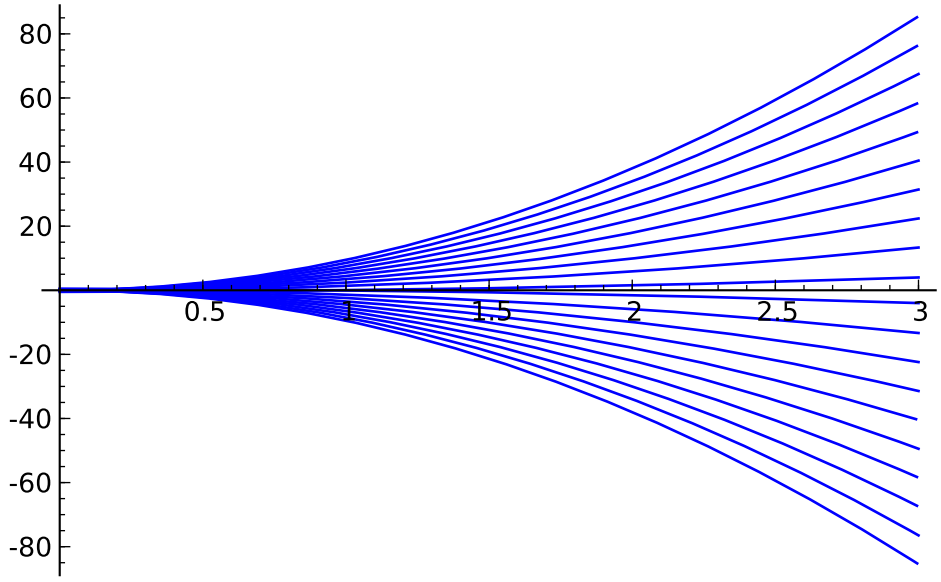
```
sage: C=var('C')
sage: solu=S[0].subs(c=log(C))
sage: solu
   u(t) == C*t - sqrt(u(t)^2 + 1)
sage: solu=((t*C-solu)^2).solve(u)
sage: solu
   [u(t) == 1/2*(C^2*t^2 - 1)/(C*t)]
```

Il ne reste plus qu'à revenir à x :

```
sage: x(t)=t*solu[0].rhs()
sage: x(t)
   1/2*(C^2*t^2 - 1)/C
```

Nous obtenons bien les solutions sous forme explicite : $x(t) = \dfrac{(Ct)^2 - 1}{2C}$.
Il ne reste plus qu'à tracer les solutions sur $]0; +\infty[$ en faisant attention à prendre des constantes C non nulles :

```
sage: P=Graphics()
sage: for k in range(-19,19,2):
....:  P+=plot(x(t).subs_expr(C==k),t,0,3)
sage: P
```



Exercice 37 (Équations différentielles homogènes)**.** Étudiez l'équation homogène suivante : (E_5) : $xyy' = x^2 + y^2$

11.2.6 Une équation à paramètres : le modèle de Verhulst

Le taux relatif de croissance d'une population est une fonction linéairement décroissante de la population.
Pour l'étudier, on peut être amené à résoudre une équation du type :

$$x' = ax - bx^2$$

avec a et b des paramètres réels positifs.

```
sage: var('a,b')
sage: DE=(diff(x,t)-a*x==-b*x**2)
sage: sol=desolve(DE,[x,t])
sage: sol
   -(log(b*x(t) - a) - log(x(t)))/a == c + t
```

Nous n'avons pas x explicitement. Essayons de l'isoler avec `solve` :

```
sage: Sol=solve(sol,x(t))[0]
sage: Sol
   log(x(t)) == (c + t)*a + log(b*x(t) - a)
```

Nous n'avons toujours pas de solution explicite. Nous allons regrouper les termes à gauche et simplifier cette expression à l'aide de `simplify_log()` :

```
sage: Sol=Sol.lhs()-Sol.rhs()
sage: Sol
   -(c + t)*a - log(b*x(t) - a) + log(x(t))
sage: Sol=Sol.simplify_log()
sage: solve(Sol,x(t))[0].simplify()
   x(t) == a*e^(a*c + a*t)/(b*e^(a*c + a*t) - 1)
```

11.3 Équations d'ordre 2

11.3.1 Équations linéaires à coefficients constants

Résolvons maintenant une équation du second ordre linéaire à coefficients constants, par exemple :

$$x'' + 3x = t^2 - 7t + 31.$$

On utilise la même syntaxe que pour les équations d'ordre 1, la dérivée seconde de x par rapport à t s'obtenant avec `diff(x,t,2)`.

```
sage: desolve(diff(x,t,2)+3*x==t^2-7*t+31,x).expand()
   k1*sin(sqrt(3)*t) + k2*cos(sqrt(3)*t) + 1/3*t^2 - 7/3*t
   + 91/9
```

c'est-à-dire : $k_1 \sin\left(\sqrt{3}t\right) + k_2 \cos\left(\sqrt{3}t\right) + \frac{1}{3}\,t^2 - \frac{7}{3}\,t + \frac{91}{9}$. Ajoutons des conditions initiales, par exemple $x(0) = 1$ et $x'(0) = 2$:

```
sage:desolve(diff(x,t,2)+3*x==t^2-7*t+31,x,ics=[0,1,2])
              .expand()
   1/3*t^2+13/9*sqrt(3)*sin(sqrt(3)*t)-7/3*t
   -82/9*cos(sqrt(3)*t)+91/9
```

ou bien $x(0) = 1$ et $x(-1) = 0$:

```
sage:desolve(diff(x,t,2)+3*x==t^2-7*t+31,x,ics=[0,1,-1,0])
              .expand()
   1/3*t^2-7/3*t-82/9*sin(sqrt(3)*t)*cos(sqrt(3))/sin(sqrt(3))
   +115/9*sin(sqrt(3)*t)/sin(sqrt(3))-82/9*cos(sqrt(3)*t)+91/9
```

C'est-à-dire $\frac{1}{3}\,t^2 - \frac{7}{3}\,t + \frac{-82\,\sin\left(\sqrt{3}t\right)\cos\left(\sqrt{3}\right)}{9\,\sin\left(\sqrt{3}\right)} + \frac{115\,\sin\left(\sqrt{3}t\right)}{9\,\sin\left(\sqrt{3}\right)} - \frac{82}{9}\,\cos\left(\sqrt{3}t\right) + \frac{91}{9}$

11.3.2 Sage mis en défaut ?

Étudions la célèbre équation de la chaleur. La température z se répartit dans une tige rectiligne homogène de longueur ℓ selon l'équation :

$$\frac{\partial^2 z}{\partial x^2}(x,t) = C\frac{\partial z}{\partial t}(x,t).$$

On étudiera cette équation pour :

$$\forall t \in \mathbb{R}^+, \quad z(0,t) = 0 \quad z(\ell,t) = 0 \quad \forall x \in]0;\ell[, \quad z(x,0) = 1$$

Attention ! On sort ici du domaine des équations différentielles ordinaires (EDO) pour étudier une équation aux dérivées partielles (EDP). On va chercher des solutions ne s'annulant pas sous la forme :

$$z(x,t) = f(x)g(t)$$

C'est la méthode de séparation des variables.

```
sage: var('x,t')
sage: z=function('z',x,t)
sage: f=function('f',x)
sage: g=function('g',t)
sage: z=f*g
sage: eq=(diff(z,x,2)==diff(z,t))
sage: eq
   g(t)*D[0, 0](f)(x) == f(x)*D[0](g)(t)
```

L'équation devient donc :

$$g(t)\cdot\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = f(x)\cdot\frac{\mathrm{d}g(t)}{\mathrm{d}t}.$$

Divisons par $f(x)g(t)$ supposé non nul :

```
sage: eqn=eq/(f(x)*g(t))
sage: eqn
   D[0, 0](f)(x)/f(x) == D[0](g)(t)/g(t)
```

On obtient alors une équation où chaque membre ne dépend que d'une variable :

$$\frac{1}{f(x)}\cdot\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = \frac{1}{g(t)}\cdot\frac{\mathrm{d}g(t)}{\mathrm{d}t}$$

Chaque membre ne peut donc qu'être constant. Séparons les équations et introduisons une constante k :

```
sage: k=var('k')
sage: eq1=eqn.lhs()==k
sage: eq2=eqn.rhs()==k
```

Résolvons séparément les équations en commençant par la deuxième :

```
sage: g(t)=desolve(eq2,[g,t])
   c*e^(k*t)
```

donc $g(t) = ce^{kt}$ avec c une constante réelle.
Pour la première, nous n'y arrivons pas directement :

```
sage: desolve(eq1,[f,x])
TypeError: Computation failed since Maxima requested additional
constraints(try the command 'assume(k>0)' before integral or
limit evaluation,for example):Is k positive, negative, or zero?
```

Utilisons `assume` :

```
sage: assume(k>0)
sage: desolve(eq1,[f,x])
```

Le message est le même... Insistons alors en prenant un paramètre sous la forme $-k^2 - 1$:

```
sage: desolve(diff(y(x),x,2)==(-k**2-1)*y,[y,x])
   k1*sin(sqrt(k^2 + 1)*x) + k2*cos(sqrt(k^2 + 1)*x)
```

C'est mieux ! Et avec $k^2 + 1$:

```
sage: desolve(diff(y(x),x,2)==(k**2+1)*y,[y,x])
   k1*e^(I*sqrt(-k^2 - 1)*x) + k2*e^(-I*sqrt(-k^2 - 1)*x)
```

Ce n'est pas faux mais ce n'est pas encore optimisé. Sage peut donc parfois apparaître comme un peu jeune...

11.4 Transformée de Laplace

11.4.1 Rappel

La transformée de Laplace permet de convertir une équation différentielle avec des conditions initiales en équation algébrique et la transformée inverse permet ensuite de revenir à la solution éventuelle de l'équation différentielle. Pour mémoire, si f est une fonction définie sur $\mathbb{R}$ en étant identiquement nulle sur $]-\infty; 0[$, on appelle transformée de Laplace de f la fonction F définie, sous certaines conditions, par :

$$\mathcal{L}(f(x)) = F(s) = \int_0^{+\infty} \mathrm{e}^{-sx} f(x)\, \mathrm{d}x$$

On obtient facilement les transformées de Laplace des fonctions polynomiales, trigonométriques, exponentielles, etc. qu'on regroupe dans une table. Ces transformées ont des propriétés fort intéressantes, notamment concernant la transformée d'une dérivée : si f' est continue par morceaux sur $\mathbb{R}_+$ alors

$$\mathcal{L}(f'(x)) = s\mathcal{L}(f(x)) - f(0)$$

et si f' satisfait les conditions imposées sur f :

$$\mathcal{L}(f''(x)) = s^2\mathcal{L}(f(x)) - sf(0) - f'(0)$$

11.4.2 Exemple

On veut résoudre l'équation différentielle $y'' - 3y' - 4y - \sin(x)$ en utilisant les transformées de Laplace avec les conditions initiales $y(0) = 1$ et $y'(0) = -1$. Alors :

$$\mathcal{L}\left(y'' - 3y' - 4y\right) = \mathcal{L}\left(\sin(x)\right)$$

c'est-à-dire :

$$\left(s^2 - 3s - 4\right)\mathcal{L}(y) - sy(0) - y'(0) + 3y(0) = \mathcal{L}(\sin(x))$$

Si on a oublié les tables des transformées de Laplace des fonctions usuelles, on peut utiliser Sage pour retrouver la transformée du sinus :

```
sage: x=var('x')
sage: s=var('s')
sage: f=function('f',x)
sage: f(x)=sin(x)
sage: f.laplace(x,s)
   t |--> 1/(s^2 + 1)
```

Ainsi on obtient une expression de la transformée de Laplace de y :

$$\mathcal{L}(y) = \frac{1}{(s^2 - 3s - 4)(s^2 + 1)} + \frac{s - 4}{s^2 - 3s - 4}$$

Utilisons alors Sage pour obtenir la transformée inverse :

```
sage: X=function('X',s)
sage: X(s)=1/(s^2-3*s-4)/(s^2+1)+(s-4)/(s^2-3*s-4)
sage: X(s).inverse_laplace(s,x)
   9/10*e^(-x) + 1/85*e^(4*x) - 5/34*sin(x) + 3/34*cos(x)
```

Si on veut à moitié « tricher », on peut décomposer $X(s)$ en éléments simples d'abord :

```
sage: X(s).partial_fraction()
   1/34*(3*s - 5)/(s^2 + 1) + 1/85/(s - 4) + 9/10/(s + 1)
```

et il ne reste plus qu'à lire une table d'inverses. On peut cependant utiliser directement utiliser la boîte noire `desolve_laplace` qui donnera directement la solution :

```
sage: desolve_laplace(diff(y(x),x,2)-3*diff(y(x),x)-4*y(x)
                      ==sin(x),[y,x],ics=[0,1,-1])
   9/10*e^(-x) + 1/85*e^(4*x) - 5/34*sin(x) + 3/34*cos(x)
```

Quatrième partie

Probabilités, combinatoire et statistiques

12

Dénombrement et combinatoire

Ce chapitre aborde principalement le traitement avec Sage des problèmes combinatoires suivants : le dénombrement (combien y a-t-il d'éléments dans un ensemble S ?), l'énumération (calculer tous les éléments de S, ou itérer parmi eux), le tirage aléatoire (choisir au hasard un élément de S selon une loi, par exemple uniforme). Ces questions interviennent naturellement dans les calculs de probabilités (quelle est la probabilité au poker d'obtenir une suite ou un carré d'as ?), en physique statistique, mais aussi en calcul formel (nombre d'éléments dans un corps fini), ou en analyse d'algorithmes. La combinatoire couvre un domaine beaucoup plus vaste (graphes, ordres partiels, théorie des représentations, ...) pour lesquels nous nous contentons de donner quelques pointeurs vers les possibilités offertes par Sage.

Une caractéristique de la combinatoire effective est la profusion de types d'objets et d'ensembles que l'on veut manipuler. Il serait impossible de les décrire et a fortiori de les implanter tous. Ce chapitre illustre donc la méthodologie sous-jacente : fournir des briques de base pour décrire les ensembles combinatoires usuels §12.2, des outils pour les combiner et construire de nouveaux ensembles §12.3, et des algorithmes génériques pour traiter uniformément de grandes classes de problèmes §12.4.

C'est un domaine où Sage a des fonctionnalités bien plus étendues que la plupart des systèmes de calcul formel et est en pleine expansion ; en revanche il reste encore très jeune avec de multiples limitations arbitraires et incohérences.

12.1 Premiers exemples

12.1.1 Jeu de poker et probabilités

Nous commençons par résoudre un problème classique : dénombrer certaines combinaisons de cartes dans un jeu de poker, pour en déduire leur probabilité.

Une carte de poker est caractérisée par une couleur (cœur, carreau, pique ou trèfle) et une valeur (2, 3, ..., 9, valet, dame, roi, ou as). Le jeu de poker est constitué de toutes les cartes possibles ; il s'agit donc du produit cartésien de l'ensemble des couleurs et de l'ensemble des valeurs :

$$\text{Cartes} = \text{Symboles} \times \text{Valeurs} = \{(s, v) \mid s \in \text{Symboles} \text{ et } v \in \text{Valeurs}\}.$$

Construisons ces ensembles dans Sage :

```
sage: Symboles = Set(["Coeur", "Carreau", "Pique", "Trefle"])
sage: Valeurs = Set([2, 3, 4, 5, 6, 7, 8, 9, 10,
....:                "Valet", "Dame", "Roi", "As"])
sage: Cartes = CartesianProduct(Symboles, Valeurs)
```

Il y a 4 couleurs et 13 valeurs possibles donc $4 \times 13 = 52$ cartes dans le jeu de poker :

```
sage: Symboles.cardinality()
4
sage: Valeurs.cardinality()
13
sage: Cartes.cardinality()
52
```

Tirons une carte au hasard :

```
sage: Cartes.random_element()
['Trefle', 6]
```

Une petite digression technique est ici nécessaire. Les éléments du produit cartésien sont renvoyés sous forme de listes :

```
sage: type(Cartes.random_element())
<type 'list'>
```

Une liste Python n'étant pas immuable, on ne peut pas la mettre dans un ensemble (voir §3.2.7), ce qui nous poserait problème par la suite. Nous redéfinissons donc notre produit cartésien pour que les éléments soient représentés par des tuples :

```
sage: Cartes = CartesianProduct(Symboles, Valeurs).map(tuple)
sage: Cartes.random_element()
('Trefle', 'As')
```

On peut maintenant construire un ensemble de cartes :

```
sage: Set([Cartes.random_element(), Cartes.random_element()])
{('Coeur', 2), ('Pique', 4)}
```

Ce problème devrait disparaître à terme : il est prévu de changer l'implantation des produits cartésiens pour que leurs éléments soient immuables par défaut.

Revenons à notre propos. On considère ici une version simplifiée du jeu de poker, où chaque joueur pioche directement cinq cartes, qui forment une *main*. Toutes les cartes sont distinctes et l'ordre n'a pas d'importance ; une main est donc un sous-ensemble de taille 5 de l'ensemble des cartes. Pour tirer une main au hasard, on commence par construire l'ensemble de toutes les mains possibles puis on en demande un élément aléatoire :

```
sage: Mains = Subsets(Cartes, 5)
sage: Mains.random_element()
{('Coeur', 4), ('Carreau', 9), ('Pique', 8),
 ('Trefle', 9), ('Coeur', 7)}
```

Le nombre total de mains est donné par le nombre de sous-ensembles de taille 5 d'un ensemble de taille 52, c'est-à-dire le coefficient binomial $\binom{52}{5}$:

```
sage: binomial(52,5)
2598960
```

On peut aussi ne pas se préoccuper de la méthode de calcul, et simplement demander sa taille à l'ensemble des mains :

```
sage: Mains.cardinality()
2598960
```

La force d'une main de poker dépend de la combinaison de ses cartes. Une de ces combinaisons est la *couleur* ; il s'agit d'une main dont toutes les cartes ont le même symbole (en principe il faut exclure les quintes flush ; ce sera l'objet d'un exercice ci-dessous). Une telle main est donc caractérisée par le choix d'un symbole parmi les quatre possibles et le choix de cinq valeurs parmi les treize possibles. Construisons l'ensemble de toutes les couleurs, pour en calculer le nombre :

```
sage: Couleurs = CartesianProduct(Symboles, Subsets(Valeurs, 5))
sage: Couleurs.cardinality()
5148
```

La probabilité d'obtenir une couleur en tirant une main au hasard est donc de :

```
sage: Couleurs.cardinality()  / Mains.cardinality()
33/16660
```

soit d'environ deux sur mille :

```
sage: 1000.0 * Couleurs.cardinality()  / Mains.cardinality()
1.98079231692677
```

Faisons une petite simulation numérique. La fonction suivante teste si une main donnée est une couleur :

```
sage: def est_couleur(main):
....:     return len(set(symbole for (symbole, val) in main)) == 1
```

Nous tirons maintenant 10000 mains au hasard, et comptons le nombre de couleurs obtenues (cela prend environ 10 s) :

```
sage: n = 10000
sage: ncouleurs = 0
sage: for i in range(n):
....:     main = Mains.random_element()
....:     if est_couleur(main):
....:         ncouleurs += 1
sage: print n, ncouleurs
10000, 18
```

Exercice 38. Une main contenant quatre cartes de la même valeur est appelée un *carré*. Construire l'ensemble des carrés (indication : utiliser `Arrangements` pour tirer au hasard un couple de valeurs distinctes puis choisir un symbole pour la première valeur). Calculer le nombre de carrés, en donner la liste, puis déterminer la probabilité d'obtenir un carré en tirant une main au hasard.

Exercice 39. Une main dont les cartes ont toutes le même symbole et dont les valeurs se suivent est appelée une *quinte flush* et non une *couleur*. Compter le nombre de quintes flush, puis en déduire la probabilité correcte d'obtenir une couleur en tirant une main au hasard.

Exercice 40. Calculer la probabilité de chacune des combinaisons de cartes au poker (voir http://fr.wikipedia.org/wiki/Main_au_poker) et comparer avec le résultat de simulations.

12.1.2 Dénombrement d'arbres par séries génératrices

Dans cette section, nous traitons l'exemple des arbres binaires complets, et illustrons sur cet exemple plusieurs techniques de dénombrement où le calcul formel intervient naturellement. Ces techniques sont en fait générales, s'appliquant à chaque fois que les objets combinatoires considérés admettent une définition récursive (grammaire) (voir §12.4.3 pour un traitement automatisé). L'objectif n'est pas de présenter ces méthodes formellement ; aussi les calculs seront rigoureux mais la plupart des justifications seront passées sous silence.

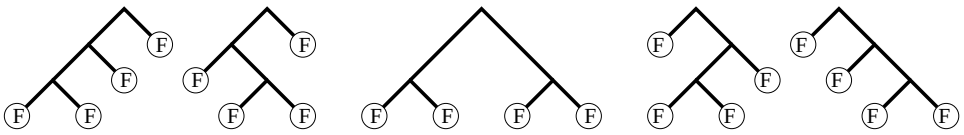


FIG. 12.1 – Les cinq arbres binaires complets à quatre feuilles.

Un *arbre binaire complet* est soit une feuille F, soit un nœud sur lequel on a greffé deux arbres binaires complets (voir figure 12.1).

Exercice 41. Chercher à la main tous les arbres binaires complets à $n = 1, 2, 3, 4, 5$ feuilles (voir l'exercice 49 pour les chercher avec Sage).

Notre objectif est de compter le nombre c_n d'arbres binaires complets à n feuilles (dans la section, et sauf mention explicite du contraire, tous les arbres sont binaires complets). C'est une situation typique où l'on ne s'intéresse pas seulement à un ensemble isolé, mais à une famille d'ensembles, typiquement paramétrée par $n \in \mathbb{N}$.

D'après l'exercice 41, les premiers termes sont donnés par $c_1, \ldots, c_5 = 1, 1, 2, 5, 14$. Le simple fait d'avoir ces quelques nombres est déjà précieux. En effet, ils permettent une recherche dans une mine d'or : l'*encyclopédie en ligne des suites de nombres entiers* http://www.research.att.com/~njas/sequences/ appelée communément le Sloane, du nom de son auteur principal, et qui contient plus de 170000 suites d'entiers :

```
sage: sloane_find([1,1,2,5,14])
Searching Sloane's online database...
[[108, 'Catalan numbers: C(n) = binomial(2n,n)/(n+1) ...
```

Le résultat suggère que les arbres sont comptés par l'une des plus fameuses suites, les nombres de Catalan. En fouillant dans les références fournies par le Sloane, on trouverait que c'est effectivement le cas : les quelques nombres ci-dessus forment une empreinte digitale de nos objets, qui permettent de retrouver en quelques secondes un résultat précis dans une abondante littérature.

L'objectif de la suite est de retrouver ce résultat avec l'aide de Sage. Soit C_n l'ensemble des arbres à n feuilles, de sorte que $c_n = |C_n|$; par convention, on définira $C_0 = \emptyset$, soit $c_0 = 0$. L'ensemble de tous les arbres est alors la réunion disjointe des C_n :

$$C = \biguplus_{n \in \mathbb{N}} C_n \,.$$

Du fait d'avoir nommé l'ensemble C de tous les arbres, on peut traduire la définition récursive des arbres en une équation ensembliste :

$$C \quad \approx \quad \{\mathrm{F}\} \quad \uplus \quad C \times C \,.$$

En mots : un arbre t (donc dans C) est soit une feuille (donc dans $\{\mathrm{F}\}$) soit un nœud sur lequel on a greffé deux arbres t_1 et t_2 et que l'on peut donc identifier avec le couple (t_1, t_2) (donc dans le produit cartésien $C \times C$).

L'idée fondatrice de la combinatoire algébrique, introduite par Euler dans une lettre à Goldbach en 1751 pour traiter un problème similaire [Vie07], est de manipuler simultanément tous les coefficients c_n en les encodant sous la forme d'une série formelle, dite *série génératrice* des c_n :

$$C(z) = \sum_{n \in \mathbb{N}} c_n z^n$$

où z est une indéterminée formelle (on n'a donc pas besoin de se préoccuper de questions de convergence). La beauté de cette idée est que les opérations ensemblistes ($A \uplus B$, $A \times B$) se traduisent naturellement en opérations algébriques sur les séries ($A(z) + B(z)$, $A(z) \cdot B(z)$), de sorte que l'équation ensembliste vérifiée par C se traduit en une équation algébrique sur $C(z)$:

$$C(z) = z + C(z) \cdot C(z)$$

Résolvons cette équation avec Sage. Pour cela, on introduit deux variables C et z, et on pose le système :

```
sage: var("C,z");
sage: sys = [ C == z + C*C ]
```

On a alors deux solutions, qui par chance sont sous forme close :

```
sage: sol = solve(sys, C, solution_dict=True); sol
[{C: -1/2*sqrt(-4*z + 1) + 1/2}, {C: 1/2*sqrt(-4*z + 1) + 1/2}]
sage: s0 = sol[0][C]; s1 = sol[1][C]
```

et dont les développements de Taylor commencent par :

```
sage: taylor(s0, z, 0, 5)
14*z^5 + 5*z^4 + 2*z^3 + z^2 + z
sage: taylor(s1, z, 0, 5)
-14*z^5 - 5*z^4 - 2*z^3 - z^2 - z + 1
```

La deuxième solution est clairement aberrante ; par contre, on retrouve les coefficients prévus sur la première. Posons donc :

```
sage: C = s0
```

On peut maintenant calculer les termes suivants :

```
sage: taylor(C, z, 0, 10)
4862*z^10 + 1430*z^9 + 429*z^8 + 132*z^7 + 42*z^6 +
14*z^5 + 5*z^4 + 2*z^3 + z^2 + z
```

ou calculer quasi instantanément le 100-ème coefficient :

```
sage: taylor(C, z, 0, 100).coeff(z,100)
22750883079422934966181954039568885395604168260154104734O
```

Il est cependant dommage de devoir tout recalculer si jamais on voulait le 101-ième coefficient. Les séries formelles paresseuses (voir §9.1.8) prennent alors tout leur sens, d'autant que l'on peut les définir directement à partir du système d'équations, sans le résoudre, et donc en particulier sans avoir besoin de forme close pour le résultat. On commence par définir l'anneau des séries formelles paresseuses :

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
```

Puis l'on crée une série formelle « libre », à laquelle on donne un nom, et que l'on définit ensuite par une équation récursive :

```
sage: C = L()
sage: C._name = 'C'
sage: C.define( z + C * C );

sage: [C.coefficient(i) for i in range(11)]
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

On peut à tout moment demander un coefficient quelconque sans avoir à redéfinir C :

```
sage: C.coefficient(100)
22750883079422934966181954039568885395604168260154104734O
sage: C.coefficient(200)
1290131580644291140012229076696766751343495305527288824998\
10851598901419013348319045534580850847735528275750122188940
```

Nous revenons maintenant à la forme close pour $C(z)$:

```
sage: var('z');
sage: C = s0; C
```

Le n-ième coefficient du développement de Taylor de $C(z)$ étant donné par $\frac{1}{n!}C(z)^{(n)}(0)$, regardons les dérivées successives $C(z)^{(n)}$ de $C(z)$:

```
sage: derivative(C, z, 1)
1/sqrt(-4*z + 1)
sage: derivative(C, z, 2)
2/(-4*z + 1)^(3/2)
sage: derivative(C, z, 3)
12/(-4*z + 1)^(5/2)
```

Cela suggère l'existence d'une formule explicite simple que l'on recherche maintenant. La petite fonction suivante renvoie $d_n = n!\,c_n$:

```
sage: def d(n): return derivative(s0, n).subs(z=0)
```

En en prenant les quotients successifs :

```
sage: [ (d(n+1) / d(n)) for n in range(1,18) ]
[2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62, 66]
```

on constate que d_n satisfait la relation de récurrence $d_{n+1} = (4n-2)d_n$, d'où l'on déduit que c_n satisfait la relation de récurrence $c_{n+1} = \frac{(4n-2)}{n+1}c_n$. En simplifiant, on obtient alors que c_n est le $(n-1)$-ième nombre de Catalan :

$$c_n = \text{Catalan}(n-1) = \frac{1}{n}\binom{2(n-1)}{n-1}.$$

Vérifions cela :

```
sage: var('n');
sage: c = 1/n*binomial(2*(n-1),n-1)
sage: [c.subs(n=k) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
sage: [catalan_number(k-1) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

On peut maintenant calculer les coefficients beaucoup plus loin ; ici on calcule c_{100000} qui a plus de 60000 chiffres :

```
sage: %time x = c(100000)
CPU times: user 2.34 s, sys: 0.00 s, total: 2.34 s
Wall time: 2.34 s
sage: ZZ(x).ndigits()
60198
```

Les méthodes que nous avons utilisées se généralisent à tous les objets définis récursivement : le système d'équations ensembliste se traduit en un système d'équations sur la série génératrice ; celui-ci permet de calculer récursivement ses coefficients. Lorsque les équations ensemblistes sont suffisamment simples (par exemple ne font intervenir que des produits cartésiens et unions disjointes), l'équation sur $C(z)$ est algébrique. Elle admet rarement une solution en forme close, mais en calculant les dérivées successives, on peut en déduire en général une équation différentielle *linéaire* sur $C(z)$, qui se traduit en une équation de récurrence de longueur fixe sur les coefficients c_n (la série est alors dite *holonomique*). Au final, après le précalcul de cette équation de récurrence, le calcul des coefficients devient très rapide.

12.2 Ensembles énumérés usuels

12.2.1 Premier exemple : les sous-ensembles d'un ensemble

Fixons un ensemble E de taille n et considérons les sous-ensembles de E de taille k. On sait que ces sous-ensembles sont comptés par les coefficients binomiaux $\binom{n}{k}$. On peut donc calculer le nombre de sous-ensembles de taille $k = 2$ de $E = \{1, 2, 3, 4\}$ avec la fonction `binomial` :

```
sage: binomial(4, 2)
6
```

Alternativement, on peut *construire* l'ensemble $\mathcal{P}_2(E)$ de tous les sous-ensembles de taille 2 de E, puis lui demander sa cardinalité :

```
sage: S = Subsets([1,2,3,4], 2)
sage: S.cardinality()
6
```

Une fois `S` construit, on peut aussi obtenir la liste de ses éléments :

```
sage: S.list()
[{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
```

ou tirer un élément au hasard :

```
sage: S.random_element() #doctest: random
{1, 4}
```

Plus précisément, l'objet `S` modélise l'ensemble $\mathcal{P}_2(E)$, muni d'une énumération fixée (donnée ici par l'ordre lexicographique). On peut donc demander son 5-ième élément, en prenant garde au fait que, comme pour les listes Python, le premier élément est de rang 0 :

```
sage: S.unrank(4)
{2, 4}
```

À titre de raccourci, on peut utiliser ici la notation :

```
sage: S[4]
{2, 4}
```

mais cela est à utiliser avec prudence car certains ensembles sont munis d'une indexation naturelle autre que par $(0, \ldots,)$.

Réciproquement, on peut calculer le rang d'un objet dans cette énumération :

```
sage: s = S([2,4]); s
{2, 4}
sage: S.rank(s)
4
```

À noter que `S` n'*est pas* la liste de ses éléments. On peut par exemple modéliser l'ensemble $\mathcal{P}(\mathcal{P}(\mathcal{P}(E)))$ et calculer sa cardinalité ($2^{2^{2^4}}$) :

```
sage: E = Set([1,2,3,4])
sage: S = Subsets(Subsets(Subsets(E)))
sage: S.cardinality()
200352993040684646497907235156025575044782547556975141926501697 3
...736L
```

soit environ $2 \cdot 10^{19728}$:

```
sage: ZZ(S.cardinality()).ndigits()
19729
```

ou demander son 237102124-ième élément :

```
sage: S.unrank(237102123)
{{{2, 4}, {1, 2}, {1, 4}, {}, {2, 3, 4}, {3, 4}, {1, 3, 4}, {1}, {4}},
 {{2, 4}, {3, 4}, {1, 2, 3, 4}, {1, 2, 3}, {}, {2, 3, 4}}}
```

Il serait physiquement impossible de construire explicitement tous les éléments de S car il y en a bien plus que de particules dans l'univers (estimées à 10^{82}).

Remarque : il serait naturel avec Python d'utiliser `len(S)` pour demander la cardinalité de `S`. Cela n'est pas possible car Python impose que le résultat de `len` soit un entier de type `int` ; cela pourrait causer des débordements et ne permettrait pas de renvoyer `Infinity` pour les ensembles infinis.

```
sage: len(S)
...
AttributeError: __len__ has been removed; use .cardinality() instead
```

12.2.2 Partitions d'entiers

Nous considérons maintenant un autre problème classique : étant donné un entier positif n, de combien de façons peut-on l'écrire sous la forme d'une somme $n = i_1 + i_2 + \cdots + i_\ell$, où $i_1, \ldots, i_\ell$ sont des entiers strictement positifs ? Il y a deux cas à distinguer :

- l'ordre des éléments dans la somme n'a pas d'importance, auquel cas $(i_1, \ldots, i_\ell)$ est une *partition* de n ;
- l'ordre des éléments dans la somme revêt une importance, auquel cas $(i_1, \ldots, i_\ell)$ est une *composition* de n.

Regardons pour commencer les partitions de $n = 5$; comme précédemment, on commence par construire l'ensemble de ces partitions :

```
sage: P5 = Partitions(5); P5
Partitions of the integer 5
```

puis on demande sa cardinalité :

```
sage: P5.cardinality()
7
```

Regardons ces 7 partitions ; l'ordre n'ayant pas d'importance, les entrées sont triées, par convention, par ordre décroissant :

```
sage: P5.list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],
 [1, 1, 1, 1, 1]]
```

Le calcul du nombre de partitions utilise la formule de Rademacher, implantée en `C` et fortement optimisée, ce qui lui confère une grande rapidité :

```
sage: Partitions(100000).cardinality()
2749351056977569651267751632098635268817342931598005475820312598\
4302147328114964173055050741660736621590157844774296248940493063\
0702004617927644930335101160793424571901557189435097253124661084\
5200636955893446424871682878983218234500926285383140459702130713\
0674510624419227311238999702284408609370935531629697851569569892\
196108480158600569421098519
```

Les partitions d'entiers sont des objets combinatoires naturellement munis de multiples opérations. Elles sont donc renvoyées sous la forme d'objets plus riches que de simples listes :

```
sage: P7 = Partitions(7)
sage: p = P7.unrank(5); p
[4, 2, 1]
sage: type(p)
<class 'sage.combinat.partition.Partition_class'>
```

On peut par exemple les représenter graphiquement par un diagramme de Ferrer :

```
sage: print p.ferrers_diagram()
****
**
*
```

Nous laissons l'utilisateur explorer par introspection les opérations offertes.

À noter que l'on peut aussi construire une partition directement avec :

```
sage: Partition([4,2,1])
[4, 2, 1]
```

ou bien :

```
sage: P7([4,2,1])
[4, 2, 1]
```

Si l'on souhaite restreindre les valeurs possibles pour les parts $i_1, \ldots, i_\ell$ de la partition, comme par exemple dans les problèmes de rendu de monnaie, on peut utiliser `WeightedIntegerVectors`. Par exemple, le calcul suivant :

```
sage: WeightedIntegerVectors(8, [2,3,5]).list()
[[0, 1, 1], [1, 2, 0], [4, 0, 0]]
```

indique que pour former 8 dollars à partir de billets de 2$, 3$ et 5$ dollars, on peut utiliser un billet de 3$ et un de 5$ ou un billet de 2$ et deux de 3$ ou 4 billets de 2$.

Les compositions d'entiers se manipulent de la même façon :

```
sage: C5 = Compositions(5); C5
Compositions of 5
sage: C5.cardinality()
16
sage: C5.list()
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 3],
 [1, 2, 1, 1], [1, 2, 2], [1, 3, 1], [1, 4], [2, 1, 1, 1],
 [2, 1, 2], [2, 2, 1], [2, 3], [3, 1, 1], [3, 2], [4, 1], [5]]
```

Le 16 ci-dessus ne paraît pas anodin et suggère l'existence d'une éventuelle formule. Regardons donc le nombre de compositions de n pour n variant de 0 à 9 :

```
sage: [ Compositions(n).cardinality() for n in range(10) ]
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256]
```

De même, si l'on compte le nombre de compositions de 5 par longueur, on retrouve une ligne du triangle de Pascal :

```
sage: sum( x^len(c) for c in C5 )
x^5 + 4*x^4 + 6*x^3 + 4*x^2 + x
```

L'exemple ci-dessus utilise une fonctionnalité que l'on n'avait pas encore croisée : `C5` étant itérable, on peut l'utiliser comme une liste dans une boucle `for` ou une compréhension (§12.2.4).

Exercice 42. Démontrer la formule suggérée par les exemples ci-dessus pour le nombre de compositions de n et le nombre de compositions de n de longueur k et chercher par introspection si Sage utilise ces formules pour le calcul de cardinalité.

12.2.3 Quelques autres ensembles finis énumérés

Au final, le principe est le même pour tous les ensembles finis sur lesquels on veut faire de la combinatoire avec Sage ; on commence par construire un objet qui modélise cet ensemble puis on utilise les méthodes idoines qui suivent une interface uniforme[1]. Nous donnons maintenant quelques autres exemples typiques.

Les intervalles d'entiers :

```
sage: C = IntegerRange(3, 13, 2); C
{3, 5 .. 11}
sage: C.cardinality()
5
sage: C.list()
[3, 5, 7, 9, 11]
```

Les permutations :

[1] ou en tout cas cela devrait être le cas ; il reste de nombreux coins à nettoyer.

```
sage: C = Permutations(4); C
Standard permutations of 4
sage: C.cardinality()
24
sage: C.list()
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
 [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3],
 [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1],
 [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
 [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2],
 [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]
```

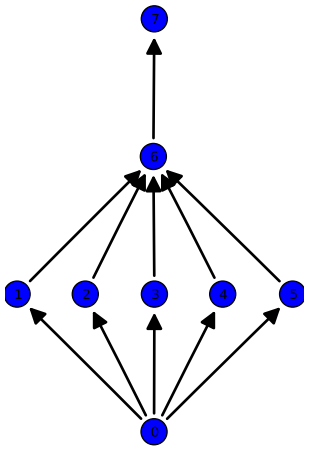
Les partitions ensemblistes :

```
sage: C = SetPartitions([1,2,3])
sage: C
Set partitions of [1, 2, 3]
sage: C.cardinality()
5
sage: C.list()
[{{1, 2, 3}}, {{2, 3}, {1}}, {{1, 3}, {2}}, {{1, 2}, {3}},
 {{2}, {3}, {1}}]
```

Les ordres partiels sur 5 sommets, à un isomorphisme près :

```
sage: C = Posets(8); C
Posets containing 8 vertices
sage: C.cardinality()
16999

sage: C.unrank(20)).plot()
```

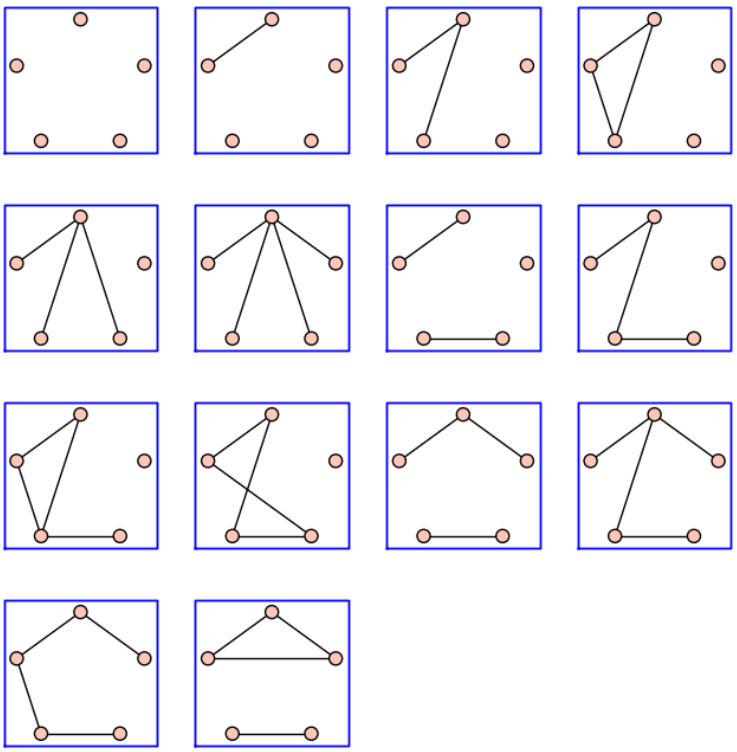


À noter que l'on peut construire la liste des 34 graphes simples à un isomorphisme près mais pas encore l'ensemble `C` de ces graphes (par exemple pour calculer `C.cardinality()`) :

```
sage: len(list(graphs(5)))
34
```

Voici tous les graphes à 5 sommets et moins de 4 arêtes :

```
sage: show(graphs(5, lambda G: G.size() <= 4))
```



Ce que l'on a vu s'applique aussi, en principe, aux structures algébriques finies comme le groupe dihédral :

```
sage: G = DihedralGroup(4); G
Dihedral group of order 8 as a permutation group
sage: G.cardinality()
8
sage: G.list()
[(), (2,4), (1,2)(3,4), (1,2,3,4), (1,3), (1,3)(2,4),
 (1,4,3,2), (1,4)(2,3)]
```

ou l'algèbre des matrices 2×2 sur le corps fini $\mathbb{Z}/2\mathbb{Z}$:

```
sage: C = MatrixSpace(GF(2), 2)
sage: C.list()
[
[0 0]  [1 0]  [0 1]  [0 0]  [0 0]  [1 1]  [1 0]  [1 0]
[0 0], [0 0], [0 0], [1 0], [0 1], [0 0], [1 0], [0 1],
```

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

Cependant ceci devrait renvoyer 16, mais n'est pas encore implanté :

```
sage: C.cardinality()
Traceback (most recent call last):
...
AttributeError:
'MatrixSpace_generic' object has no attribute 'cardinality'
```

Exercice 43. Lister tous les monômes de degré 5 dans les polynômes en trois variables (voir `IntegerVectors`). Manipuler les partitions ensemblistes ordonnées (`OrderedSetPartitions`) et les tableaux standard (`StandardTableaux`).

Exercice 44. Lister les matrices à signe alternant de taille 3, 4 et 5 (`AlternatingSignMatrices`), et essayer de deviner leur définition. La découverte et la démonstration de la formule de dénombrement de ces matrices (voir la méthode `cardinality`), motivée par des calculs de déterminants en physique, a été l'objet de toute une épopée. En particulier la première démonstration, donnée par Zeilberger en 1992 a été produite automatiquement par un programme, occupe 84 pages, et a nécessité l'intervention de presque cent vérificateurs [Zei96].

Exercice 45. Calculer à la main le nombre de vecteurs dans $(\mathbb{Z}/2\mathbb{Z})^5$ puis le nombre de matrices dans $GL_3(\mathbb{Z}/2\mathbb{Z})$ (c'est-à-dire le nombre de matrices 3×3 à coefficients dans $\mathbb{Z}/2\mathbb{Z}$ et inversibles). Vérifier votre réponse avec Sage. Généraliser à $GL_n(\mathbb{Z}/q\mathbb{Z})$.

12.2.4 Compréhensions et itérateurs

Nous allons maintenant montrer quelques possibilités offertes par Python pour construire (et itérer sur) des ensembles avec une notation flexible et proche des mathématiques, et le profit que l'on peut en tirer en combinatoire.

Commençons par construire l'ensemble fini $\{i^2 \mid i \in \{1, 3, 7\}\}$:

```
sage: [ i^2 for i in [1, 3, 7] ]
[1, 9, 49]
```

puis le même ensemble avec i variant cette fois entre 1 et 9 :

```
sage: [ i^2 for i in range(1,10) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On appelle *compréhension* une telle construction Python. On peut rajouter un prédicat pour ne garder que les éléments avec i premier :

```
sage: [ i^2 for i in range(1,10) if is_prime(i) ]
[4, 9, 25, 49]
```

En combinant plusieurs compréhensions, on peut construire l'ensemble $\{(i,j) \mid 1 \leqslant j < i < 5\}$:

```
sage: [ (i,j) for i in range(1,6) for j in range(1,i) ]
[(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3),
 (5, 1), (5, 2), (5, 3), (5, 4)]
```

ou bien afficher le triangle de Pascal :

```
sage: [ [ binomial(n, i) for i in range(n+1) ] for n in range(10) ]
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

L'exécution d'une compréhension se fait en deux étapes ; tout d'abord un *itérateur* est construit, puis une liste est remplie avec les éléments renvoyés successivement par l'itérateur. Techniquement, un *itérateur* est un objet avec une méthode `next` qui renvoie à chaque appel une nouvelle valeur, jusqu'à épuisement. Par exemple, l'itérateur `it` suivant :

```
sage: it = (binomial(3, i) for i in range(4))
```

renvoie successivement les coefficients binomiaux $\binom{3}{i}$ avec $i = 0, 1, 2, 3$:

```
sage: it.next()
1
sage: it.next()
3
sage: it.next()
3
sage: it.next()
1
```

Lorsque l'itérateur est finalement épuisé, une exception est levée :

```
sage: it.next()
Traceback (most recent call last):
  ...
StopIteration
```

Dans la pratique on n'utilise que très rarement directement cette méthode *next*. Un *itérable* est un object Python `L` (une liste, un ensemble, ...) sur les éléments duquel on peut itérer. Techniquement, on construit l'itérateur avec `iter(L)`, mais là encore on ne le fait que rarement explicitement.

Quel est l'intérêt d'un itérateur ? Considérons l'exemple suivant :

```
sage: sum( [ binomial(8, i) for i in range(9) ] )
256
```

À l'exécution, une liste avec 9 éléments est construite, puis elle est passée en argument à `sum` pour les ajouter. Si au contraire on passe directement l'itérateur à `sum` (noter l'absence de crochets) :

```
sage: sum( binomial(8, i) for i in xrange(9) )
256
```

la fonction `sum` reçoit directement l'itérateur, et peut ainsi court-circuiter la construction de la liste intermédiaire. Lorsqu'il y a un grand nombre d'éléments, cela évite donc d'allouer une grosse quantité de mémoire pour stocker une liste qui sera immédiatement détruite[2].

La plupart des fonctions prenant une liste d'éléments en entrée acceptent un itérateur (ou un itérable) à la place. Et pour commencer, on peut obtenir la liste (ou le tuple) des éléments d'un itérateur avec :

```
sage: list(binomial(8, i) for i in xrange(9))
[1, 8, 28, 56, 70, 56, 28, 8, 1]
sage: tuple(binomial(8, i) for i in xrange(9))
(1, 8, 28, 56, 70, 56, 28, 8, 1)
```

Considérons maintenant les fonctions `all` and `any` (qui dénotent respectivement le *et* et le *ou* n-aire) :

```
sage: all([True, True, True, True])
True
sage: all([True, False, True, True])
False
sage: any([False, False, False, False])
False
sage: any([False, False, True, False])
True
```

L'exemple suivant vérifie que tous les entiers premiers entre 3 et 100 exclus sont impairs :

```
sage: all( is_odd(p) for p in range(3,100) if is_prime(p) )
True
```

Les nombres de Mersenne M_p sont les nombres de la forme $2^p - 1$. Ici nous vérifions, pour $p < 1000$, que si M_p est premier alors p est premier aussi :

[2] Détail technique : `xrange` renvoie un itérateur sur {0,...,8} alors que `range` en renvoie la liste. À partir de Python 3.0, `range` se comportera comme `xrange`, et on pourra oublier ce dernier.

```
sage: def mersenne(p): return 2^p -1
sage: [ is_prime(p)
....:   for p in range(1000) if is_prime(mersenne(p)) ]
[True, True, True, True, True, True, True, True, True, True,
 True, True, True, True]
```

La réciproque est-elle vraie ?

Exercice 46. Essayer les deux commandes suivantes et expliquer la différence considérable de temps de calcul :

```
sage: all(   is_prime(mersenne(p))
....:        for p in range(1000) if is_prime(p)  )
False
sage: all( [ is_prime(mersenne(p))
....:        for p in range(1000) if is_prime(p)] )
False
```

On cherche maintenant à trouver le plus petit contre-exemple. Pour cela on utilise la fonction Sage `exists` :

```
sage: exists( (p for p in range(1000) if is_prime(p)),
....:         lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

Alternativement, on peut construire un itérateur sur tous les contre-exemples :

```
sage: contre_exemples = \
....:   (p for p in range(1000)
....:      if is_prime(p) and not is_prime(mersenne(p)))
sage: contre_exemples.next()
11
sage: contre_exemples.next()
23
```

Exercice 47. Que font les commandes suivantes ?

```
sage: cubes = [t**3 for t in range(-999,1000)]
sage: exists([(x,y) for x in cubes for y in cubes],
....:        lambda (x,y): x+y == 218)
(True, (-125, 343))
sage: exists(((x,y) for x in cubes for y in cubes),
....:        lambda (x,y): x+y == 218)
(True, (-125, 343))
```

Laquelle des deux dernières est-elle la plus économe en temps ? En mémoire ? De combien ?

Exercice 48. Essayer tour à tour les commandes suivantes, et expliquer leurs résultats. Attention : il sera nécessaire d'interrompre l'exécution de certaines de ces commandes en cours de route.

```
sage: sum( x^len(s) for s in Subsets(8) )

sage: sum( x^p.length() for p in Permutations(3) )

sage: factor(sum( x^p.length() for p in Permutations(3) ))

sage: P = Permutations(5)
sage: all( p in P for p in P )
True

sage: for p in GL(2, 2): print p; print

sage: for p in Partitions(3): print p

sage: for p in Partitions(): print p

sage: for p in Primes(): print p

sage: exists( Primes(), lambda p: not is_prime(mersenne(p)) )
(True, 11)

sage: contre_exemples = (p for p in Primes()
....:                    if not is_prime(mersenne(p)))
sage: for p in contre_exemples: print p
```

Opérations sur les itérateurs

Python fournit de nombreux utilitaires pour manipuler des itérateurs ; la plupart d'entre eux sont dans la bibliothèque `itertools` que l'on peut importer avec :

```
import itertools
```

Nous en montrons quelques applications, en prenant comme point de départ l'ensemble des permutations de 3 :

```
sage: list(Permutations(3))
[[1, 2, 3], [1, 3, 2], [2, 1, 3],
 [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Nous pouvons énumérer les éléments d'un ensemble en les numérotant :

```
sage: list(enumerate(Permutations(3)))
[(0, [1, 2, 3]), (1, [1, 3, 2]), (2, [2, 1, 3]),
 (3, [2, 3, 1]), (4, [3, 1, 2]), (5, [3, 2, 1])]
```

sélectionner seulement les éléments en position 2, 3 et 4 (analogue de `l[1 :4]`) :

```
sage: import itertools
sage: list(itertools.islice(Permutations(3), 1, 4))
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]
```

appliquer une fonction sur tous les éléments :

```
sage: list(itertools.imap(lambda z: z.cycle_type(),
....:                     Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

ou sélectionner les éléments vérifiant un certain prédicat :

```
sage: list(itertools.ifilter(lambda z: z.has_pattern([1,2]),
....:                        Permutations(3)))
```

Dans toutes ces situations, `attrcall` est une alternative avantageuse à la création d'une fonction anonyme :

```
sage: list(itertools.imap(attrcall("cycle_type"),
....:                     Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

Implantation de nouveaux itérateurs

Il est possible de construire très facilement de nouveaux itérateurs, en utilisant le mot clef `yield` plutôt que `return` dans une fonction :

```
sage: def f(n):
....:     for i in range(n):
....:         yield i
```

À la suite du `yield`, l'exécution n'est pas arrêtée, mais seulement suspendue, et prête à reprendre au même point. Le résultat de la fonction est alors un itérateur sur les valeurs successives renvoyées par `yield` :

```
sage: g = f(4)
sage: g.next()
0
sage: g.next()
1
sage: g.next()
2
sage: g.next()
3

sage: g.next()
Traceback (most recent call last):
  ...
StopIteration
```

En utilisation courante, cela donnera :

```
sage: [ x for x in f(5) ]
[0, 1, 2, 3, 4]
```

Ce paradigme de programmation, appelé *continuation* est très utile en combinatoire, surtout quand on le combine avec la récursivité (voir aussi §7.2.2 pour d'autres applications). Voici comment engendrer tous les mots d'une longueur et sur un alphabet donnés :

```
sage: def words(alphabet,l):
....:     if l == 0:
....:         yield []
....:     else:
....:         for word in words(alphabet, l-1):
....:             for l in alphabet:
....:                 yield word + [l]
sage: [ w for w in words(['a','b'], 3) ]
[['a','a','a'], ['a','a','b'], ['a','b','a'], ['a','b','b'],
 ['b','a','a'], ['b','a','b'], ['b','b','a'], ['b','b','b']]
```

On peut les compter avec :

```
sage: sum(1 for w in words(['a','b','c','d'], 10))
1048576
```

Compter les mots un par un n'est évidemment pas une méthode efficace dans ce cas, puisque l'on pourrait utiliser la formule n^ℓ ; au moins cela ne consomme quasiment aucune mémoire.

On considère maintenant les mots de Dyck, c'est-à-dire les mots bien parenthésés en les lettres « (» et «) ». La fonction ci-dessous engendre tous les mots de Dyck d'une longueur donnée (ou la longueur est le nombre de paires de parenthèses), en utilisant la définition récursive disant qu'un mot de Dyck est soit vide, soit de la forme $(w_1)w_2$ avec w_1 et w_2 des mots de Dyck :

```
sage: def dyck_words(l):
...       if l==0:
...           yield ''
...       else:
...           for k in range(l):
...               for w1 in dyck_words(k):
...                   for w2 in dyck_words(l-k-1):
...                       yield '('+w1+')'+w2
```

Voici tous les mots de Dyck de longueur 4 :

```
sage: list(dyck_words(4))
['()()()()', '()()(())', '()(())()', '()((()))', '(())()()',
 '(())(())', '((()))()', '(((())))']
```

On retrouve, en les comptant, une suite bien connue :

```
sage: [ sum(1 for w in dyck_words(l)) for l in range(10) ]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

Exercice 49. Construire un itérateur sur l'ensemble C_n des arbres binaires complets à n feuilles (voir §12.1.2).

Indication : Sage 4.4.4 n'a pas encore de structure de donnée native pour représenter des arbres binaires complets. Une manière simple de représenter des arbres est de définir une variable formelle `Leaf` pour les feuilles et une fonction formelle `Node` d'arité 2 :

```
sage: var("Leaf")
sage: function("Node", nargs=2)
```

Le deuxième arbre de la figure 12.1 peut alors être représenté par l'expression :

```
sage: tr = Node(Node(Leaf, Node(Leaf, Leaf)), Leaf)
```

12.3 Constructions

Nous allons voir maintenant comment construire de nouveaux ensembles à partir de briques de base. En fait, nous avons déjà commencé à le faire lors de la construction de $\mathcal{P}(\mathcal{P}(\mathcal{P}(\{1,2,3,4\})))$ dans la section précédente, ou pour construire des ensembles de cartes en §12.1.

Considérons un produit cartésien un peu conséquent :

```
sage: C = CartesianProduct(Compositions(8), Permutations(20)); C
Cartesian product of Compositions of 8, Standard permutations of 20
sage: C.cardinality()
311411457046609920000
```

Il ne serait évidemment pas envisageable de construire la liste de tous les éléments de ce produit cartésien.

Pour l'instant, la construction `CartesianProduct` ignore les propriétés algébriques de ses arguments. Cela est partiellement corrigé avec Sage 4.4.4, avec la construction `cartesian_product`. À terme, les deux constructions seront fusionnées et, dans l'exemple suivant, H sera à la fois muni des opérations combinatoires usuelles, mais aussi de sa structure de groupe produit :

```
sage: G = DihedralGroup(4)
sage: H = cartesian_product([G,G])
```

Nous construisons maintenant la réunion de deux ensembles existants disjoints :

```
sage: C = DisjointUnionEnumeratedSets(
....:       [ Compositions(4), Permutations(3)] )
sage: C
Union of Family (Compositions of 4, Standard permutations of 3)
```

```
sage: C.cardinality()
14
sage: C.list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3],
 [2, 1, 1], [2, 2], [3, 1], [4],
 [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

On peut faire une union de plus de deux ensembles disjoints, voire même d'une infinité d'entre eux. Nous allons construire l'ensemble de toutes les permutations, vu comme la réunion de tous les ensembles P_n des permutations de taille n. On commence par construire la famille infinie $F = (P_n)_{n\in\mathbb{N}}$:

```
sage: F = Family(NonNegativeIntegers(), Permutations); F
Lazy family (Permutations(i))_{i in Non negative integers}
sage: F.keys()
Non negative integers
sage: F[1000]
Standard permutations of 1000
```

On peut maintenant construire la réunion disjoint $\bigcup_{n\in\mathbb{N}} P_n$:

```
sage: U = DisjointUnionEnumeratedSets(F); U
Disjoint union of
Lazy family (Permutations(i))_{i in Non negative integers}
```

C'est un ensemble infini :

```
sage: U.cardinality()
+Infinity
```

ce qui n'empêche pas d'itérer à travers ses éléments, quoiqu'il faille bien entendu interrompre le calcul à un moment donné :

```
sage: for p in U:
....:     print p
[]
[1]
[1, 2]
[2, 1]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
...
```

Note : on aurait pu construire l'ensemble ci-dessus directement avec :

```
sage: U = Permutations(); U
Standard permutations
```

12.3.1 Résumé

En résumé, Sage fournit une bibliothèque d'ensembles énumérés usuels, qui peuvent être combinés entre eux par les constructions usuelles, ce qui donne une boîte à outils flexible, quoique encore loin d'être aboutie. Il est de plus possible de rajouter en quelques lignes de nouvelles briques dans Sage (voir le code de `FiniteEnumeratedSets().example()`). Cela est rendu possible par l'uniformité des interfaces et le fait que Sage soit basé sur un langage orienté objet. D'autre part, on peut manipuler des ensembles très grands, voire infinis, grâce aux stratégies d'évaluation paresseuse (itérateurs, ...).

Il n'y a rien de magique : en arrière-boutique, Sage se contente d'appliquer les règles de calculs usuelles (par exemple, la cardinalité de $E \times E$ vaut $|E|^2$) ; la valeur ajoutée provient de la possibilité de manipuler des constructions compliquées. La situation est à rapprocher de l'analyse où, pour différencier une formule, Sage se contente d'appliquer les règles usuelles de différentiation des fonctions usuelles et de leur compositions, et où la valeur ajoutée vient de la possibilité de manipuler des formules compliquées. En ce sens, Sage implante un *calculus* sur les ensembles énumérés finis.

12.4 Algorithmes génériques

12.4.1 Génération lexicographique de listes d'entiers

Parmi les ensembles énumérés classiques, en particulier en combinatoire algébrique, un certain nombre sont composés de listes d'entiers de somme fixée comme par exemple les partitions, les compositions ou les vecteurs d'entiers. Ces ensembles peuvent de plus prendre des contraintes supplémentaires. Voici quelques exemples. Les vecteurs d'entiers de somme 10 et longueur 3, dont les parts sont entre 2 et 5, et bornées inférieurement par $[2, 4, 2]$:

```
sage: IntegerVectors(10, 3, min_part = 2, max_part = 5,
....:                inner = [2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]
```

Les compositions de 5 dont chaque part est au plus 3, et dont la longueur est entre 2 et 3 inclus :

```
sage: Compositions(5, max_part = 3,
....:              min_length = 2, max_length = 3).list()
[[1, 1, 3], [1, 2, 2], [1, 3, 1], [2, 1, 2], [2, 2, 1], [2, 3],
 [3, 1, 1], [3, 2]]
```

Les partitions de 5 strictement décroissantes :

```
sage: Partitions(5, max_slope = -1).list()
[[5], [4, 1], [3, 2]]
```

Ces ensembles partagent la même algorithmique sous-jacente, implantée dans la classe `IntegerListLex`. Cette dernière permet de modéliser des ensembles de vecteurs $\ell_0, \ldots, \ell_k$ d'entiers non négatifs, avec des contraintes de somme, de longueur et de bornes sur les parts et sur les différences entre parts consécutives. Voici quelques autres exemples :

```
sage: IntegerListsLex(10, length=3,
....:                 min_part = 2, max_part = 5,
....:                 floor = [2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]

sage: IntegerListsLex(5, min_part = 1, max_part = 3,
....:                 min_length = 2, max_length = 3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2],
 [1, 3, 1], [1, 2, 2], [1, 1, 3]]

sage: IntegerListsLex(5, min_part = 1, max_slope = -1).list()
[[5], [4, 1], [3, 2]]

sage: Compositions(5)[4]
[1, 2, 1, 1]
sage: IntegerListsLex(5, min_part = 1)[4]
[1, 2, 1, 1]
```

L'intérêt du modèle de `IntegerListsLex` provient du bon compromis entre généralité et efficacité de l'itération. L'algorithme principal permet en effet d'itérer à travers les éléments d'un tel ensemble S dans l'ordre lexicographique inverse, et en complexité constante amortie (CAT), sauf cas dégénéré ; en gros, le temps nécessaire pour parcourir tous les éléments est proportionnel au nombre de ces éléments, ce qui est optimal. De plus, la mémoire utilisée est proportionnelle au plus gros élément rencontré, c'est-à-dire négligeable en pratique.

Cet algorithme repose sur un principe très général de parcours d'arbre de décision (ou *branch and bound*) : au plus haut niveau, on parcourt tous les choix possibles pour la première part ℓ_0 du vecteur ; pour chaque choix pour ℓ_0, on parcourt récursivement tous les choix pour ℓ_1, et ainsi de suite. Mathématiquement parlant, on a mis une structure d'arbre préfixe sur les éléments de S : un nœud de l'arbre à la profondeur k correspond à un préfixe $\ell_0, \ldots, \ell_k$ d'un (ou plusieurs) éléments de S (voir figure 12.2)

Le problème usuel dans ce type d'approche est d'éviter les mauvaises décisions amenant à sortir de l'arbre préfixe et à l'exploration de branches mortes, ce d'autant que la croissance du nombre d'éléments avec la profondeur est exponentielle. Il se trouve que les contraintes listées ci-dessus sont suffisamment simples pour garantir la propriété suivante : étant donné un préfixe $\ell_0, \ldots, \ell_k$ de S, l'ensemble des ℓ_{k+1} tels que $\ell_0, \ldots, \ell_{k+1}$ est un préfixe de S est soit vide, soit un intervalle de la forme $[a, b]$, et les bornes a et b

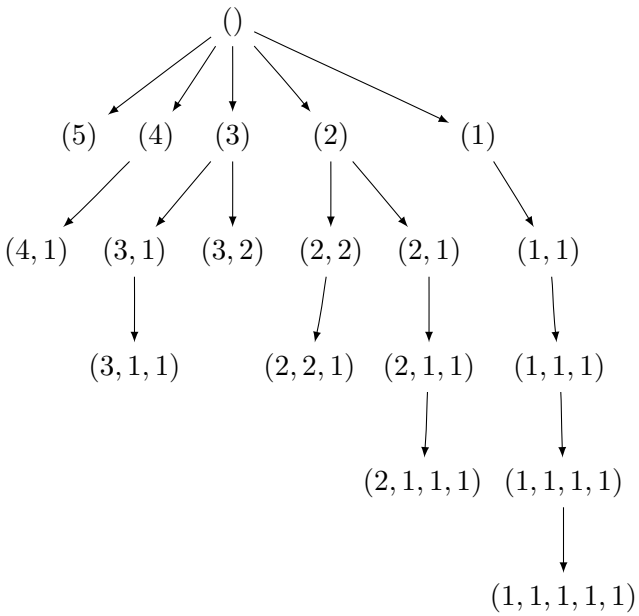


FIG. 12.2 – L'arbre préfixe des partitions de 5.

peuvent être calculées en temps linéaire en la longueur du plus long élément de S ayant $\ell_0, \ldots, \ell_k$ comme préfixe.

À noter : cet algorithme a été originellement développé et implanté dans `MuPAD-Combinat` et porté à l'identique sous `Sage` ; cette implantation n'est pas robuste en cas d'entrées subtilement incohérentes, ce qui peut donner des résultats surprenants. Par exemple, l'exemple suivant devrait donner les partitions de 2 de longueur 2 strictement décroissantes :

```
sage: Partitions(2, max_slope=-1, length=2).list()
[[1, 1]]
```

12.4.2 Points entiers dans les polytopes

Si l'algorithme d'itération de `IntegerListsLex` est efficace, son algorithme de comptage est naïf : il se contente d'itérer à travers tous les éléments.

Il y a une approche alternative pour traiter ce problème : modéliser les listes d'entiers désirées comme l'ensemble des points entiers d'un polytope, c'est-à-dire l'ensemble des solutions à coordonnées entières d'un système d'inéquations linéaires. C'est un cadre très général pour lequel il existe des algorithmes de comptage avancés (par ex. Barvinok), qui sont implantés dans des bibliothèques comme LattE. L'itération ne pose en principe pas de grosse difficulté. Il y a cependant deux limitations qui justifient l'existence de `IntegerListsLex`. La première est d'ordre théorique : les points d'entiers

d'un polytope ne permettent de modéliser que des problèmes en dimension (longueur) fixe ; la deuxième d'ordre pratique : à l'heure actuelle seule la bibliothèque PALP a une interface avec Sage ; si elle offre de multiples fonctionnalités sur l'étude des polytopes, pour ce qui nous intéresse ici elle ne permet que de construire la liste les points entiers, sans fournir d'itérateur ni de comptage non naïf :

```
sage: A=random_matrix(ZZ,3,6,x=7)
sage: L=LatticePolytope(A)
sage: L.points()
[1 6 6 2 5 5 6 5 4 5 4 5 4 2 5 3 4 5 3 4 5 3 3]
[4 4 2 6 4 1 3 2 3 3 4 4 3 4 3 4 4 4 4 4 4 5 5]
[3 1 1 6 5 1 1 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 5]
sage: L.npoints()
23
```

Ce polytope peut être visualisé en 3D avec `L.plot3d()` (voir figure 12.3).

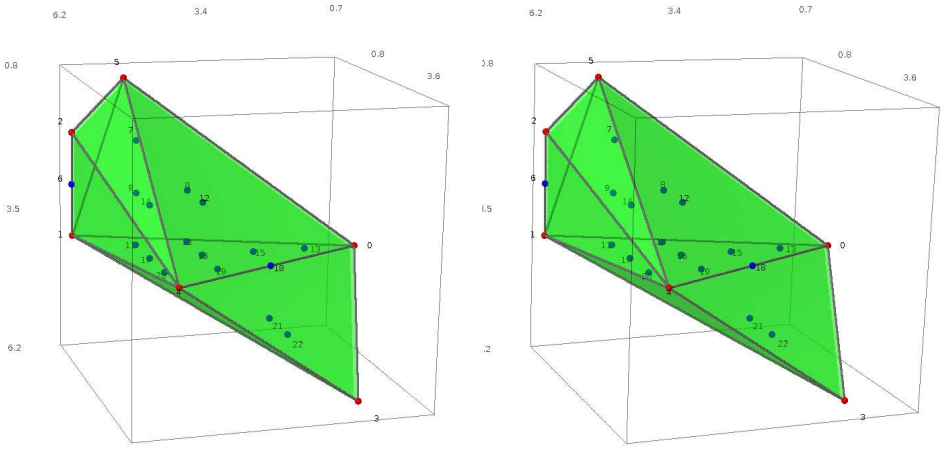


FIG. 12.3 – Le polytope L et ses points entiers, en vision stéréographique yeux croisés.

12.4.3 Espèces, classes combinatoires décomposables

En §12.1.2, nous avons montré comment utiliser la définition récursive des arbres binaires pour les dénombrer efficacement au moyen de séries génératrices. Les techniques exposées sont très générales, et s'appliquent dès qu'un ensemble peut être défini récursivement (selon les communautés, un tel ensemble est appelé classe combinatoire décomposable ou une espèce). Cela inclut toutes les variétés d'arbres, mais aussi les permutations, ...

Nous nous contentons ici d'illustrer quelques exemples d'utilisation de la bibliothèque de Sage sur les espèces :

```
sage: from sage.combinat.species.library import *
sage: o = var("o")
```

Nous commençons par redéfinir les arbres binaires complets ; pour cela, on stipule l'équation de récurrence directement sur les ensembles :

```
sage: BT = CombinatorialSpecies()
sage: Leaf =  SingletonSpecies()
sage: BT.define( Leaf + (BT*BT) )
```

On peut maintenant construire l'ensemble des arbres à cinq nœuds, et les lister, les compter, ... :

```
sage: BT5 = BT.isotypes([o]*5)
sage: BT5.cardinality()
14
sage: BT5.list()
[o*(o*(o*(o*o))), o*(o*((o*o)*o)), o*((o*o)*(o*o)),
 o*((o*(o*o))*o), o*(((o*o)*o)*o), (o*o)*(o*(o*o)),
 (o*o)*((o*o)*o), (o*(o*o))*(o*o), ((o*o)*o)*(o*o),
 (o*(o*(o*o)))*o, (o*((o*o)*o))*o, ((o*o)*(o*o))*o,
 ((o*(o*o))*o)*o, (((o*o)*o)*o)*o]
```

Les arbres sont construits en utilisant une structure de donnée récursive générique ; l'affichage ne peut donc être fameux ; pour faire mieux, il faudrait fournir à Sage une structure de donnée plus spécialisée, avec l'affichage désiré.

On retrouve la série génératrice des nombres de Catalan :

```
sage: g = BT.isotype_generating_series(); g
x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + O(x^6)
```

qui est renvoyée sous forme d'une série paresseuse :

```
sage: g[100]
227508830794229349661819540395688853956041682601541047340
```

Nous finissons avec les mots de Fibonacci, qui sont les mots binaires sans deux « 1 » consécutifs. Ils admettent une définition récursive naturelle :

```
sage: Eps =  EmptySetSpecies()
sage: Z0  =  SingletonSpecies()
sage: Z1  =  Eps*SingletonSpecies()
sage: FW  = CombinatorialSpecies()
sage: FW.define(Eps + Z0*FW  +  Z1*Eps + Z1*Z0*FW)
```

On reconnaît la fameuse suite de Fibonacci, d'où le nom :

```
sage: L = FW.isotype_generating_series().coefficients(15); L
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]

sage: sloane_find(L)
Searching Sloane's online database...
[[45, 'Fibonacci numbers: F(n) = F(n-1) + F(n-2),
 F(0) = 0, F(1) = 1, F(2) = 1, ...',
 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
```

ce qui est une conséquence directe de l'équation de récurrence. Là encore, on peut aussi générer immédiatement tous les mots de Fibonacci d'une longueur donnée, avec les mêmes limitations de l'affichage générique :

```
sage: FW3 = FW.isotypes([o]*3)
sage: FW3.list()
[o*(o*(o*{})), o*(o*(({}*o)*{})), o*((({}*o)*o)*{}),
 (({}*o)*o)*(o*{}), (({}*o)*o)*(({}*o)*{})]
```

Annexes

A

Solutions des exercices

A.1 Analyse et algèbre avec Sage

Exercice 1. (*Un calcul de somme par récurrence*)

```
var('n k'); p = 4; s = [n + 1]
for k in (1..p):
    s = s + [factor((((n + 1)^(k + 1) - sum(binomial(k + 1, j)*s[j]
             for j in (0..k - 1))) / (k + 1)))]
s
```

On obtient ainsi :

$$\sum_{k=0}^{n} k = \frac{1}{2}\,(n+1)n, \qquad \sum_{k=0}^{n} k^2 = \frac{1}{6}\,(n+1)(2\,n+1)n,$$
$$\sum_{k=0}^{n} k^3 = \frac{1}{4}\,(n+1)^2 n^2, \qquad \sum_{k=0}^{n} k^4 = \frac{1}{30}\,(n+1)(2\,n+1)\Big(3\,n^2+3\,n-1\Big)n.$$

Exercice 2. (*Un calcul symbolique de limite*) Pour répondre à la question, on va utiliser une fonction symbolique, dont on va calculer le polynôme de Taylor en 0 à l'ordre 3.

```
sage: var('x h a'); f = function('f', x)
sage: g(x) = taylor(f, x, a, 3)
sage: phi(h) = (g(a+3*h)-3*g(a+2*h)+3*g(a+h)-g(a))/h**3
sage: phi(h).expand()
```

$$D[0,0,0]\,(f)\,(a).$$

La fonction g diffère de f d'un reste qui est négligeable devant h^3, donc la fonction φ diffère du quotient étudié d'un reste qui est $o(1)$; donc φ a

pour limite en zéro la limite cherchée. En conclusion,

$$\lim_{h\to 0} \frac{1}{h^3}(f(a+3h) - 3f(a+2h) + 3f(a+h) - f(a)) = f'''(a).$$

Cette formule permet notamment d'effectuer un calcul approché de la dérivée tierce de f en a sans effectuer aucune dérivation.

On peut supposer que la formule se généralise sous la forme suivante :

$$\lim_{h\to 0} \frac{1}{h^n}\left(\sum_{k=0}^{n} \binom{n}{k} (-1)^{n-k}\, g(a+kh)\right) = f^{(n)}(a).$$

Pour tester cette formule pour de plus grandes valeurs de n, on peut alors facilement adapter ce qui précède :

```
sage: n = 7; var('x h a'); f = function('f', x)
sage: g(x) = taylor(f, x, a, n)
sage: phi(h) = sum(binomial(n,k)*(-1)^(n-k)*g(a+k*h)
                   for k in (0..n))/h**n
sage: phi(h).expand()
```

$$D[0,0,0,0,0,0,0]\,(f)\,(a)\,.$$

Exercice 3. (*Une formule due à Gauss*)

1. On utilise successivement les fonctions `trig_expand()` et `trig_simplify` :

```
sage: theta = 12 * arctan(1/38) + 20 * arctan(1/57)
            + 7 * arctan(1/239) + 24 * arctan(1/268)
sage: x = tan(theta)
sage: y = x.trig_expand()
sage: y.trig_simplify()
1
```

2. La fonction tangente est convexe sur l'intervalle $I = \left[0, \frac{\pi}{4}\right]$, donc sa courbe représentative est en-dessous de sa corde ; autrement dit $\forall x \in I,\ 0 \leqslant x \leqslant \frac{4}{\pi}$.

```
sage: M = 12*(1/38)+20*(1/57)+ 7*(1/239)+24*(1/268)
sage: M
37735/48039
```

On en déduit :

$$\begin{aligned}\theta &= 12\arctan\frac{1}{38} + 20\arctan\frac{1}{57} + 7\arctan\frac{1}{239} + 24\arctan\frac{1}{268}\\ &\leqslant 12\cdot\frac{1}{38} + 20\cdot\frac{1}{57} + 7\cdot\frac{1}{239} + 24\cdot\frac{1}{268}\\ &= \frac{4}{\pi}\cdot\frac{37735}{48039} \leqslant \frac{4}{\pi} < \frac{\pi}{2}.\end{aligned}$$

Donc $\theta \in I$; or $\tan\theta = 1 = \tan\frac{\theta}{4}$ et tan est injective sur I.
On en conclut $\theta = \frac{\pi}{4}$.

3.
```
sage: x = var('x')
sage: f(x) = taylor(arctan(x), x, 0, 21)
sage: approx = 4 * (12 * f(1/38) + 20 * f(1/57)
                + 7 * f(1/239) + 24 * f(1/268))
sage: approx.n(digits = 50); pi.n(digits = 50)
3.1415926535897932384626433832795028851616168852864
3.1415926535897932384626433832795028841971693993751
sage: approx.n(digits = 50) - pi.n(digits = 50)
9.6444748591132486785420917537404705292978817080880e-37
```

Exercice 4. (*Développement asymptotique d'une suite*) L'encadrement de x_n permet d'affirmer $x_n \sim \pi n$, soit $x_n = \pi n + o(n)$.

On injecte alors cette égalité dans l'équation suivante :

$$x_n = \pi n + \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right).$$

On réinjecte ensuite les développements asymptotiques de x_n ainsi obtenus dans cette équation, bis repetita... (méthode des raffinements successifs).

Sachant qu'à chaque étape, un développement d'ordre p permet d'obtenir un développement à l'ordre $p+2$, on peut obtenir en quatre étapes un développement d'ordre 6. En anticipant sur la section 3, on peut effectuer ces quatre étapes à l'intérieur d'une boucle.

```
1 var('n'); phi = lambda x: n*pi +pi/2 - arctan(1/x); x = pi * n
2 for i in range(4): x = taylor(phi(x), n, oo, 2 * i); x
```

Au final, on obtient :

$$x_n = \frac{1}{2}\pi + \pi n - \frac{1}{\pi n} + \frac{1}{2}\frac{1}{\pi n^2} - \frac{1}{12}\frac{(3\,\pi^2+8)}{\pi^3 n^3} + \frac{1}{8}\frac{(\pi^2+8)}{\pi^3 n^4} - \frac{1}{240}\frac{(15\,\pi^4+240\,\pi^2+208)}{\pi^5 n^5} + \frac{1}{96}\frac{(3\,\pi^4+80\,\pi^2+208)}{\pi^5 n^6} + o\left(\frac{1}{n^6}\right)$$

Exercice 5. (*Expression du Laplacien en coordonnées polaires*)
```
sage: x, y, r, t = var('x, y, r, t'); f = function('f',x,y)
sage: F = f(x = r*cos(t), y = r*sin(t))
sage: d = (diff(F,r,2) + diff(F,t,2)/r**2 + diff(F,r)/r)
sage: d.simplify_full()
D[0, 0](f)(r, t) + D[1, 1](f)(r, t)
```

On en déduit :

$$\frac{\partial^2 F}{\partial \rho^2} + \frac{1}{\rho^2}\frac{\partial^2 F}{\partial \theta^2} + \frac{1}{\rho}\frac{\partial F}{\partial \rho} = \Delta f.$$

Exercice 6. (*Un contre-exemple dû à Péano au théorème de Schwarz*) Les applications partielles en $(0,0)$ sont identiquement nulles ; on en déduit, sans

calculs, $\partial_1 f(0,0) = \partial_2 f(0,0) = 0$. On calcule alors les valeurs des dérivées partielles secondes en $(0,0)$:

```
sage: f(x, y) = x * y * (x**2 - y**2) / (x**2 + y**2)
sage: D1f(x, y) = diff(f(x,y), x)
sage: limit((D1f(0,h) - 0) / h, h=0)
-1
sage: D2f(x, y) = diff(f(x,y), y)
sage: limit((D2f(h,0) - 0) / h, h=0)
1
sage: g = plot3d(f(x, y), (x, -3, 3), (y, -3, 3))
```

On en déduit $\partial_1\partial_2 f(0,0) = 1$ et $\partial_2\partial_1 f(0,0) = -1$. Donc cette fonction fournit un contre-exemple au théorème de Schwarz.

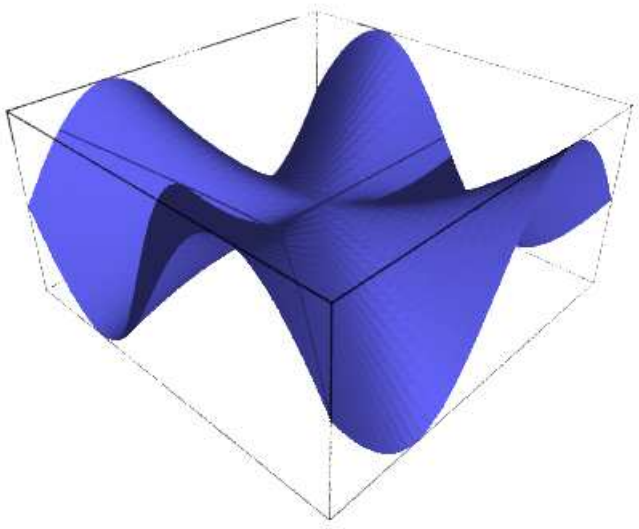


FIG. A.1 – La surface de Peano.

Exercice 7. (*Formule de Simon Plouffe*)

1. On commence par comparer

$$u_n = \int_0^{1/\sqrt{2}} f(t)\cdot t^{8n}\,\mathrm{d}t \quad \text{et} \quad v_n = \left(\tfrac{4}{8n+1} - \tfrac{2}{8n+4} - \tfrac{1}{8n+5} - \tfrac{1}{8n+6}\right)\left(\tfrac{1}{16}\right)^n.$$

```
sage: var('n')
sage: v(n) = (4/(8*n+1)-2/(8*n+4)-1/(8*n+5)-1/(8*n+6))*1/16^n
sage: assume(8*n+1>0)
sage: u(n) = integrate((4*sqrt(2)-8*t^3-4*sqrt(2)*t^4-8*t^5)\
                       * t^(8*n), t, 0, 1/sqrt(2))
sage: (u(n)-v(n)).simplify_full()
0
```

On en déduit $u_n = v_n$. Par linéarité de l'intégrale, on obtient :

$$I_N = \int_0^{1/\sqrt{2}} f(t) \cdot \left(\sum_{n=0}^{N} t^{8n} \right) \mathrm{d}t = \sum_{n=0}^{N} u_n = \sum_{n=0}^{N} v_n = S_N.$$

2. La série entière $\left(\sum_{n \geqslant 0} t^{8n} \right)$ a pour rayon de convergence 1, donc elle converge normalement sur le segment $\left[0, \frac{1}{\sqrt{2}}\right]$. Sur ce segment, on peut donc intervertir l'intégrale et la limite :

$$\begin{aligned} \lim_{N \to \infty} S_N &= \lim_{N \to \infty} \int_0^{1/\sqrt{2}} f(t) \cdot \left(\sum_{n=0}^{N} t^{8n} \right) \mathrm{d}t \\ &= \int_0^{1/\sqrt{2}} f(t) \cdot \left(\sum_{n=0}^{\infty} t^{8n} \right) \mathrm{d}t \\ &= \int_0^{1/\sqrt{2}} f(t) \cdot \frac{1}{1-t^8} \, \mathrm{d}t. \end{aligned}$$

Donc $\lim_{N \to +\infty} S_N = J$.

3. On procède ensuite au calcul de J :

```
sage: J = integrate((4*sqrt(2)-8*t^3-4*sqrt(2)*t^4-8*t^5)\
                    / (1-t^8), t, 0, 1/sqrt(2))
sage: J.simplify_full()
```

$$\pi + 2 \log\left(\sqrt{2} - 1\right) + 2 \log\left(\sqrt{2} + 1\right)$$

Pour simplifier cette expression, on va utiliser une astuce : Sage ne sait pas factoriser une somme de logarithmes ; en revanche il sait développer le logarithme d'un produit. Il nous suffit donc de passer à l'exponentielle.

```
sage: ln(exp(J).simplify_log())
pi
```

En définitive, on obtient la formule demandée :

$$\sum_{n=0}^{+\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n = \pi \, .$$

À l'aide de cette formule, donnons de nouveau une valeur approchée de π :

```
sage: l = sum(v(n) for n in (0..40)); l.n(digits=60)
3.14159265358979323846264338327950288419716939937510581474759
sage: pi.n(digits=60)
3.14159265358979323846264338327950288419716939937510582097494
sage: print "%e" % (l-pi).n(digits=60)
-6.227358e-54
```

Exercice 8. (*Approximation polynomiale du sinus*) On munit l'espace vectoriel $\mathcal{C}^\infty([-\pi,\pi])$ du produit scalaire $\langle f \,|\, g\rangle = \int_{-\pi}^{\pi} fg$. Le polynôme cherché est la projection orthogonale de la fonction sinus sur le sous-espace vectoriel $\mathbb{R}_5[X]$. La détermination de ce polynôme se ramène à la résolution d'un système linéaire : en effet, P est le projeté du sinus si et seulement si la fonction $(P - \sin)$ est orthogonale à chacun des vecteurs de la base canonique de $\mathbb{R}_5[X]$. Voici le code Sage :

```
var('X'); ps = lambda f,g : integral(f * g, X, -pi, pi)
n = 5; Q = sin(X)
var('a a0 a1 a2 a3 a4 a5'); a= [a0, a1, a2, a3, a4, a5]
P = sum(a[k] * X^k for k in (0..n))
equ = [ps(P - Q, X^k) for k in (0..n)]
sol = solve(equ, a)
P = sum(sol[0][k].rhs() * X^k for k in (0..n))
g = plot(P,X,-4,4,color='red') + plot(Q,X,-4,4,color='blue')
```

Le polynôme cherché est donc :

$$P = \frac{105}{8}\,\frac{\left(\pi^4 - 153\,\pi^2 + 1485\right)X}{\pi^6} - \frac{315}{4}\,\frac{\left(\pi^4 - 125\,\pi^2 + 1155\right)X^3}{\pi^8} + \frac{693}{8}\,\frac{\left(\pi^4 - 105\,\pi^2 + 945\right)X^5}{\pi^{10}}.$$

On peut ensuite tracer la fonction sinus et son projeté orthogonal pour apprécier la qualité de cette approximation polynomiale (cf. fig. A.2).

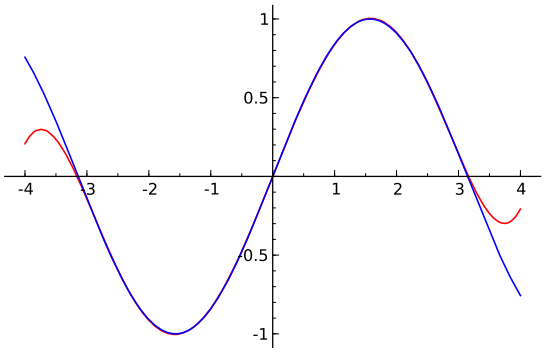


FIG. A.2 – Approximation du sinus par la méthode des moindres carrés.

Exercice 9. (*Le problème de Gauss*) On prouve tout d'abord formellement les relations demandées. S'ensuivra l'application numérique. On commence par définir les vecteurs $\overrightarrow{r_i}$:

```
var('p e theta1 theta2 theta3')
r(theta) = p / (1-e * cos(theta))
```

```
r1 = r(theta1); r2 = r(theta2); r3 = r(theta3)
R1 = vector([r1 * cos(theta1), r1 * sin(theta1), 0])
R2 = vector([r2 * cos(theta2), r2 * sin(theta2), 0])
R3 = vector([r3 * cos(theta3), r3 * sin(theta3), 0])
```

• On vérifie que $\overrightarrow{S} + e \cdot \left(\overrightarrow{\imath} \wedge \overrightarrow{D}\right)$ est le vecteur nul :

```
D = R1.cross_product(R2) + R2.cross_product(R3) + R3.cross_product(R1)
i = vector([1, 0, 0])
S = (r1 - r3) * R2 + (r3 - r2) * R1 +   (r2 - r1) * R3
V =  S + e * i.cross_product(D)
map(lambda x:x.simplify_full(), V) # rép. : [0, 0, 0]
```

D'où la relation demandée. On en déduit : $e = \frac{\left\| \overrightarrow{S} \right\|}{\left\| \overrightarrow{\imath} \wedge \overrightarrow{S} \right\|} = \frac{\left\| \overrightarrow{S} \right\|}{\left\| \overrightarrow{S} \right\|}$, puisque $\overrightarrow{D}$ est normal au plan de l'orbite, et donc à $\overrightarrow{\imath}$.

• On vérifie ensuite que $\overrightarrow{S} \wedge \overrightarrow{D}$ est colinéaire à $\overrightarrow{\imath}$:

```
map(lambda x:x.simplify_full(), S.cross_product(D))
```

En effet, le résultat renvoyé montre que les deuxième et troisième composantes sont nulles.

• De même, on vérifie que $-p \cdot \overrightarrow{S} + e \cdot \left(\overrightarrow{\imath} \wedge \overrightarrow{N}\right)$ est le vecteur nul :

```
N = r3 * R1.cross_product(R2)\
  + r1 * R2.cross_product(R3)\
  + r2 * R3.cross_product(R1)
W =  p * S + e * i.cross_product(N)
print map(lambda x:x.simplify_full(), W)  # rép. : [0, 0, 0]
```

D'où la relation demandée. On en déduit :

$$p = e\frac{\left\| \overrightarrow{\imath} \wedge \overrightarrow{N} \right\|}{\left\| \overrightarrow{S} \right\|} = e\frac{\left\| \overrightarrow{N} \right\|}{\left\| \overrightarrow{S} \right\|} = \frac{\left\| \overrightarrow{N} \right\|}{\left\| \overrightarrow{D} \right\|},$$

puisque $\overrightarrow{N}$ est normal au plan de l'orbite, et donc à $\overrightarrow{\imath}$.

• D'après le formulaire sur les coniques, on a $a = \frac{p}{1-e^2}$.

• On passe à présent à l'application numérique demandée :

```
R1=vector([0,1.,0]);R2=vector([2.,2.,0]);R3=vector([3.5,0,0])
r1 = R1.norm(); r2 = R2.norm(); r3 = R3.norm()
D = R1.cross_product(R2) + R2.cross_product(R3) \
  + R3.cross_product(R1)
S = (r1 - r3) * R2 + (r3 - r2) * R1 + (r2 - r1) * R3
V =  S + e * i.cross_product(D)
N = r3 * R1.cross_product(R2) + r1 * R2.cross_product(R3) \
```

```
25      + r2 * R3.cross_product(R1)
26    W =  p * S + e * i.cross_product(N)
27    e = S.norm() / D.norm()
28    p = N.norm() / D.norm()
29    a = p/(1-e^2)
30    c = a * e
31    b = sqrt(a^2 - c^2)
32    X = S.cross_product(D)
33    i = X / X.norm()
34    phi = atan2(i[1],i[0]) * 180 / pi.n()
35    print "%.3f %.3f %.3f %.3f %.3f %.3f" % (a, b, c, e, p, phi)
```

En conclusion, on trouve :

$$a \approx 2.36, \quad b \approx 1.33, \quad c \approx 1.95, \quad e \approx 0.827, \quad p \approx 0.745, \quad \varphi \approx 17.917.$$

L'inclinaison du grand axe par rapport à l'axe des abscisses vaut 17.92°.

Exercice 10. (*Bases de sous-espaces vectoriels*)

1. L'ensemble $\mathcal{S}$ des solutions du système linéaire homogène associé à A est un sous-espace vectoriel de $\mathbb{R}^5$, dont on obtient la dimension et une base grâce à la fonction `right_kernel` :

```
sage: A = matrix(QQ, [[2, -3, 2, -12, 33],
                      [ 6, 1, 26, -16, 69],
                      [10, -29, -18, -53, 32],
                      [2, 0, 8, -18, 84]])
sage: A.right_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[     1      0  -7/34   5/17   1/17]
[     0      1  -3/34 -10/17  -2/17]
```

   $\mathcal{S}$ est donc le plan vectoriel engendré par les deux vecteurs précédents.

2. On extrait de la famille génératrice donnée une base du sous-espace recherché de la manière suivante. On réduit la matrice A (formée par les colonnes des u_i) par rapport aux lignes jusqu'à la forme d'Hermite :

```
sage: H = A.echelon_form()
```

$$\begin{pmatrix} 1 & 0 & 4 & 0 & -3 \\ 0 & 1 & 2 & 0 & 7 \\ 0 & 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

   Soit $F = \mathrm{Vect}(u_1, u_2, u_3, u_4, u_5)$ la famille des vecteurs colonnes de A. C'est un sous-espace vectoriel de $\mathbb{R}^4$. Sur H, on observe que les pivots

sont dans les colonnes 1, 2 et 4. Plus précisément, on a :

$$\begin{cases} (u_1, u_2, u_4) \text{ est une famille libre,} \\ u_3 = 4u_1 + 2u_2, \\ u_5 = -3u_1 + 7u_2 - 5u_4. \end{cases}$$

Donc $F = \mathrm{Vect}(u_1, u_2, u_3, u_4, u_5) = \mathrm{Vect}(u_1, u_2, u_4)$ est engendré par la famille (u_1, u_2, u_4) ; or cette famille est libre ; donc (u_1, u_2, u_4) est une base de F. On aurait également pu utiliser, plus directement, la fonction `column_space` :

```
sage: A.column_space()
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[       1        0        0 1139/350]
[       0        1        0    -9/50]
[       0        0        1   -12/35]
```

3. On cherche à présent des équations du sous-espace engendré. Pour cela, on réduit la matrice A augmentée d'un second membre, en indiquant à Sage qu'on travaille dans un anneau de polynômes à quatre indeterminées :

```
sage: S.<x,y,z,t>=QQ[]
sage: C = matrix(S, 4,1,[x,y,z,t])
sage: B = block_matrix([A,C], ncols=2)
sage: C = B.echelon_form()
sage: C[3,5]*350
```

$$-1139\,x + 63\,y + 120\,z + 350\,t$$

On en déduit que F est l'hyperplan de $\mathbb{R}^4$ d'équation

$$-1139\,x + 63\,y + 120\,z + 350\,t = 0.$$

On aurait également pu obtenir cette équation en calculant le noyau à droite de A, qui donne les coordonnées des formes linéaires définissant F (ici il n'y en a qu'une) :

```
sage: K = A.kernel(); K
Vector space of degree 4 and dimension 1 over Rational Field
Basis matrix:
[        1  -63/1139 -120/1139 -350/1139]
```

L'hyperplan défini par cette forme linéaire a pour base les quatre vecteurs suivants :

```
sage: matrix(K.0).right_kernel()
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[       1        0        0 1139/350]
```

```
[       0        1        0    -9/50]
[       0        0        1   -12/35]
```

Exercice 11. (*Une équation matricielle*) On commence par définir les matrices A et C :

```
sage: A = matrix(QQ, [[-2, 1, 1], [8, 1, -5], [4, 3, -3]])
sage: C = matrix(QQ, [[1, 2, -1], [2, -1, -1], [-5, 0, 3]])
```

L'équation $A = BC$ est une équation linéaire, donc l'ensemble des solutions est un sous-espace affine de $\mathcal{M}_3(\mathbb{R})$. On cherche donc une solution particulière de notre équation.

```
sage: B = C.solve_left(A); B
```

$$\begin{pmatrix} 0 & -1 & 0 \\ 2 & 3 & 0 \\ 2 & 1 & 0 \end{pmatrix}$$

Ensuite, on détermine la forme générale des solutions de l'équation homogène, autrement dit, le noyau à gauche de C.

```
sage: C.left_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 1]
```

Enfin, on donne la forme générale des solutions de notre équation :

```
sage: var('x y z'); v = matrix([[1, 2, 1]])
sage: B = B+(x*v).stack(y*v).stack(z*v); B
```

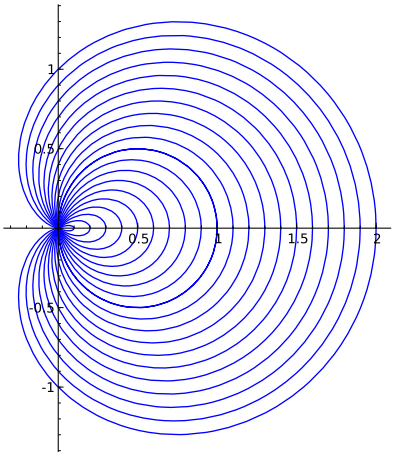
$$\begin{pmatrix} x & 2\,x-1 & x \\ y+2 & 2\,y+3 & y \\ z+2 & 2\,z+1 & z \end{pmatrix}$$

On peut effectuer une vérification rapide :

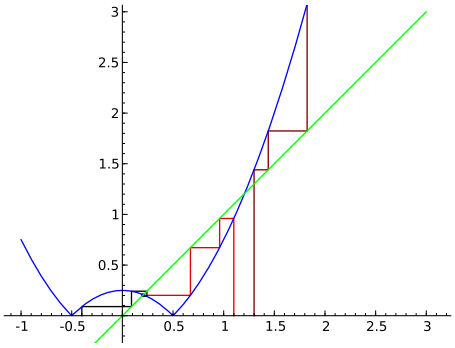
```
sage: A == B*C
True
```

En conclusion, l'ensemble des solutions est un sous-espace affine de dimension 3 :

$$\left\{ \begin{pmatrix} x & 2\,x-1 & x \\ y+2 & 2\,y+3 & y \\ z+2 & 2\,z+1 & z \end{pmatrix} \;\middle|\; (x,y,z) \in \mathbb{R}^3 \right\}.$$



(A) Conchoïdes de Pascal.



(B) Étude d'une suite récurrente.

A.1.1 Programmation

A.1.2 Graphiques

Exercice 12. (*Conchoïdes de Pascal*)

```
t = var('t'); liste = [a + cos(t) for a in srange(0, 10, 0.1)]
g = polar_plot(liste, (t, 0, 2 * pi)); g.show(aspect_ratio=1)
```

Exercice 13. (*Tracé des termes d'une suite récurrente*)

```
f = lambda x: abs(x**2 - 1/4)
def liste_pts(u0, n):
    u = u0; liste = [[u0,0]]
    for k in range(n):
        v, u = u, f(u)
        liste.extend([[v,u], [u,u]])
    return(liste)
g = line(liste_pts(1.1, 8), rgbcolor = (.9,0,0))
g += line(liste_pts(-.4, 8), rgbcolor = (.01,0,0))
g += line(liste_pts(1.3, 3), rgbcolor = (.5,0,0))
g += plot(f, -1, 3, rgbcolor = 'blue')
g += plot(x, -1, 3, rgbcolor = 'green')
g.show(aspect_ratio=1, ymin = -.2, ymax = 3)
```

Exercice 14. (*Équation différentielle linéaire, du premier ordre, non résolue*)

```
x = var('x'); y = function('y',x); DE = x^2 * diff(y, x) -  y == 0
desolve(DE, [y,x])   # rép. : c*e^(-1/x)
g = plot([c*e^(-1/x) for c in srange(-8, 8, 0.4)], (x, -3, 3))
y = var('y'); g += plot_vector_field((x^2, y), (x,-3,3), (y,-5,5))
g.show(ymin = -5, ymax = 5)
```

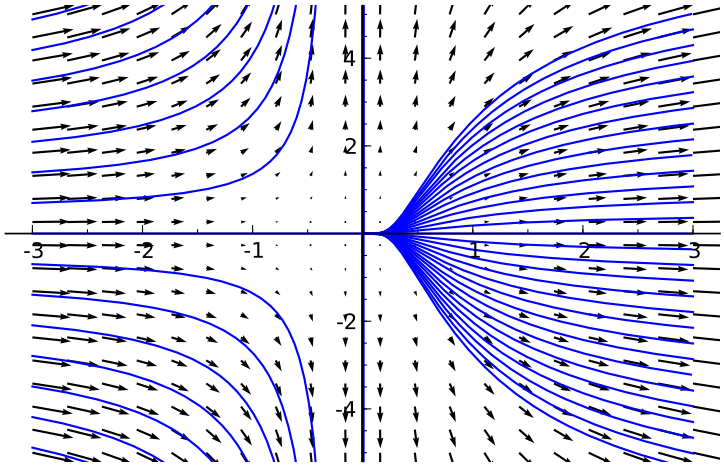


FIG. A.3 – Courbes intégrales de $x^2y' - y = 0$.

Exercice 15. (*Modèle proie-prédateur*)

```
from sage.calculus.desolvers import desolve_system_rk4
f = lambda x, y:[a*x-b*x*y,-c*y+d*b*x*y]
x,y,t = var('x y t')
a, b, c, d = 1., 0.1, 1.5, 0.75
P = desolve_system_rk4(f(x,y),[x,y],
       ics=[0,10,5],ivar=t,end_points=15)
Ql = [ [i,j] for i,j,k in P]
p = line(Ql,color='red')
p += text("Lapins",(12,37),fontsize=10,color='red')
Qr = [ [i,k] for i,j,k in P]
p += line(Qr,color='blue')
p += text("Renards",(12,7),fontsize=10,color='blue')
p.axes_labels(["temps","population"])
p.show(gridlines=True)
### Deuxième graphique
n = 10;  L = srange(6, 18, 12 / n); R = srange(3, 9, 6 / n)
def g(x,y):
    v = vector(f(x, y))
    return v/v.norm()
q = plot_vector_field(g(x, y), (x, 0, 60), (y, 0, 36))
for j in range(n):
    P = desolve_system_rk4(f(x,y),[x,y],
           ics=[0,L[j],R[j]],ivar=t,end_points=15)
    Q = [ [j,k] for i,j,k in P]
    q += line(Q, color=hue(.8-j/(2*n)))
q.axes_labels(["nombre de lapins","nombre de renards"])
```

```
27    q.show()
```

Exercice 16. (*Un système différentiel autonome*)

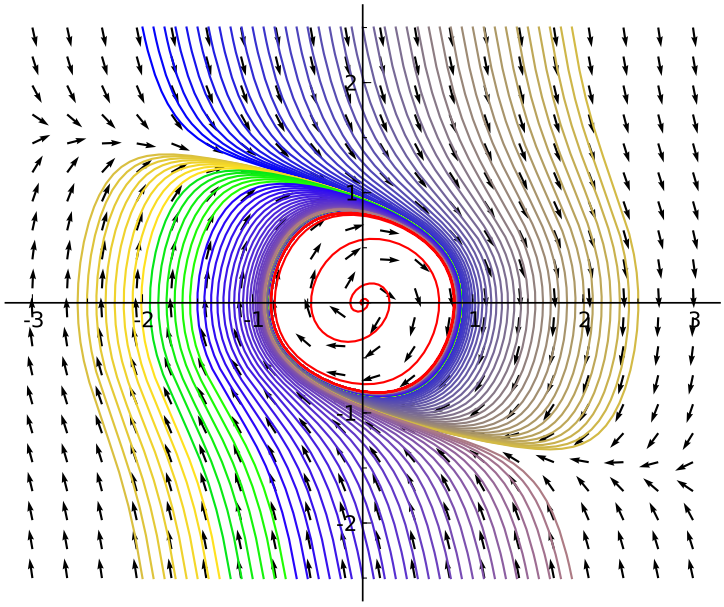


FIG. A.4 – Un système différentiel autonome.

```
1     import scipy; from scipy import integrate
2     def dX_dt(X, t=0):
3         return [X[1], 0.5*X[1] - X[0] - X[1]**3]
4
5     t = srange(0, 40, 0.01);  x0 = srange(-2, 2, 0.1)
6     y0 = 2.5
7     CI = [[i, y0] for i in x0] + [[i, -y0] for i in x0]
8
9     def g(x,y):
10        v = vector(dX_dt([x, y]))
11        return v/v.norm()
12
13    x, y = var('x, y')
14    q = plot_vector_field(g(x, y), (x, -3, 3), (y, -y0, y0))
15    for j in xrange(len(CI)):
16        X = integrate.odeint(dX_dt, CI[j], t)
17        q += line(X, color=(1.7*j/(4*n),1.5*j/(4*n),1-3*j/(8*n)))
18
19    X = integrate.odeint(dX_dt, [0.01,0], t)
```

```
  q += line(X, color = 'red')
  q.show()
```

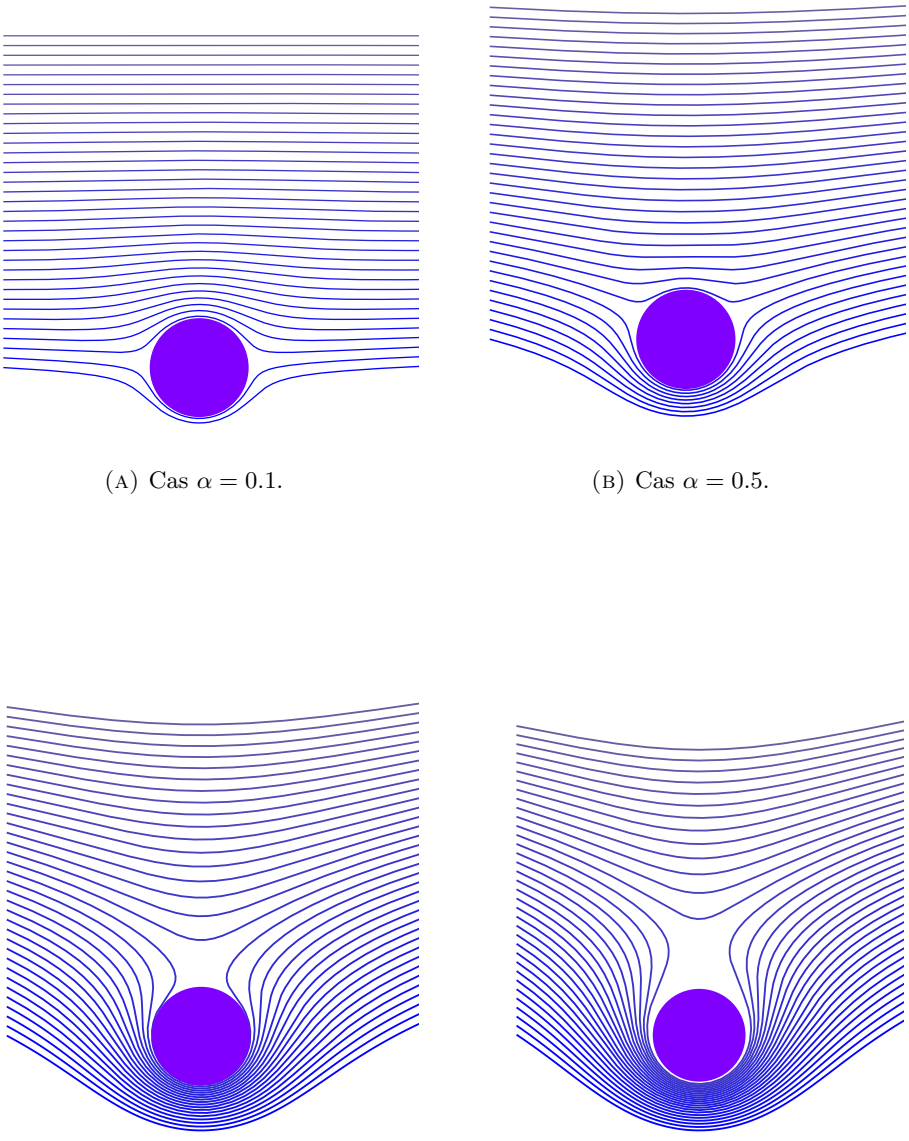
Exercice 17. (*Écoulement autour d'un cylindre avec effet Magnus*) Pour résoudre cet exercice, on peut par exemple utiliser la fonction `odeint` de SciPy :

```
import scipy
from scipy import integrate
# Intervalle de temps :
t = srange(0, 40, 0.01)
# Conditions initiales en cartésiennes :
n = 35; CI_cart = [[4, .2 * i] for i in range(n)]
# Conditions initiales en polaires :
CI = map(lambda x:[sqrt(x[0]**2+x[1]**2),\
     pi - arctan(x[1] / x[0])], CI_cart)
# Choix du paramètre alpha :
alpha = [0.1, 0.5, 1, 1.25]
for i in range(len(alpha)):
    # Définition du sytème différentiel :
    def dX_dt(X, t=0):
        return [cos(X[1]) * (1 - 1 / X[0]^2),\
        -sin(X[1]) * (1 / X[0] + 1 / X[0]^3) \
        + 2 * alpha[i] / X[0]^2]
    # tracé du disque :
    q = circle((0, 0), 1, fill=True, rgbcolor='purple')
    for j in range(n):
        # résolution du sytème autonome :
        X = integrate.odeint(dX_dt, CI[j], t)
        # passage en cartésiennes :
        Y = [[u[0] * cos(u[1]), u[0] * sin(u[1])] for u in X]
        # tracé stocké dans la variable q
        q += line(Y, xmin = -4, xmax = 4, color='blue')
    # Tracé final :
    q.show(aspect_ratio = 1, axes = False)
    q.save('effet_Magnus'+str(i)+'.pdf',\
            aspect_ratio = 1, axes = False)
```

A.2 Algèbre linéaire numérique

Exercice 18. On considère la factorisation de Cholesky $A = C\,{}^tC$, puis la décomposition en valeurs singulières de C : $C = U\Sigma\,{}^tV$. Alors, $X = U\Sigma\,{}^tU$. En effet : $A = C\,{}^tC = (U\Sigma\,{}^tV)(V\Sigma\,{}^tU) = U\Sigma\,{}^tU\,U\Sigma\,{}^tU = X^2$.

```
# fabriquer une matrice symétrique définie positive:
```



(A) Cas $\alpha = 0.1$.

(B) Cas $\alpha = 0.5$.

(C) Cas $\alpha = 1$.

(D) Cas $\alpha = 1.25$.

FIG. A.5 – Effet Magnus.

```
m = random_matrix(RDF,4)
a = transpose(m)*m
c = a.cholesky_decomposition()
U,S,V = c.SVD()
X = U*S*transpose(U)
# verification
print(X*X-a)
```

A.3 Intégration numérique

Exercice 19. (*Calcul des coefficients de Newton-Cotes*)

1. En remarquant que P_i est de degré $n-1$ (donc la formule (6.1) s'applique) et que $P_i(j) = 0$ pour $j \in \{0, \dots, n-1\}$ et $j \neq i$, on en déduit
$$\int_0^{n-1} P_i(x)\,\mathrm{d}x = w_i P_i(i)$$
soit
$$w_i = \frac{\int_0^{n-1} P_i(x)\,\mathrm{d}x}{P_i(i)}.$$

2. Le calcul des poids s'en déduit simplement :

```
def NCRule(n):
    P = prod([x - j for j in xrange(n)])
    return [integrate(P / (x-i), x, 0, n-1) \
      / (P/(x-i)).subs(x=i) for i in xrange(n)]
```

3. Par un simple changement de variable :
$$\int_a^b f(x)\,\mathrm{d}x = \frac{b-a}{n-1}\int_0^{n-1} f\left(a + \frac{b-a}{n-1}u\right)\mathrm{d}u.$$

4. En appliquant la formule précédente, on trouve le programme suivant :

```
def QuadNC(f, a, b, n):
    W = NCRule(n)
    ret = 0
    for i in xrange(n):
        ret += f(a + (b-a)/(n-1)*i) * W[i]
    return (b-a)/(n-1)*ret
```

Avant de comparer la précision de cette méthode avec d'autres, nous pouvons déjà vérifier qu'elle ne renvoie pas de résultats incohérents :

```
sage: QuadNC(lambda u: 1, 0, 1, 12)
1
sage: N(QuadNC(sin, 0, pi, 10))
1.99999989482634
```

Comparons succinctement la méthode obtenue avec les fonctions de GSL sur les intégrales I_2 et I_3 :

```
sage: numerical_integral(x * log(1+x), 0, 1)
(0.25, 2.7755575615628914e-15)
N(QuadNC(lambda x: x * log(1+x), 0, 1, 19)
0.250000000000000
sage: numerical_integral(sqrt(1-x^2), 0, 1)
(0.78539816772648219, 9.0427252245671193e-07)
sage: N(pi/4)
0.785398163397448
sage: N(QuadNC(lambda x: sqrt(1-x^2), 0, 1, 20))
0.784586419900198
```

Remarquons que la qualité du résultat dépend du nombre de points utilisés :

```
sage: [N(QuadNC(lambda x: x * log(1+x), 0, 1, n) - 1/4)
       for n in [2, 8, 16]]
[0.0965735902799726, 1.17408932943930e-7, 2.13546194616221e-13]
sage: [N(QuadNC(lambda x: sqrt(1-x^2), 0, 1, n) - pi/4)
       for n in [2, 8, 16]]
[-0.285398163397448, -0.00524656673640445, -0.00125482109302663]
```

Une comparaison plus intéressante entre les différentes fonctions d'intégration de Sage et notre méthode `QuadNC` demanderait de la convertir en une méthode adaptative qui subdivise automatiquement l'intervalle considéré comme le fait `numerical_integral`.

A.4 Équations non linéaires

Exercice 20. On a vu que pour stopper l'exécution de la fonction il faut utiliser le mot clé `return`. Il suffit donc de tester si `f(u)` est nul ou pas. Pour éviter d'évaluer la fonction `f` en `u` à plusieurs reprises, on stocke sa valeur dans une variable. Après modification on obtient donc la fonction suivante.

```
def intervalgen(f, phi, s, t):
    msg = 'Wrong arguments: f({0}) * f({1}) >= 0)'.format(s, t)
    assert (f(s) * f(t) < 0), msg
    yield s
    yield t
    while 1:
        u = phi(s, t)
        yield u
        fu = f(u)
        if fu == 0:
            return
```

```
        if fu * f(s) < 0:
            t = u
        else:
            s = u
```

Testons cette fonction avec une équation dont on connaît une solution, par exemple construite à partir d'une fonction linéaire.

```
sage: f(x) = 4 * x - 1
sage: a, b = 0, 1
sage: phi(s, t) = (s + t) / 2
sage: bisection = intervalgen(f, phi, a, b)
sage: for x in bisection:
....:     print(x)
0
1
1/2
1/4
```

Exercice 21. La fonction `phi` passée en paramètre de `intervalgen` détermine le point où diviser un intervalle. Il suffit donc de donner à cette fonction la définition adéquate.

```
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: a, b = RR(-pi), RR(pi)
sage: phi(s, t) = RR.random_element(s, t)
sage: random = intervalgen(f, phi, a, b)
sage: try:
....:     iterate(random)
....: except RuntimeError:
....:     print('Sequence failed to converge')
Sequence failed to converge
sage: random = intervalgen(f, phi, a, b)
sage: try:
....:     iterate(random, maxiter=10000)
....: except RuntimeError:
....:     print('Sequence failed to converge')
'After 30 iterations: 2.15850191762026'
```

Exercice 22.

```
from collections import deque
basering = PolynomialRing(SR, 'x')
def quadraticgen(f, r, s):
    t = (r + s) / 2
    yield t
    points = deque([(r, f(r)), (s, f(s)), (t, f(t))], maxlen=3)
    while 1:
        temp = basering.lagrange_polynomial(points)
        polynomial = 0
        for c in temp.coefficients():
```

```
            polynomial = x*polynomial + c
        roots = polynomial.roots(x)
        approx = None
        for root in roots:
            if root.is_real() and
                (root - points[2][0]) * (root - points[1][0]) < 0:
                approx = root
        if approx == None:
            approx = (points[2][0] + points[2][0]) / 2
        points.append((approx, f(approx)))
        yield points[2][0]
a, b = pi/2, pi
```

A.5 Corps finis et théorie des nombres

Exercice 23. On suppose $p < q < r$. Nécessairement $p^3 \leqslant n$, donc la fonction principale s'écrit :

```
def enum_carmichael(n,verbose=True):
   p=3; s=0
   while p^3 <= n:
      s+=enum_carmichael_p(n,p,verbose); p=next_prime(p)
   return s
```

où la fonction `enum_carmichael_p` compte les nombres de Carmichael multiples de p, qui sont de la forme $a + \lambda m$ où $a = p$ et $m = p(p-1)$, puisque n doit être multiple de p et $n-1$ de $p-1$:

```
def enum_carmichael_p(n,p,verbose):
   a=p; m=p*(p-1); q=p; s=0
   while p*q^2 <= n:
      q=next_prime(q); s+=enum_carmichael_pq(n,a,m,p,q,verbose)
   return s
```

La fonction `enum_carmichael_pq` compte les nombres de Carmichael multiples de pq, qui sont de la forme $a + \lambda m$ où $a = p$ et $m = p(p-1)$:

```
def enum_carmichael_pq(n,a,m,p,q,verbose):
   s = 0
   try:
      a=crt(a,0,m,q); m=lcm(m,q)
      a=crt(a,1,m,q-1); m=lcm(m,q-1)
      while a <= p*q^2:
         a += m
      for t in range(a,n+1,m):
         r = t // (p*q)
 if is_prime(r) and t % (r-1) == 1:
```

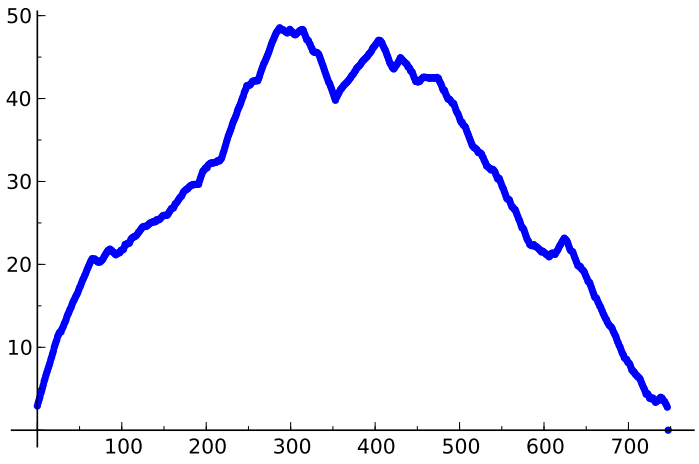
```
            if verbose:
         print p*q*r, factor(p*q*r)
            s += 1
      return s
   except ValueError:
      return 0
```

Avec ces fonctions, on obtient :

```
sage: enum_carmichael(10^4)
3 11 17 561
5 13 17 1105
5 17 29 2465
7 13 19 1729
7 13 31 2821
7 19 67 8911
7 23 41 6601
7
sage: enum_carmichael(10^5,False)
12
sage: enum_carmichael(10^6,False)
23
sage: enum_carmichael(10^7,False)
47
```

Exercice 24. On commence par écrire une fonction `aliq` calculant la suite aliquote partant de s, et s'arrêtant dès qu'on atteint 1 ou un cycle :

```
def aliq(s):
   l=[s]
   while s <> 1:
      s = sigma(s) - s
      if s in l:
         break
      l.append(s)
   return l
l = aliq(840)
len(l), l[:5], l[-5:]
p=points([(i,log(l[i])/log(10)) for i in range(len(l))]); p
```



Exercice 25. Pour la question 1, sans restriction de généralité on peut supposer que l'origine O est sur le cercle — en effet, il y a au moins un point de $\mathbb{Z}^2$ sur le cercle, sinon celui-ci n'est pas optimal. On peut aussi supposer que le centre du cercle est dans le premier quadrant (par rotation du cercle d'un multiple de $\pi/2$ autour de O). On admettra qu'on a également deux points A et B du premier quadrant sur le cercle, et donc le cercle C est circonscrit au triangle O, A, B. La borne $r_k < \sqrt{k/\pi}$ permet de délimiter les points A et B, car leur distance à O est au plus $2\sqrt{k/\pi}$. On peut supposer qu'un des points A et B, par exemple A, est dans le second octant (s'ils sont tous deux dans le premier octant, par symétrie par rapport à la droite $x = y$ on les ramène dans le second octant). On peut aussi supposer que l'angle en A du triangle OAB est aigu (en échangeant A et B au besoin, après une symétrie par rapport à la droite $x = y$ s'ils sont dans des octants différents). L'abscisse de A vérifie donc $x_A < \sqrt{2k/\pi}$, son ordonnée vérifie $x_A \leqslant y_A < \sqrt{4k/\pi - x_A^2}$. Pour le point B, on a $0 \leqslant x_B < 2\sqrt{k/\pi}$, et $0 \leqslant x_A y_B + y_A x_B \leqslant x_A^2 + y_A^2$ (angle aigu en A). Cela donne le code suivant, où la routine `rk_aux` calcule le nombre de points dans le disque de centre $(x_c/d, y_c/d)$, et de rayon $\sqrt{r_2}/d$, où x_c, y_c, d, r_2 sont tous entiers.

```
def rk_aux(xc,yc,d,r2):
   s = 0
   xmin = ceil((xc - sqrt(r2))/d)
   xmax = floor((xc + sqrt(r2))/d)
   for x in range(xmin,xmax+1):
      # (d*y-yc)^2 <= r2 - (d*x-xc)^2
      r3 = r2 - (d*x-xc)^2
      ymin = ceil((yc - sqrt(r3))/d)
      ymax = floor((yc + sqrt(r3))/d)
      s += ymax + 1 - ymin
   return s

def rk(k): # renvoie (r_k^2, xc, yc)
   if k == 2:
      return 1/4, 1/2, 0 # disque de centre (1/2,0) et rayon 1/2
```

```
   dmax = (2*sqrt(k/pi)).numerical_approx() # borne sur diametre
   xamax = (sqrt(2*k/pi)).numerical_approx() # borne sur xa
   sol = (dmax/2)^2, 0, 0, 0
   for xa in range(0,floor(xamax)+1):
      # si xa=0, ya > 0 car A ne peut etre en O
      yamin = max(xa,1)
      for ya in range(yamin,floor(sqrt(dmax^2-xa^2))+1):
         # on veut xb*ya <= xa^2+ya^2
         xbmin = 0
         if xa == 0:
            xbmin = 1 # O, A, B ne doivent pas etre alignes
         xbmax = min(floor(dmax),floor((xa*xa+ya*ya)/ya))
         for xb in range(xbmin,xbmax+1):
            ybmax = floor(sqrt(dmax^2-xb^2))
            # on veut xb*ya+yb*xa <= xa^2+ya^2
            if xa > 0:
               ybmax = min(ybmax,floor((xa*xa+ya*ya-xb*ya)/xa))
            # si xb=0, yb > 0 car B doit etre distinct de O
            ybmin = 0
            if xb == 0:
               ybmin = 1
            for yb in range(ybmin,ybmax+1):
               # calcul du centre du cercle passant par O,A,B
               d = 2*abs(xb*ya - xa*yb)
               if d <> 0:
               ra2 = xa^2+ya^2
                  rb2 = xb^2+yb^2
                  xc = abs(ra2*yb - rb2*ya) # vrai xc est xc/d
                  yc = abs(rb2*xa - ra2*xb) # vrai yc esr yc/d
                  r2 = ra2*rb2*((xa-xb)^2+(ya-yb)^2) # carre de (r*d)
                  m = rk_aux(xc,yc,d,r2)
                  if m>=k and r2/d^2<sol[0]:
                     sol = r2/d^2, xc/d, yc/d
   return sol

sage: for k in range(2,10):
    print k, rk(k)
2 (1/4, 1/2, 0)
3 (1/2, 1/2, 1/2)
4 (1/2, 1/2, 1/2)
5 (1, 0, 1)
6 (5/4, 1/2, 1)
7 (25/16, 3/4, 1)
8 (2, 1, 1)
9 (2, 1, 1)
```

Pour la question 2, une solution est la suivante :

```
def plotrk(k):
   r2, x0, y0 = rk(k); r = numerical_approx(sqrt(r2))
```

```
    var('x,y')
    c = implicit_plot((x-x0)^2+(y-y0)^2-r2,(x,x0-r-1/2,x0+r+1/2),
                                           (y,y0-r-1/2,y0+r+1/2))
    center = points([(x0,y0)],pointsize=50,color='black')
    # on veut (i-x0)^2+(j-y0)^2 <= r2
    # donc |i-x0| <= r et |j-y0| <= r2 - (i-x0)^2
    l = [(i, j) for i in range(ceil(x0-r),floor(x0+r)+1)
                for j in range(ceil(y0-sqrt(r^2-(i-x0)^2)),
                               floor(y0+sqrt(r2-(i-x0)^2))+1)]
    d = points(l,pointsize=100)
    return (c+center+d).show(aspect_ratio=1,axes=True)
```

La question 3 demande un peu de réflexion. Notons $S_{i,j} = \sum_{k=i}^{j} 1/(\pi r_k^2)$. Partant de la borne supérieure (8.2) pour r_k, on obtient $r_k^2 < (k-1)/\pi$, donc $1/(\pi r_k^2) > 1/(k-1)$, et $S_{n,N} > \sum_{k=n}^{N} 1/(k-1) > \int_n^{N+1} \mathrm{d}k/k = \log((N+1)/n)$.

La borne inférieure pour r_k donne $1/(\pi r_k^2) < 1/k + 2/k^{3/2}$ pour $k \geqslant 407$, qui conduit pour $n \geqslant 407$ à

$$S_{2,n-1} + \log(1/n) \leqslant \delta \leqslant S_{2,n-1} + \log(1/(n-1)) + 4/\sqrt{n-1}.$$

```
def bound(n):
    s = sum(1/pi/rk(k)[0] for k in range(2,n+1))
    return float(s+log(1/n)), float(s+log(1/(n-1))+4/sqrt(n-1))
sage: bound(60)
(1.7327473659779615, 2.2703101282176377)
```

On en déduit $1.73 < \delta < 2.28$, soit l'approximatation $\delta \approx 2.00$ avec une erreur bornée par 0.28.

Exercice 26. On reprend ici les mêmes notations que dans l'article de Beauzamy. On pose $s_i = 1 - x_i - \cdots - x_k$ avec $s_{k+1} = 1$. On doit donc avoir $x_1 + \cdots + x_{i-1} \leqslant s_i$, et en particulier $x_2 \leqslant x_1 \leqslant s_2$. Soit

$$C_1 = \int_{x_1=x_2}^{s_2} x_1^{n_1} \mathrm{d}x_1 = \frac{1}{n_1+1}(s_2^{n_1+1} - x_2^{n_1+1}).$$

```
sage: var('x1,x2,s2');
sage: n1=9; C1=integrate(x1^n1,x1,x2,s2); C1
1/10*s2^10 - 1/10*x2^10
```

Ensuite on a $x_3 \leqslant x_2 \leqslant s_3 = s_2 + x_2$, donc en remplaçant s_2 par $s_3 - x_2$ dans C_1, et en intégrant pour x_2 allant de x_3 à $s_3/2$ — car $x_1 + x_2 \leqslant s_3$ et $x_2 \leqslant x_1$ — on obtient :

```
sage: var('x3,s3');
sage: n2=7; C2=integrate(C1.subs(s2=s3-x2)*x2^n2,x2,x3,s3/2); C2
```

```
44923/229417943040*s3^18 - 1/80*s3^10*x3^8 + 1/9*s3^9*x3^9
- 9/20*s3^8*x3^10 + 12/11*s3^7*x3^11 - 7/4*s3^6*x3^12
+ 126/65*s3^5*x3^13 - 3/2*s3^4*x3^14 + 4/5*s3^3*x3^15
- 9/32*s3^2*x3^16 + 1/17*s3*x3^17
```

et ainsi de suite. À chaque itération C_i est un polynôme en x_{i+1} et s_{i+1} à coefficients rationnels, homogène et de degré total $n_1 + \ldots + n_i + i$.

A.6 Polynômes

Exercice 27. Une solution simple consiste à effectuer des divisions euclidiennes successives par les polynômes de Tchebycheff pris par degrés décroissants : si le polynôme p à réécrire sur la base de Tchebycheff est de degré n, on pose $p = c_n T_n + R_{n-1}$ avec $c_n \in \mathbb{Q}$ et $\deg R_{n-1} \leqslant n-1$, puis $R_{n-1} = c_{n-1} T_{n-1} + R_{n-2}$, et ainsi de suite.

Dans le code Sage suivant, plutôt que de renvoyer les coefficients c_n obtenus comme une simple liste, on a choisi de construire une expression symbolique où le polynôme T_n est représenté comme une fonction « inerte » (c'est-à-dire gardée sous forme non évaluée) `T`.

```
T = sage.symbolic.function_factory.function('T', nargs=2)
def to_chebyshev_basis(pol):
    (x,) = pol.variables()
    res = 0
    for n in xrange(pol.degree(), -1, -1):
        quo, pol = pol.quo_rem(chebyshev_T(n, x))
        res += quo * T(n, x)
    return res
```

Testons cette fonction. Pour vérifier les résultats, il suffit de substituer à notre fonction inerte `T` la fonction qui calcule les polynômes de Tchebycheff, et de développer :

```
sage: p = QQ['x'].random_element(degree=6); p
4*x^6 + 4*x^5 + 1/9*x^4 - 2*x^3 + 2/19*x^2 + 1
sage: p_cheb = to_chebyshev_basis(p); p_cheb
1069/456*T(0, x) + T(1, x) + 2713/1368*T(2, x) + 3/4*T(3, x)
+ 55/72*T(4, x) + 1/4*T(5, x) + 1/8*T(6, x)
sage: p_cheb.substitute_function(T, chebyshev_T).expand()
4*x^6 + 4*x^5 + 1/9*x^4 - 2*x^3 + 2/19*x^2 + 1
```

Exercice 28. Une traduction directe de l'algorithme en Sage donne quelque chose comme :

```
def mydiv(num, den, n):
    cc = den.constant_coefficient()
    quo = 0; rem = num
    for k in xrange(n):
```

```
        u = rem[0]/cc
        rem = (rem - u*den) >> 1  # division par x par décalage
        quo += u*x^k
    return quo, rem
```

(On pourra chronométrer cette fonction sur des exemples un peu gros, et essayer de rendre le code plus efficace sans changer l'algorithme.)

Mais le quotient dans la division par les puissances croissantes jusqu'à l'ordre n est simplement le développement en série de la fraction rationnelle u/v, tronqué à l'ordre n. En utilisant la division de séries formelles (voir §9.1.8), on peut donc calculer la division suivant les puissances croissantes comme suit.

```
def mydiv2(num, den, n):
    x = num.parent().gen()
    quo = (num/(den + O(x^n))).polynomial()
    rem = (num - quo*den) >> n
    return quo, rem
```

La ligne `quo = ...` utilise, premièrement, qu'ajouter un terme d'erreur $O(\cdot)$ à un polynôme le convertit automatiquement en série tronquée, et deuxièmement, que la division d'un polynôme par une série se fait par défaut à la précision du diviseur.

Exercice 29. Le polynôme p est un polynôme pair, de degré au plus 40, à coefficients tirés au hasard entre 0 et 9. Il n'est guère surprenant qu'il soit irréductible. On pourrait faire le même raisonnement sur sa dérivée, et en effet, celle-ci est irréductible sur $\mathbb{Q}$ avec forte probabilité. Mais comme p est pair, tous les coefficients de p' sont divisibles par 2, et p' (qui n'est pas constant) ne peut donc pas être irréductible sur $\mathbb{Z}$.

Exercice 30. Tout d'abord, on s'attend à ce que $u_{10^{100}}$ ait de l'ordre de 10^{100} chiffres. Il est donc complètement hors de question de chercher à le calculer entièrement, mais puisqu'on ne s'intéresse qu'aux cinq derniers chiffres, ce n'est pas vraiment un problème : on fera tout le calcul modulo 10^5. La méthode par exponentiation rapide présentée en §3.1.4 demande alors quelques centaines de multiplications de matrices 1000×1000 à coefficients dans $\mathbb{Z}/10^5\mathbb{Z}$. Chacun de ces produits de matrices revient à un milliard de multiplications modulo 10^5, ou un peu moins avec un algorithme rapide. Ce n'est pas complètement inaccessible, mais un essai sur une seule multiplication laisse penser que le calcul avec Sage sans astuce particulière[1] prendrait un à deux jours :

```
sage: m1, m2 = (Mat.random_element() for i in (1,2))
```

[1] Le lecteur intéressé pourra chercher une solution *avec* astuce plus efficace, par exemple en s'appuyant sur les remarques de la section 5.2.10 sur les performances des divers produits matriciels.

```
sage: %time m1*m2
CPU times: user 410.49 s, sys: 0.16 s, total: 410.65 s
Wall time: 410.87 s
```

Il est possible de faire beaucoup mieux du point de vue algorithmique. Notons S l'opérateur de décalage $(a_n)_{n\in\mathbb{N}} \mapsto (a_{n+1})_{n\in\mathbb{N}}$. L'équation satisfaite par $u = (u_n)_{n\in\mathbb{N}}$ se réécrit $P(S)\cdot u = 0$, où $P(x) = x^{1000} - 23x^{729} + 5x^2 - 12x - 7$; et pour tout N (notamment $N = 10^{100}$), le terme u_N est le premier de la suite $S^N \cdot u$. Soit R le reste de la division euclidienne de x^N par P. Comme $P(S)\cdot u = 0$, on a $S^N \cdot u = R(S)\cdot u$. Il suffit donc de calculer l'image de x^N dans $(\mathbb{Z}/10^5\mathbb{Z})\,[x]/\langle P(x)\rangle$. On aboutit au code suivant :

```
sage: Poly.<x> = Integers(100000)[]
sage: P = x^1000 - 23*x^729 + 5*x^2 - 12*x - 7
sage: Quo.<s> = Poly.quo(P)
sage: op = s^(10^100)
sage: add(op[n]*(n+7) for n in range(1000))
7472
```

Les cinq chiffres cherchés sont donc 07472.

Exercice 31.

1. Supposons $a_s u_{n+s} + a_{s-1}u_{n+s-1} + \cdots + a_0 u_n = 0$ pour tout $n \geqslant 0$, et notons $u(z) = \sum_{n=0}^{\infty} u_n z^n$. Soit $Q(z) = a_s + a_{s-1}z + \cdots + a_0 z^s$. Alors

$$S(z) = Q(z)\,u(z) = \sum_{n=0}^{\infty} (a_s u_n + a_{s-1}u_{n-1} + \cdots + a_0 u_{n-s}) z^n,$$

   avec la convention que $u_n = 0$ pour $n < 0$. Le coefficient de z^n dans $S(z)$ est nul pour $n \geqslant s$, donc $S(z)$ est un polynôme, et $u(z) = S(z)/Q(z)$. Le dénominateur $Q(z)$ est le polynôme réciproque du polynôme caractéristique de la récurrence, et le numérateur code les conditions initiales.

2. Les quelques premiers coefficients suffisent pour deviner une récurrence d'ordre 3 que satisfont manifestement les coefficients donnés. Avec `rational_reconstruct`, on obtient une fraction rationnelle qu'il suffit de développer en série pour retrouver tous les coefficients donnés, et des coefficients suivants vraisemblables :

```
sage: p = previous_prime(2^30); ZpZx.<x> = Integers(p)[]
sage: s = ZpZx([1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339])
sage: num, den = s.rational_reconstruct(x^12, 6, 6)
sage: S = ZpZx.completion(x)
sage: map(signed_lift, S(num)/S(den))
[1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339, 3056, 371,
14602, -4257, 72268, -50489, 369854, -396981]
```

(La fonction `signed_lift` est celle définie dans le texte du chapitre. Les 20 premiers coefficients de la suite sont largement inférieurs à 2^{29}, de sorte qu'on peut se permettre de dérouler la récurrence modulo 2^{30} puis de remonter le résultat dans $\mathbb{Z}$ plutôt que le contraire.)

Avec `berlekamp_massey`, le résultat est le polynôme caractéristique de la récurrence, directement à coefficients dans $\mathbb{Z}$:

```
sage: berlekamp_massey([1, 1, 2, 3, 8, 11, 34, 39, 148, 127])
x^3 - 5*x + 2
```

On vérifie que tous les coefficients donnés satisfont $u_{n+3} = 5u_{n+1} - 2u_n$, et l'on devine à partir de là les coefficients manquants $72268 = 5 \cdot 14602 - 2 \cdot 371$, $-50489 = 5 \cdot (-4257) - 2 \cdot 14602$, et ainsi de suite.

Exercice 32. On commence par calculer un polynôme de degré 3 qui satisfait la condition d'interpolation donnée, ce qui fournit une solution avec $\deg p = 4$:

```
sage: R.<x> = GF(17)[]
sage: s = R(QQ['x'].lagrange_polynomial([(0,-1),(1,0),(2,7),(3,5)]))
sage: s
6*x^3 + 2*x^2 + 10*x + 16
sage: [s(i) for i in range(4)]
[16, 0, 7, 5]
```

On s'est ainsi ramené au problème de reconstruction rationnelle

$$p/q \equiv s \mod x(x-1)(x-2)(x-3).$$

Comme s n'est pas inversible modulo $x(x-1)(x-2)(x-3)$ (car $s(1) = 0$), il n'y a pas de solution avec p constant. Avec $\deg p = 1$, on trouve :

```
sage: s.rational_reconstruct(mul(x-i for i in range(4)), 1, 2)
(15*x + 2, x^2 + 11*x + 15)
```

Exercice 33. Le raisonnement est le même que dans l'exemple du texte : on réécrit l'équation $\tan x = \int_0^x (1 + \tan^2 t)\,\mathrm{d}t$, et l'on cherche un point fixe en partant de la condition initiale $\tan(0) = 0$.

```
sage: S.<x> = PowerSeriesRing(QQ)
sage: t = S(0)
sage: for i in range(8):
....:     # le O(x^15) évite que l'ordre de troncature ne grandisse
....:     t = (1+t^2).integral() + O(x^15)
sage: t
x + 1/3*x^3 + 2/15*x^5 + 17/315*x^7 + 62/2835*x^9 + 1382/155925*x^11
+ 21844/6081075*x^13 + O(x^15)
```

A.7 Algèbre linéaire

Exercice 34. (*Polynôme minimal de vecteurs*)

1. φ_A est un polynôme annulateur de tous les vecteurs e_i de la base. Il est donc un commun multiple des φ_{A,e_i}. Soit ψ le ppcm des φ_{A,e_i}. Donc $\psi|\varphi_A$. Par ailleurs, $\psi(A) = \begin{bmatrix}\psi(A)e_1 & \dots & \psi(A)e_n\end{bmatrix} = 0$ est annulateur de la matrice A. D'où $\varphi_A|\psi$. Comme ces polynômes sont unitaires, ils sont égaux.
2. Dans ce cas, tous les φ_{A,e_i} sont sous la forme χ^{ℓ_i}, où χ est un polynôme irréductible. D'après la question précédente, φ_A coïncide donc avec celui des χ^{ℓ_i} ayant la puissance ℓ_i maximale.
3. Soit φ un polynôme annulateur de $e = e_i + e_j$ et $\varphi_1 = \varphi_{A,e_i}, \varphi_2 = \varphi_{A,e_j}$. On a $\varphi_2(A)\varphi(A)e_i = \varphi_2(A)\varphi(A)e - \varphi(A)\varphi_2(A)e_j = 0$. Donc $\varphi_2\varphi$ est annulateur de e_i et est donc divisible par φ_1. Or φ_1 et φ_2 étant premiers entre eux, on a $\varphi_1|\varphi$. De la même façon on montre que $\varphi_2|\varphi$, donc φ est un multiple de $\varphi_1\varphi_2$. Or $\varphi_1\varphi_2$ est annulateur de e, donc $\varphi = \varphi_1\varphi_2$.
4. P_1 et P_2 étant premiers entre eux, il existe deux polynômes α et β tels que $1 = \alpha P_1 + \beta P_2$. Ainsi pour tout x, on a $x = \alpha(A)P_1(A)x + \beta(A)P_2(A)x = x_2 + x_1$, où $x_1 = \beta(A)P_2(A)x$ et $x_2 = \alpha(A)P_a(A)x$. P_1 est annulateur de x_1. Si pour tout x, $x_1 = 0$, alors βP_1 est annulateur de A et est donc un multiple de P_1P_2, d'où $1 = P_1(\alpha + \gamma P_2)$, qui implique $\deg P_1 = 0$. Il existe donc un x_1 non nul tel que P_1 soit un polynôme annulateur de x_1. Montrons que P_1 est minimal : soit $\tilde{P}_1$ un polynôme annulateur de x_1. Alors $\tilde{P}_1(A)P_2(A)x = P_2(A)\tilde{P}_1(A)x_1 + \tilde{P}_1(A)P_2x_2 = 0$, donc $\tilde{P}_1P_2$ est un multiple de $\varphi_A = P_1P_2$. Ainsi $P_1|\tilde{P}_1$ et P_1 est donc le polynôme minimal de x_1. Le raisonnement est identique pour x_2.
5. Pour chaque facteur $\varphi_i^{m_i}$, il existe un vecteur x_i dont $\varphi_i^{m_i}$ est le polynôme minimal et le vecteur $x_1 + \dots + x_k$ a φ_A pour polynôme minimal.
6.
```
sage: A=matrix(GF(7),5,
....:   [0,0,3,0,0,1,0,6,0,0,0,1,5,0,0,0,0,0,0,5,0,0,0,1,5]); A
[0 0 3 0 0]
[1 0 6 0 0]
[0 1 5 0 0]
[0 0 0 0 5]
[0 0 0 1 5]
sage: P=A.minpoly();P
x^5 + 4*x^4 + 3*x^2 + 3*x + 1
sage: P.factor()
(x^2 + 2*x + 2) * (x^3 + 2*x^2 + x + 4)
```
   Le polynôme minimal de la matrice A est de degré maximal.
```
sage: e1=identity_matrix(GF(7),5)[0];
sage: e4=identity_matrix(GF(7),5)[3];
sage: A.transpose().maxspin(e1)
```

```
[(1, 0, 0, 0, 0), (0, 1, 0, 0, 0), (0, 0, 1, 0, 0)]
sage: A.transpose().maxspin(e4)
[(0, 0, 0, 1, 0), (0, 0, 0, 0, 1)]
sage: A.transpose().maxspin(e1+e4)
[(1, 0, 0, 1, 0), (0, 1, 0, 0, 1), (0, 0, 1, 5, 5),
 (3, 6, 5, 4, 2), (1, 5, 3, 3, 0)]
```

La fonction `maxspin` itère un vecteur *à gauche*. On l'appliquera donc sur la transposée de la matrice afin d'obtenir la liste des itérés de Krylov linéairement indépendants à partir des vecteurs e_1 et e_4. On notera que la forme particulière de la matrice fait que les vecteurs e_1 et e_4 engendreront comme itérés d'autres vecteurs de la base canonique. Cette forme est appelée la forme normale de Frobenius, et sera étudiée à la fin de la partie 10.2.3. Elle décrit de quelle manière la matrice décompose l'espace en sous-espaces cycliques invariants qui sont engendrés par des vecteurs de la base canonique.

Exercice 35. (*Test si deux matrices sont semblables*)

```
def Semblables(A,B):
    F1,U1 = A.frobenius(2)
    F2,U2 = B.frobenius(2)
    if F1==F2:
        return True, ~U2*U1
    else:
        return False,  A.matrix_space().zero_matrix()
```

A.8 Équations différentielles

Exercice 36. (*Équations différentielles à variables séparables*)

1. Utilisons la même méthode que dans la section 11.2.4 :

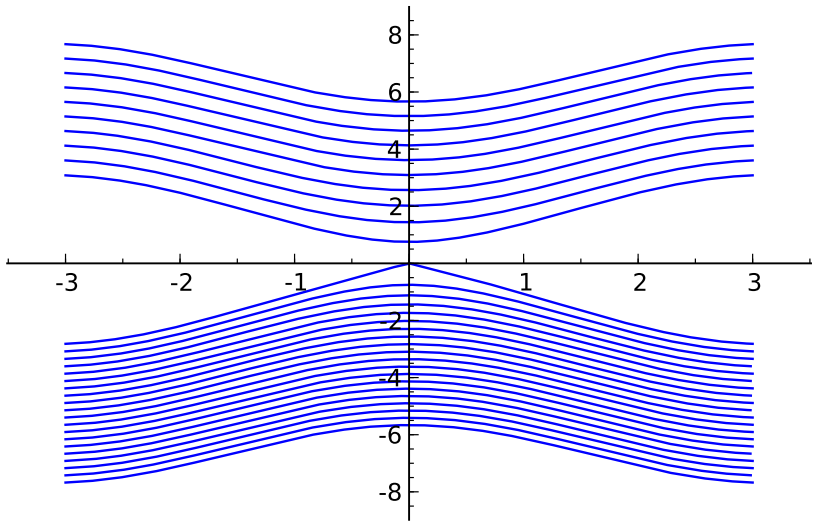
```
sage: x=var('x')
sage: y=function('y',x)
sage: ed=(desolve(y*diff(y(x),x)/sqrt(1+y^2)==sin(x),y))
sage: ed

sqrt(y(x)^2 + 1) == c - cos(x)

sage: c=ed.variables()[0]
sage: assume(c-cos(x)>0)
sage: sol=solve(ed,y(x))
sage: sol

[y(x) == -sqrt(c^2 - 2*c*cos(x) + cos(x)^2 - 1),
 y(x) == sqrt(c^2 - 2*c*cos(x) + cos(x)^2 - 1)]
```

```
sage: P=Graphics()
sage: for j in [0,1]:
....: for k in range(0,20,2):
....: P+=plot(sol[j].subs_expr(c==2+0.25*k).rhs(),
              x,-3,3)
sage: P
```



2. Même méthode :

```
sage: ed=desolve(diff(y,x)==sin(x)/cos(y),y,
                 show_method=True)
sage: ed
[sin(y(x)) == c - cos(x), 'separable']
sage: solve(ed[0],y(x))
[y(x) == -arcsin(-c + cos(x))]
```

Exercice 37. (*Équations homogènes*)

On veut résoudre $xyy' = x^2 + y^2$. On vérifie que cette équation est bien homogène puis on essaie de la résoudre.

```
sage: x=var('x')
sage: y=function('y',x)
sage: id(x)=x
sage: u=function('u',x)
sage: d=diff(u*id,x)
sage: DE=(x*y*d==x**2+y**2).subs_expr(y==u*id)
sage: assume(x>0)
sage: equ=desolve(DE,u)
sage: assume(equ.variables()[0]+log(x)>0)
sage: solu=solve(equ,u(x))
```

```
[u(x) == -sqrt(c + log(x))*sqrt(2),
 u(x) == sqrt(c + log(x))*sqrt(2)]
sage: Y=[x*solu[0].rhs(),x*solu[1].rhs()]
sage: Y[0]
-sqrt(c + log(x))*sqrt(2)*x
```

B
Bibliographie

[AP98] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[Cia82] Philippe G. Ciarlet. *Introduction à l'analyse numérique matricielle et à l'optimisation*. Collection Mathématiques Appliquées pour la Maîtrise. Masson, Paris, 1982.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics 138. Springer-Verlag, 1993. 534 pages.

[CP00] Richard Crandall and Carl Pomerance. *Prime Numbers : A Computational Perspective*. Springer-Verlag, 2000.

[DGSZ95] Philippe Dumas, Claude Gomez, Bruno Salvy, and Paul Zimmermann. *Calcul formel : mode d'emploi*. Masson, Paris, 1995.

[Gan90] Félix Rudimovich Gantmacher. *Théorie des Matrices*. Éditions Jacques Gabay, 1990.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[HT04] Florent Hivert and Nicolas M. Thiéry. MuPAD-Combinat, an open-source package for research in algebraic combinatorics. *Sém. Lothar. Combin.*, 51 :Art. B51z, 70 pp. (electronic), 2004. `http ://mupad-combinat.sf.net/`.

[LA04] Henri Lombardi and Journaïdi Abdeljaoued. *Méthodes matricielles - Introduction à la complexité algébrique*. Berlin, Heidelberg, New-York : Springer, 2004.

[LT93] P. Lascaux and R. Théodor. *Analyse numérique matricielle appliquée à l'art de l'ingénieur. Tome 1*. Masson, Paris, second edition, 1993.

[LT94] P. Lascaux and R. Théodor. *Analyse numérique matricielle appliquée à l'art de l'ingénieur. Tome 2*. Masson, Paris, second edition, 1994.

[Mor05] Masatake Mori. Discovery of the Double Exponential Transformation and Its Developments. *Publ. RIMS*, 41 :897–935, 2005.

[Sch91] Michelle Schatzman. *Analyse numérique*. InterEditions, Paris, 1991.

[TMF00] Gérald Tenenbaum and Michel Mendès France. *Les nombres premiers*. Que sais-je ? P.U.F., Paris, 2000.

[TSM05] Ken'ichiro Tanaka, Masaaki Sugihara, and Kazuo Murota. Numerical indefinite integration by double exponential sinc method. *Mathematics of Computation*, 74(250) :655–679, 2005.

[Vie07] Xavier Viennot. Leonhard Euler, père de la combinatoire contemporaine, Mai 2007. exposé à la journée Leonhard Euler, mathématicien universel, tricentenaire de sa naissance, IHES, Bure-sur-Yvette.

[vzGG03] J. von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.

[Zei96] D. Zeilberger. Proof of the alternating sign matrix conjecture. *Electron. J. Combin*, 3(2), 1996.