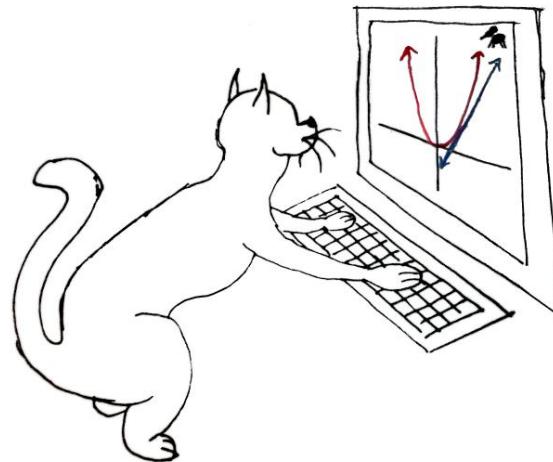


## Explorations mathématiques avec ordinateur

### Technologie de l'algèbre



John Harris,

Karen Kohl et

John Perry,

tous employés à l'Université du sud du Mississippi (du moins  
jusqu'à ce que le bureau du prévôt lise cette débâcle)

Classification des matières mathématiques de 2000. 97U30, 97N80

Copyright © 2016–2021 John Harris, Karen Kohl et John Perry  
Brouillon  $2 \cdot 3^2$ , Juin  $(x + 1)(x + 5)$ , où  $x$  est le sens de la vie, de l'univers et de tout  
[Licence CC-BY-SA](#)

# Contenu

Remerciements	7
Préface	8
Pourquoi avez-vous écrit ce texte ?	8
Non, vraiment, pourquoi as-tu écrit ce texte ?	8
Quelle valeur pédagogique ce texte offre-t-il ?	8
Pourquoi Sage ?	8
N'y a-t-il pas déjà de bonnes références sur Sage ?	9
Comment utilisez-vous généralement ce texte ?	9
Un dernier mot ?	9
Partie 1. Leçons	11
Chapitre 1. Contexte S'agit-il	12
simplement d'un autre manuel de programmation ?	12
C'est quoi ce truc Sage dont tu parles sans cesse ?	14
Comment démarrer avec Sage ?	18
Feuille de calcul ou ligne de commande ?	19
Obtenir de l'aide	21
Exercices	23
Chapitre 2. Calculs de base Votre arithmétique de base	25
Constantes, variables et indéterminées Expressions et commandes pour les manipuler Votre calcul de base Structures mathématiques dans	29
Sage Exercices	31
Chapitre 3. Jolies (et moins jolies) images Objets 2D Tracés 2D	36
Animation	43
Tracés implicites	48
Exercices	52
Chapitre 4. Définir vos propres procédures Définir une procédure Arguments Mettre fin à la communication dysfonctionnelle	65
communication dysfonctionnelle	69
Exercices	71
Exercices	75
Chapitre 5. Générer des fichiers de données	79
Génération de fichiers de données	80
Exercices	83
Chapitre 6. Utiliser les fonctions de Sage	89

## CONTENU

4

Pseudocode	92
Script	94
Fiches de travail interactives	97
Exercices	101
Chapitre 5. Se répéter définitivement avec des collections Comment faire en sorte qu'un ordinateur répète un nombre fixe de fois ?	106
Comment ça marche ? ou, une introduction aux collections Répéter sur une collection Compréhensions :	109
répéter dans une collection Animation à nouveau	118
Exercices	124
	126
	126
Chapitre 6. Résolution d'équations Les bases Une	133
équation ou une inégalité, pour une indéterminée Erreurs ou surprises qui surviennent lors de la résolution Solutions approximatives d'une équation Systèmes	133
d'équations Matrices	138
Exercices	140
	141
	150
Chapitre 7. Prise de décision La méthode de la bisection	153
Logique	156
booléenne Briser	159
une boucle	160
Exceptions Exercices	170
Chapitre 8. Se répéter indéfiniment Mettre en œuvre une boucle indéfinie Qu'est-ce qui pourrait mal se passer ?	175
Exercices sur la division des entiers gaussiens	176
	180
	182
	187
Chapitre 9. Se répéter de manière inductive Récursivité Alternatives	193
à la récursivité Exercices	193
	197
	207
Chapitre 10. Rendre vos images tridimensionnelles Objets 3D Tracés 3D	213
de base Outils	213
avancés pour les tracés 3D	215
Ascension d'une colline Exercices	222
	225
	229
Chapitre 11. Techniques avancées Fabriquer ses propres objets	232
	232

## CONTENU

5

Cython	241
Exercices	247
Chapitre 12. Commandes de base	251
LATEX utiles	251
Délimiteurs	251
Matrices	251
Partie 2. Travaux pratiques	254
Prérequis pour chaque laboratoire	255
Mathématiques générales	257
Polynômes irréductibles	258
Différents types de parcelles	260
Cauchy, le cercle et la parabole croisée	262
Calcul et équations différentielles	263
Un quotient de différence important	264
Illustrer le calcul	265
La règle de Simpson	267
La méthode Runge-Kutta	268
Coefficients de Maclaurin	269
série p	270
Sur une tangente	271
Traversée en angle	272
Maxima et Minima en 3D	273
Algèbre linéaire	274
Propriétés algébriques et géométriques des systèmes linéaires	275
Matrices de transformation	277
Visualisation des valeurs propres et des vecteurs propres	278
Moyenne des moindres carrés	279
Méthode Bareiss	280
Méthode Dodgson	281
Mathématiques discrètes	282
Fonctions un à un	283
Le jeu des ensembles	284
Le nombre de façons de sélectionner m éléments dans un ensemble de n	285
Algèbre et théorie des nombres	286
Propriétés des anneaux finis	287
L'horloge chinoise du reste	288
La géométrie des racines radicales	289
Séquences de Lucas	291
Introduction à la théorie des groupes	292
Théorie du codage et cryptographie	296
Fractions continues	298

CONTENU	6
---------	---

Bibliographie	299
---------------	-----

Indice	300
--------	-----

## Remerciements

Les auteurs tiennent à remercier :

- Le Bureau du prévôt de l'Université du sud du Mississippi, qui a soutenu en 2016 la création de cette œuvre avec une subvention d'été pour l'amélioration de l'instruction.
- William Stein et divers développeurs et utilisateurs de Sage pour leur soutien moral et leurs contributions occasionnelles soutien financier pour participer aux ateliers Sage Days.
- Tous les développeurs qui ont contribué au projet Sage, que ce soit directement ou indirectement.
- Amber Desrosiers et Candice Bardwell Mitchell ont souffert pendant la première ébauche de ce texte, et nous avons trouvé plus d'erreurs typographiques que nous ne voulons l'admettre.

Valerio de Angelis a aimablement signalé un grand nombre d'erreurs typographiques et suggéré une amélioration à une blague sur les boucles infinies. Il a également contribué à quelques travaux pratiques. dans l'Encyclopædia Laboratorica.

- Deux d'entre nous ont des conjoints et des enfants qui méritent plus des excuses que des remerciements, deux d'entre nous avoir des chats qui méritent tout ce que les chats méritent (tout ?), et l'un de nous a un un lapin et une tortue, qui méritent tous deux plus d'attention.

En outre:

- L'image de Glenda, le lapin du Plan 9 à la p. 209 a été téléchargée depuis [le Plan des Bell Labs 9 site Web](#) et est utilisé selon les termes donnés.
- L'image de l'île aux lapins à la p. 195 a été prise par [Kim Bui](#) et est utilisé sous une licence spécialement accordée ([CC BY-SA 2.0](#)).

## Préface

Pourquoi avez-vous écrit ce texte ?

L'objectif principal de ce texte est de justifier un salaire d'été partiel qui nous a été accordé par notre employeur. Prévôt, qui sans doute ne commettra plus jamais d'erreur de cette façon.

Non, vraiment, pourquoi as-tu écrit ce texte ?

Notre établissement propose aux étudiants un cours de résolution de problèmes technologiques intitulé « Calcul mathématique ». Nous ne trouvons pas de texte qui corresponde à son caractère unique. Nous croyons en ce cours et pensons que c'est une bonne idée ; de nombreux étudiants qui le suivent finissent par approuver.

À l'origine, le cours utilisait un manuel basé sur un autre système de calcul formel, mais pour les raisons énumérées ci-dessous, nous avons opté pour Sage. L'interface de Sage repose sur Python ; nous avons donc utilisé un excellent ouvrage de programmation Python pendant un certain temps, mais un ouvrage de programmation Python ne vous mènera pas loin avec Sage.

D'où ce livre. Avec un peu de chance, le texte qui en résultera poussera comme un champignon, ici et là, invulnérable à l'éradication, jusqu'à dominer le paysage de l'enseignement des mathématiques.

Quelle valeur pédagogique ce texte offre-t-il ?

Nous espérons que ce livre sera utile à toute personne passant des mathématiques du lycée, qui incluent le calcul différentiel et intégral, aux mathématiques universitaires, qui incluent le calcul intermédiaire et bien d'autres matières. Nombre de nos élèves éprouvent des difficultés à passer des mathématiques essentiellement concrètes et computationnelles de leur jeunesse aux mathématiques de plus en plus abstraites et théoriques qu'ils rencontrent en mathématiques supérieures.

La technologie est devenue un aspect indispensable de la plupart des programmes d'enseignement des mathématiques. La puissance de calcul des téléphones portables actuels est bien supérieure à celle des ordinateurs personnels de l'enfance des auteurs, et la plupart des familles n'avaient même pas les moyens d'en posséder un.

Notre monde est magique, et pourtant les élèves sont souvent réticents à y jouer. S'ils ne savent pas résoudre un problème immédiatement, ils pensent immédiatement que quelque chose ne va pas. Nous pouvons leur enseigner les groupes (par exemple) et leur donner des exemples, mais les élèves sont souvent trop réticents à explorer ces exemples par eux-mêmes.

Pourquoi Sage ?

Nous préférons Sage pour plusieurs raisons :

- Sage est « gratuit comme la bière ». Les auteurs travaillent dans une université située dans l'un des États les plus modestes du pays sur le plan matériel ; tandis que de nombreuses éditions étudiantes de systèmes de calcul formel sont

---

1Cet accord n'intervient souvent qu'après avoir vécu un stage ou un premier emploi, mais cela compte !

Bien que leur prix soit sans doute abordable, ils constituent néanmoins un ajout non négligeable au coût élevé que supportent déjà nos étudiants. • Sage est « libre comme la parole ». Les enseignants peuvent montrer des extraits de code aux étudiants talentueux et les encourager à s'impliquer. Il existe sans aucun doute de nombreux projets de recherche de premier cycle intéressants qui pourraient donner lieu à des contributions au code source de Sage. Les enseignants talentueux peuvent contribuer au code de manière à améliorer l'utilisation de Sage dans l'enseignement. D'ailleurs, deux d'entre nous ont démontré que même des enseignants sans talent peuvent contribuer au code de Sage. N'hésitez pas : la communauté est solidaire ! • L'interface de Sage repose sur [Python](#). Un langage de programmation standard très recherché par les employeurs. Enseigner Sage aux étudiants implique également de leur enseigner un peu de Python, ce qui améliore leurs perspectives d'emploi.

#### N'y a-t-il pas déjà de bonnes références sur Sage ?

Oui, et c'est là tout l'intérêt ; ce sont des références pour les mathématiciens et les étudiants pour apprendre Sage. Ce texte vise à aider les élèves à apprendre les mathématiques via Sage. Il se situe ainsi dans une niche différente, qui, nous l'espérons, se révélera non seulement profitable, mais aussi utile, tant à lire qu'à éditer.

#### Comment utilisez-vous généralement ce texte ?

Aucune section de laboratoire n'est rattachée au cours dans notre établissement, nous avons donc tendance à proposer des journées de laboratoire en classe ; à procéder dans un semblant d'ordre fourni ici ; et à attribuer des questions de manuel, des laboratoires et essais.

Nous utilisons différents travaux pratiques, selon l'enseignant et l'année. Nous les adaptons également occasionnellement. Il est conseillé de les adapter à nos goûts, à une époque où un nombre non négligeable d'étudiants ont appris que sous-traiter la production à un moteur de recherche en ligne permet d'obtenir des solutions préfabriquées plus rapidement que leurs propres moteurs de calcul. Même sans adaptation, le grand nombre de travaux pratiques permet de varier le cours en fonction des intérêts de recherche et des préférences pédagogiques.

Si vous nourrissez une hostilité particulière envers vos élèves, n'hésitez pas à contacter les auteurs. Nous pouvons indiquer quels laboratoires se sont avérés particulièrement difficiles dans la pratique.

#### Un dernier mot ?

Après cinq années de développement, ce texte et sa source ont été publiés en 2021 sur une plateforme publique de diffusion.

Concernant les errata. La tradition mathématique raconte qu'un professeur s'était un jour donné pour mission d'écrire un manuel exempt d'erreurs. Un critique du texte ainsi rédigé fit remarquer sèchement que l'auteur s'en était plutôt bien sorti : la première erreur apparaît à la page 9.

Nos objectifs sont plus modestes : nous promettons seulement que les erreurs que le lecteur trouvera iront de la simple erreur typographique à la plus pure erreur mensongère.

- En cas d'erreurs typographiques, nous demandons au lecteur de contacter les responsables de la maintenance. Si l'un des enregistrements de publication d'un auteur est indivisible, les corrections typographiques comporteront des erreurs typographiques. Attention, lecteur.

---

<sup>2</sup>Arrête de rire.

3Vraiment. Arrête de rire.

- Les erreurs du mensonge ont le mérite de paraître meilleures que la vérité, c'est pourquoi, dans l'esprit du temps, nous avons décidé de les inclure.<sup>4</sup> Nous laissons au lecteur le soin de trier le bon grain de l'ivraie, même si nous assurons au lecteur assidu qu'il y a plus de bon grain que d'ivraie.<sup>5</sup>

Le lecteur intéressé par le développement historique de ce texte peut souhaiter visiter

[www.math.usm.edu/dont\\_panic/](http://www.math.usm.edu/dont_panic/)

Existe-t-il une version papier ? Non. Il y en avait autrefois, mais les manuels en couleur sont chers et on perd l'expérience des animations intégrées au texte. Si vous souhaitez une version papier, n'hésitez pas à l'imprimer, de préférence avec une imprimante couleur et du papier de haute qualité.

---

<sup>4</sup>Au moins un mensonge flagrant est réfuté comme tel, et la raison de ce mensonge est expliquée.

<sup>5</sup>À moins que cette affirmation ne soit elle aussi mensongère.

## Partie 1

### Leçons

## CHAPITRE 1

# Arrière-plan

Une chose dont ils n'avaient jamais eu besoin auparavant est devenue une nécessité. (Narrateur, [11])

Pour explorer le monde des mathématiques, il ne suffit pas de lire ; il faut s'y intéresser. Certains ont suffisamment confiance en eux et/ou aptitudes pour s'y mettre seuls ; d'autres ont eu la chance d'être bien élevés et, dès leur plus jeune âge, sont habitués à explorer le monde des mathématiques.

La plupart sont moins chanceux et, même s'ils trouvent le monde des mathématiques fascinant, ils peinent à se frayer un chemin en terrain inconnu. Ce texte a pour but de vous encourager à explorer le monde des mathématiques supérieures à l'aide d'un système de calcul formel, basé sur Sage. Nous vous encouragerons à expérimenter des problèmes et à formuler des conjectures sur leurs solutions. Parfois, nous vous inciterons à utiliser ces expérimentations pour formuler une explication de la véracité de la conjecture.

Mais pour utiliser efficacement les ordinateurs, vous devez apprendre à programmer.

Est-ce juste un autre manuel de programmation ?

Non.

Pouvez-vous être plus précis ? Oui.

[Grognon.] Veuillez entrer dans les détails. Ce texte traite des mathématiques, qui elles-mêmes traitent de la résolution de problèmes. C'est suffisamment important pour être souligné, afin que même les lecteurs superficiels le remarquent.

Les mathématiques sont un outil pour résoudre des problèmes.

En particulier, nous souhaitons vous présenter des idées et des techniques de mathématiques supérieures à travers des problèmes qui sont sans doute mieux abordés avec un ordinateur, car • ils sont longs ; • ils sont répétitifs ou

fastidieux ; et/ou • ils nécessitent une expérimentation.

Les ordinateurs nécessitent des instructions, et un groupe d'instructions est appelé un [programme](#).

<sup>1</sup> Nous étudions donc

la programmation, mais en référence à un objectif précis : la résolution de problèmes mathématiques. Cela distingue ce manuel des manuels de « programmation », qui étudient la programmation en référence à un objectif différent : la résolution de problèmes informatiques. Là encore, cette distinction est suffisamment importante pour être soulignée, même les lecteurs les plus superficiels la remarqueront.

Nous étudions la programmation afin de résoudre des problèmes mathématiques.

---

Si vous lisez ce document électronique, vous remarquerez parfois des textes d'une couleur différente. En cliquant dessus, vous trouverez des liens vers des informations complémentaires, dont une très faible part est due aux auteurs de ce texte et qui, de ce fait, sont probablement beaucoup plus fiables et utiles. Nous espérons que vous suivrez ces liens et que vous vous familiariserez avec ces informations – c'est pourquoi nous les avons inclus ! – mais ce n'est pas strictement nécessaire .

Pourquoi programmer ? Si je voulais écrire des programmes, je me spécialiserais en informatique. Pour commencer, certains problèmes sont trop complexes pour être résolus à la main, il faut donc utiliser un ordinateur. L'objectif de ce texte est de vous initier à cette voie sans pour autant vous transformer en un spécialiste en informatique. Il vous présentera non seulement quelques outils d'un système de calcul formel qui vous aident à calculer, mais vous encouragera également à progresser sur la voie qui vous mènera à devenir un pionnier des mathématiques. Pour résoudre les problèmes que nous présentons, vous devrez vous arrêter à un endroit précis, près de chez vous, et vous familiariser un peu plus avec lui que vous ne le feriez sans nos encouragements.

Dans certains cas, nous vous ferons même revenir sur des sujets déjà abordés et vous demanderons de réexaminer un sujet avec un peu plus d'attention. En bref, et en le répétant, ce texte propose d'explorer le monde des mathématiques supérieures, avec l'aide d'un ordinateur.

D'un côté, les ordinateurs ne comprennent pas le langage humain. Les humains sont intuitifs et poétiques, recourant à un langage figuratif et abstrait. Les ordinateurs ne comprennent rien de tout cela ; ils ne comprennent qu'une seule chose : Allumer et éteindre. Tout ce que fait votre téléphone portable, ordinateur portable ou de bureau nécessite un agencement astucieux d'interrupteurs, commandés par des êtres humains parfaitement formés à diriger le courant électrique au bon endroit et au bon moment.

D'un autre côté, la plupart des humains ne comprennent pas le langage informatique, qui signifie « marche » et « arrêt ». Réduire chaque problème à ces deux symboles s'est avéré extrêmement efficace, mais c'est inconfortable pour les humains (ne serait-ce que parce que c'est tellement fastidieux !).

Apprendre à programmer vous permet de contrôler les circuits de l'ordinateur et de travailler à un niveau beaucoup plus confortable. Même les experts travaillent généralement avec des interfaces plus abstraites, elles-mêmes converties en signaux marche/arrêt en plusieurs étapes. Apprendre à programmer vous permet également d'approfondir votre compréhension de la technologie informatique et d'apprécier l'ampleur du travail nécessaire à la construction de ce monde magique où vous pouvez parler dans une petite boîte que vous tenez dans votre main et être entendu à l'autre bout du monde.

Quels types de langages de programmation existe-t-il ? Sans entrer dans les détails, il existe aujourd'hui trois types de langages de programmation :

- Un [langage de programmation interprété](#), L'ordinateur lit un fichier

contenant des commandes dans le langage. Il traduit chaque symbole et exécute une séquence de commandes basée sur ce symbole. (Ici, « symbole » peut inclure aussi bien des mots que des nombres et des caractères abstraits.) Il passe ensuite au symbole suivant, oubliant éventuellement sa traduction du précédent. Exemples de langages interprétés : [BASIC](#), [Python](#), et les langages « shell » des invites de ligne de commande.

- Dans un [langage de programmation compilé](#), L'ordinateur lit un fichier contenant des commandes dans le langage. Il traduit chaque symbole, mais au lieu d'exécuter des commandes, il enregistre la traduction en signaux d'activation et de désactivation et les stocke dans un nouveau fichier, généralement appelé exécutable. (Nous écrivons « habituellement » car il produit parfois une bibliothèque à la demande du programmeur.) Parmi les langages compilés, on trouve [Fortran](#), [C/C++](#), et [c'est parti](#).

Avant de décrire le troisième type de langage, mentionnons quelques avantages et inconvénients de chaque type. Les langages interprétés sont par nature généralement beaucoup plus lents que les langages compilés, car l'ordinateur doit retraduire chaque symbole, quel que soit le nombre de fois qu'il l'a traduit auparavant. (C'est une simplification excessive, mais ce n'est pas si éloigné de la vérité.) En revanche, les langages interprétés sont généralement beaucoup plus interactifs et flexibles que les langages compilés, à tel point que de nombreux utilisateurs n'écrivent jamais de programme pour eux, se contentant d'exécuter une commande à la fois.

De même, la nature des langages compilés implique que les signaux d'activation et de désactivation précis sont liés à une architecture particulière, ce qui explique en partie pourquoi les programmes compilés pour Windows ne fonctionnent pas sur Macintosh ou Linux. Les développeurs C++ doivent recompiler chaque programme pour une architecture différente, ce qui s'avère très difficile pour de nombreux programmes, surtout s'ils s'appuient fortement sur l'interface graphique spécifique de Windows. Par conséquent, les langages interprétés sont beaucoup plus portables, ce qui signifie qu'il suffit de les copier sur une autre machine. Les programmes Python sont particulièrement réputés pour leur portabilité. On peut affirmer que le succès de Microsoft Corporation est principalement dû à son brillant et omniprésent interpréteur BASIC pour ordinateurs personnels de la fin des années 70 et du début des années 80 ; il perdure aujourd'hui sous le nom de Visual BASIC. • [Langages de bytecode](#) cherchent à combler le fossé entre les deux.

Dans ce cas, l'ordinateur lit un fichier contenant des commandes dans le langage et traduit chaque symbole, mais pas en signaux d'activation et de désactivation propres à son architecture. Il les traduit plutôt en signaux conçus pour un ordinateur abstrait appelé machine virtuelle, puis les stocke dans un exécutable ou une bibliothèque qui ne fonctionnera que sur les ordinateurs contenant des programmes qui comprennent les signaux de la machine virtuelle. Parmi les langages de bytecode les plus connus, on trouve [Pascal](#),

<sup>2</sup> Java, et le [Common Language Runtime](#) des langages .NET .

<sup>3</sup>

Comme les exécutables et les bibliothèques ne sont pas intégrés aux signaux de l'ordinateur, les langages de bytecode constituent techniquement un type particulier de langage interprété. Leur dépendance à une machine virtuelle signifie qu'ils sont théoriquement plus lents que les langages compilés. En pratique, la pénalité est généralement minime, car les signaux de la machine virtuelle sont conçus pour être traduits très facilement en signaux d'un ordinateur réel. Les techniques modernes les rendent si efficaces que certains langages de bytecode surpassent souvent de nombreux langages compilés. Leur dépendance à la machine virtuelle abstraite confère aux langages de bytecode une grande portabilité ; nous avons écrit plus haut que les exécutables ne fonctionnent « que » sur les ordinateurs équipés de programmes qui comprennent les signaux de la machine virtuelle, mais cela signifie également qu'ils s'exécutent sur « tout » ordinateur équipé d'un programme qui les comprend. La popularité de Java – il semblait autrefois impossible de trouver une page web sans applet Java – était due à sa philosophie « Écrire une fois, exécuter n'importe où », qui dépendait des capacités de sa machine virtuelle.

Lequel de ces types de langage utilisons-nous dans ce texte ? Tous, en fait.

Le langage de programmation principal de Sage est Python, que nous avons listé ci-dessus comme langage interprété. Cependant, Sage et Python ne sont pas tout à fait identiques ; les programmes Sage ne fonctionnent pas en Python standard, et certaines fonctionnalités de Python sont différentes dans Sage.

Vous pouvez parfois compiler des programmes Sage en utilisant un programme [Cython](#) ; Nous aborderons ce sujet vers la fin du texte. Cependant, Sage « cythonisé » ne fonctionnera pas seul ; vous devrez l'exécuter dans un environnement Sage.

Comme vous l'apprendrez ci-dessous, Sage est en fait assemblé à partir d'un grand nombre de pièces distinctes. Certains d'entre eux sont écrits avec Java, ce qui signifie que vous utilisez un langage bytecode, même si vous n'écrivez aucun programme Java.

C'est quoi ce truc Sage dont tu parles sans cesse ?

Sage est un système d'algèbre informatique gratuit et open source.

---

<sup>2</sup>Dans ses incarnations originales, Pascal a été traduit en une idée similaire appelée P-code, plus tard en bytecode proprement dit.

<sup>3</sup>De nombreux langages interprétés sont désormais compilés « incrémentalement ». Ils enregistrent l'interprétation de chaque commande et au fur et à mesure de leur progression dans l'exécution, vérifiez si chaque nouvelle commande a déjà été interprétée.

Qu'est-ce qu'un « système de calcul formel » ? Traditionnellement, il existe trois grands types de progiciels mathématiques à grande échelle :

- Les systèmes de calcul numérique visent des calculs rapides, généralement basés sur des nombres à virgule flottante. Nous n'entrerons pas dans les détails pour l'instant, mais on peut considérer la virgule flottante comme une sorte d'« estimation précise », similaire à l'utilisation des chiffres significatifs en sciences. La science derrière le calcul numérique relève généralement de l'analyse numérique. Presque tout le monde dans les pays développés a utilisé un système de calcul numérique à un moment donné en utilisant une calculatrice. Parmi les logiciels numériques, on trouve [MATLAB](#). et [Octave](#).
- Les logiciels statistiques sont des systèmes de calcul numérique spécifiques qui utilisent des outils spécifiques à l'analyse statistique. [SAS](#) en est un exemple. et le [projet R](#). Les systèmes de calcul formel visent le calcul exact, même si cela se fait au détriment de la rapidité associée aux systèmes numériques. Plutôt que de manipuler des valeurs approximatives, les systèmes de calcul formel manipulent des symboles qui représentent des valeurs exactes : ils ne considèrent pas  $\pi$  comme un nombre décimal, mais comme un symbole doté des propriétés mathématiques qui lui sont associées, et ils permettent de manipuler des expressions impliquant des variables. De ce fait, la science qui sous-tend les systèmes de calcul formel est appelée calcul symbolique, algèbre computationnelle ou calcul formel. Quelques calculatrices parmi les plus coûteuses utilisent le calcul symbolique, mais on travaille généralement avec un logiciel comme [Maxima](#). [Érable](#), ou [Sage](#).

Pourquoi sacrifier l'exactitude au profit des valeurs approximatives d'un système numérique ? La principale raison est la rapidité ! En sacrifiant un peu de précision, un système numérique peut facilement traiter des nombres, des vecteurs et des matrices, petits comme grands, sans trop de difficultés ; il n'est pas rare de travailler avec des centaines, voire des milliers de variables dans un système d'équations.

Par exemple, que se passe-t-il si l'on additionne les fractions  $1/2$ ,  $1/3$ ,  $1/5$  et  $1/7$  ? Chacune d'elles ne nécessite que deux chiffres (numérateur et dénominateur), mais la somme exacte,  $247/210$ , nécessite six chiffres, soit une augmentation de 300 % ! Si l'on utilise des nombres à virgule flottante à quatre chiffres maximum, la somme devient

$$0,5000 + 0,3333 + 0,2000 + 0,1429 = 1,176.$$

La somme n'est pas supérieure aux nombres d'origine. Certes, elle est légèrement erronée, mais l'erreur est inférieure à  $1/500$  ; c'est bien plus précis que ce qu'exigent la plupart des tâches quotidiennes.

La croissance rapide de la taille des fractions est l'une des raisons pour lesquelles les gens n'aiment généralement pas les fractions ; personne n'aime travailler avec des objets dont la complexité augmente rapidement. Dans ce cas, ce qui est vrai pour les humains l'est aussi pour les ordinateurs ; si vous travaillez sur un problème nécessitant la division d'entiers dans Sage, vous constaterez presque certainement un ralentissement important à mesure que les fractions deviennent complexes et, par conséquent, plus difficiles à traiter pour l'ordinateur.

Dans ce cas, pourquoi s'embêter avec des calculs symboliques et des valeurs exactes ? Pour de nombreux problèmes, les imprécisions du calcul en virgule flottante le rendent totalement inadapté.

Cela est particulièrement vrai lorsque la division est une partie inévitable du problème.

Par exemple, supposons que l'ordinateur doive diviser par le nombre à virgule flottante à 4 chiffres 0,0001.

Le quotient résultant devient très grand. Pourtant, il est tout à fait possible que 0,0001 soit le résultat d'une erreur de virgule flottante, et que la valeur correcte soit en réalité 0. Comme vous le savez sans doute, la division par 0 est mauvaise. Si l'ordinateur avait su que la valeur était 0, il n'aurait pas divisé du tout ! Les problèmes où cela peut se produire sont souvent qualifiés de « mal conditionnés », et les analystes numériques passent beaucoup de temps à essayer de les éviter.

Dans certains cas, cependant, ce n'est pas possible, et on a alors recours à des valeurs exactes. C'est particulièrement vrai lorsqu'on aborde des domaines mathématiques plus abstraits. Certains pensent que les mathématiques « abstraites » sont des mathématiques « inutiles », mais ils se trompent lourdement : ce texte vous présentera plusieurs objets mathématiques abstraits dont l'exactitude même permet la communication extrêmement précise et sécurisée que vous établissez sur Internet lorsque vous consultez votre compte bancaire ou effectuez un achat en ligne.

Qu'est-ce qui rend Sage si spécial ? Sage a été « fondé » par [William Stein](#), un théoricien des nombres. Il était frustré par plusieurs inconvénients des systèmes d'algèbre informatique disponibles à l'époque :

Les systèmes commerciaux, comme Maple, ne permettent pas à l'utilisateur de visualiser le code, et encore moins de le modifier. Dans le monde du logiciel, ces systèmes sont qualifiés de « propriétaires », « fermés » ou « non libres » (par certains). Il existait également des systèmes « ouverts » ou « libres » qui, issus de recherches de pointe, étaient souvent plus performants que les progiciels commerciaux pour une tâche donnée. Cependant, ces progiciels n'excellent que dans un domaine particulier des mathématiques : • pour le calcul, on utiliserait probablement

Maxima ; • pour l'algèbre linéaire, on utiliserait probablement [Linbox](#) ; • pour la théorie des groupes, vous utiliseriez probablement [GAP](#) ; • pour la théorie des nombres, vous utiliseriez probablement [Pari](#) ; • pour l'algèbre commutative, vous utiliseriez probablement [CoCoA](#), [Macaulay](#), ou [singulier](#); Et ainsi de suite. Pire encore,

supposons que vous ayez besoin de transférer les résultats d'un package vers un autre : après avoir appliqué la théorie des nombres avec Pari, par exemple, vous pourriez vouloir analyser les résultats à l'aide de la théorie des groupes, auquel cas GAP serait l'outil idéal. Mais il n'existe aucun moyen simple de copier les résultats de Pari vers GAP.

Sage a donc été conçu pour regrouper ces outils performants en un seul package relativement simple d'utilisation. Des fonctionnalités supplémentaires ont été programmées dans Sage lui-même, et dans certains cas, Sage s'est révélé être le leader dans la résolution de certains problèmes. En prime, les développeurs de Sage ont rendu possible l'interaction avec de nombreux packages propriétaires via Sage. Ainsi, si Maple dispose des outils les plus rapides pour résoudre un problème et que vous en possédez une copie, vous pouvez le faire via Maple, puis manipuler le résultat via Sage.

Pourquoi s'intéresser aux logiciels « libres » ? Dans le monde du logiciel, le terme « libre » a deux significations : sens:

Gratuit comme la bière : vous n'avez pas à payer pour cela.

Libre comme dans la parole : le code est ouvert et visible, plutôt que « censuré ».

Un logiciel peut être « gratuit comme la bière », mais pas « libre comme la parole » ; autrement dit, il est gratuit, mais son code source est inaccessible. Par exemple, de nombreux programmes « gratuits » peuvent être téléchargés pour ordinateur ou téléphone portable.

Il existe des raisons importantes pour lesquelles un chercheur ou même un ingénieur devrait pouvoir visualiser et/ou modifier le code d'un logiciel mathématique :

- Les bonnes pratiques scientifiques exigent reproductibilité et vérifiabilité. Or, un chercheur ne peut vérifier les résultats d'un calcul mathématique s'il ne peut vérifier la manière dont il a été effectué. • Tout logiciel de taille significative comporte des erreurs, appelées bugs. Si deux logiciels produisent des résultats différents, un chercheur ne peut pas déterminer lequel est correct s'il ne peut pas visualiser le code et évaluer les algorithmes.

- Presque toutes les recherches mathématiques s'appuient sur des travaux antérieurs. Il en va de même pour les logiciels : les chercheurs doivent souvent étendre un logiciel avec de nouvelles fonctionnalités pour accomplir une tâche. Cela peut être beaucoup plus difficile à réaliser correctement si le chercheur ne peut pas visualiser le code, et encore moins le modifier.

Par exemple, supposons qu'un mathématicien prétende détenir la preuve de l'existence d'une infinité de nombres premiers. La plupart des mathématiciens souhaiteraient voir la preuve ; c'est une façon d'apprendre les uns des autres. (Dans certains cas, la preuve est bien plus intéressante que le théorème.) On trouve d'ailleurs cette preuve dans de très nombreux manuels, car le mathématicien hellénique [Euclide d'Alexandrie](#) a enregistré ce qui est considéré comme l'une des plus belles preuves de ce fait il y a plus de deux mille ans [10, Livre IX, Proposition 20] : THÉORÈME. Il existe une infinité de nombres premiers.

**PREUVE.** Considérons un ensemble fini et non vide de nombres premiers,  $P = \{ p_1, \dots, p_n \}$ . Soit  $q = p_1 \cdots p_n + 1$ . Puisque  $q > 1$ , au moins un nombre premier le divise, mais le reste de la division de  $q$  par tout  $p \in P$  est égal à 1 ; aucun des nombres premiers de  $P$  ne le divise. Il doit donc exister un nombre premier non listé dans  $P$ . Or,  $P$  est un ensemble fini arbitraire de nombres premiers, ce qui signifie qu'aucun ensemble fini de nombres premiers ne peut les lister tous. Autrement dit, il existe une infinité de nombres premiers.

En exposant clairement la preuve, Euclide facilite la vérification du résultat. Il facilite également la remise en question de l'argument : on pourrait se demander, par exemple, comment Euclide sait qu'au moins un nombre premier divise tout entier positif différent de 1. (Il l'a d'ailleurs prouvé plus haut.)

Comparez cela à un autre théorème célèbre attribué à [Pierre de Fermat](#), un juriste français qui a étudié les mathématiques comme passe-temps [9, p. 61, Observatio Domini Petri de Fermat] : THÉORÈME.

Si $n > 2$ , l'équation $a^n + b^n = c^n$ n'a pas de solution avec des entiers	$= c$
--	-------

**PREUVE.** J'ai trouvé une preuve vraiment remarquable de ce fait. La petitesse de cette marge ne suffira pas à la contenir.

Ces deux phrases ont déclenché une recherche de preuve qui a duré plus de trois siècles ; [Andrew Wiles](#) Il a trouvé une preuve en 1994, et à ce jour, il n'en existe aucune autre. La plupart des gens s'accordent à dire que Fermat n'avait pas de preuve, mais il ne faut pas le critiquer : il n'a jamais révélé à personne qu'il en avait une ; il s'est contenté d'écrire ce commentaire en marge d'un livre. Son fils a publié un exemplaire de l'ouvrage après la mort de Fermat, en y incluant les notes qu'il avait écrites en marge.

Pour pousser l'analogie plus loin, supposons que nous affirmions ce qui suit :

THÉORÈME. Le nombre  $2^n - 1$  est premier pour  $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$ .

**PREUVE.** Secret commercial.

Vous auriez raison de douter de notre affirmation, pour au moins deux raisons : il n'existe aucun moyen simple de la vérifier, et elle est même fausse ! Pourtant, cette affirmation a été formulée par un mathématicien réputé, [Marin Mersenne](#). [8], proposée sans preuve, a été généralement acceptée pendant un certain temps. Vous retrouverez cette affirmation ultérieurement en laboratoire.

Quels sont les avantages de Sage ? Comme mentionné précédemment, Sage facilite l'expérimentation avec des objets mathématiques que vous utiliserez de plus en plus en classe après celui-ci. Plus tard

les cours ne nécessiteront probablement pas explicitement Sage, mais si vous n'utilisez plus jamais Sage après ce cours, ce serait comme suivre un cours de statistiques et faire tout le travail à la main :4 POURQUOI?!?

Un autre avantage de Sage est que vous interagissez avec lui via une interface Python ; la programmation dans Sage est, dans une certaine mesure, identique à la programmation en Python. Cela confère les avantages suivants : • Rappelons que Python est l'un des

langages interprétés les plus répandus.

- De nombreux employeurs recherchent une expérience Python. Bien apprendre Sage vous aide à apprendre Python et vous aide à devenir plus employable.
- De nombreux packages sont disponibles pour améliorer Python et fonctionnent parfaitement avec Sage. De nombreux packages de ce type sont d'ailleurs déjà inclus dans Sage.
- Comme mentionné précédemment, vous pouvez souvent accélérer un programme en le « Cythonisant ».
- Python est un langage moderne, offrant de nombreuses façons d'exprimer une solution élégante à un problème. En apprenant Sage, vous apprenez de bonnes pratiques de programmation.

Gardez à l'esprit que Python et Sage sont différents, et qu'ils ne sont pas non plus un sous-ensemble de l'autre. Les commandes Sage ne fonctionnent pas en Python standard, et certaines commandes Python ne fonctionnent pas de la même manière dans Sage qu'en Python.

#### Comment démarrer avec Sage ?

Le moyen le plus simple est probablement de visiter [le serveur CoCalc sur cocalc.com](https://cocalc.com), Inscrivez-vous gratuitement, démarrez un projet et créez une feuille de calcul Sage. Nous vous encourageons vivement à investir dans un abonnement, qui, au moment de la rédaction de cet article, coûte 7 \$ par mois et donne accès à des serveurs plus rapides et plus fiables. L'utilisation est gratuite, mais pour diverses raisons, les serveurs gratuits peuvent parfois se réinitialiser. (Si vous participez à un cours, demandez à l'enseignant si le département a proposé un forfait ; la réduction est substantielle.) L'inconvénient de cette approche est qu'il faut payer pour bénéficier d'un service de qualité. L'avantage est que vous disposez toujours d'une version relativement récente de Sage à portée de main, et vous n'avez pas à craindre une panne matérielle qui effacerait toutes vos données, car celles-ci sont stockées sur des serveurs qui s'appuient sur de multiples sauvegardes et mécanismes de secours.

Une autre façon d'utiliser Sage est d'utiliser un serveur en ligne autre qu'un serveur CoCalc. Cela nécessite de connaître quelqu'un qui connaît quelqu'un qui... connaît quelqu'un qui gère un serveur que vous pouvez utiliser. De nombreuses institutions le font, y compris celle qui emploie les auteurs ; d'ailleurs, notre département gère au moins deux serveurs de ce type au moment de la rédaction de cet article. L'inconvénient de cette approche est que vous dépendez du mainteneur du serveur pour maintenir le logiciel à jour et sauvegarder les données. D'ailleurs, l'un des serveurs de notre département utilise une version de Sage obsolète depuis des années.

La dernière façon d'utiliser Sage est d'en disposer sur votre propre machine. Vous pouvez le télécharger ici. [www.sagemath.org](http://www.sagemath.org) et installez-le sur votre ordinateur. (Recherchez le lien vers « Téléchargements ».)

Si votre machine fonctionne sous Linux, le processus est relativement simple, même si vous devrez peut-être installer certains paquets via votre gestionnaire de paquets. (Par le passé, nous avons dû installer un système appelé m4. Nous ne nous souvenons plus de ce que c'est.) Des binaires sont disponibles pour certaines distributions Linux ; Ubuntu en fait généralement partie, et Fedora l'a été occasionnellement. Pour les autres, vous devrez probablement télécharger les sources de Sage et les compiler sur votre ordinateur. La bonne nouvelle, c'est que c'est généralement assez simple, à condition de l'avoir déjà fait.

---

Au moins un des auteurs a réellement tenté cette expérience lorsqu'il était à l'université. En fait, les statistiques ont été le seul cours qui a brisé sa résistance à l'utilisation d'une calculatrice. La capacité de la calculatrice à effectuer des calculs précis sur des fractions l'a impressionné ; jusque-là, il n'avait vu des calculatrices effectuer des calculs avec des fractions qu'en virgule flottante.

J'ai installé les paquets requis, qui sont listés dans les instructions. La mauvaise nouvelle, c'est que l'installation depuis les sources est assez longue ; préparez-vous donc à une certaine attente. • Si votre machine fonctionne sous OS X, essayez de télécharger un binaire adapté à votre architecture. Vous pouvez essayer de compiler depuis les sources, comme avec Linux, mais dans ce cas, espérez qu'Apple n'ait pas récemment « mis à jour » Xcode, car vous avez besoin de Xcode pour installer Sage, et chaque version majeure de Xcode casse généralement Sage d'une manière ou d'une autre.<sup>5</sup> • Si votre machine fonctionne sous Windows, vous êtes dans la situation inhabituelle de subir ce que subissent généralement les utilisateurs de Linux et d'OS X : Sage ne fonctionne pas nativement sous Windows, vous devez donc l'utiliser via une machine virtuelle. La configuration peut être un peu fastidieuse, mais une fois prise en main, cela fonctionne parfaitement. La bonne façon de procéder a changé plusieurs fois au fil des ans, nous hésitons donc à donner des conseils plus précis. La bonne nouvelle, c'est que les instructions d'installation et d'exécution de Sage seront disponibles sur le site web.

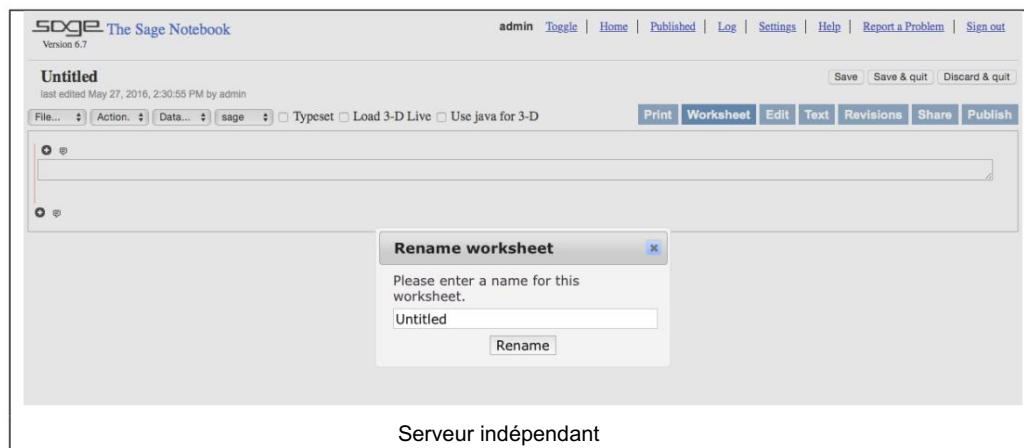
Une fois que vous avez mis en place l'une de ces méthodes, vous pouvez commencer à l'utiliser !

### Feuille de calcul ou ligne de commande ?

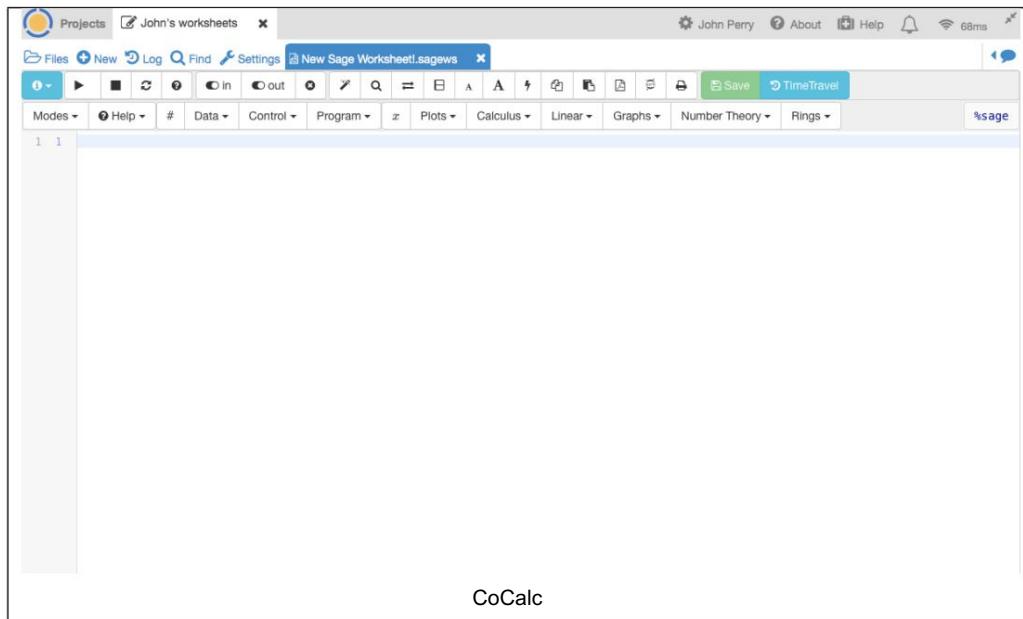
Il existe deux manières courantes d'utiliser Sage : depuis un navigateur, dans une feuille de calcul Sage, ou depuis la ligne de commande. Si Sage est installé sur votre ordinateur, vous pouvez utiliser les deux ; consultez la section « Sage en ligne de commande » pour savoir comment démarrer une feuille de calcul Sage depuis la ligne de commande.

Feuilles de calcul Sage. Si vous avez accès à Sage via un navigateur web (CoCalc ou un autre serveur en ligne), vous préferez probablement travailler avec une feuille de calcul Sage. Nous recommandons à nos étudiants de commencer avec Sage de cette manière, car la feuille de calcul offre un environnement plus convivial : il est facile de réviser et de réexécuter des commandes, d'ajouter du texte pour l'organisation et la narration, et de sauvegarder l'intégralité de votre session, puis de recharger ultérieurement le travail et le résultat. Tout cela est impossible avec la ligne de commande.

Lorsque vous démarrez une feuille de calcul, vous devriez voir l'un de ces deux écrans :



<sup>5</sup>Sage n'est en aucun cas le seul logiciel à subir cette conséquence.



Vous pouvez (et devriez) modifier le titre. • Sur le

serveur indépendant, vous pouvez le faire au début grâce à la boîte de dialogue « Renommer la feuille de calcul » illustrée dans la capture d'écran. Vous pouvez le faire ultérieurement en cliquant sur le titre (en haut à gauche, actuellement « Sans titre ») et la même boîte de dialogue apparaîtra. • Dans CoCalc, vous pouvez le faire en cliquant sur le « i » entouré en haut à gauche et en choisissant « Renommer... ». Un nouvel écran apparaîtra, vous invitant à renommer le fichier. Assurez-vous de conserver l' extension .sagews ajoutée à la fin.

Il existe d'autres options que vous pouvez utiliser, mais pour l'instant, nous vous recommandons de passer au chapitre suivant, car la plupart d'entre elles sont de peu d'importance pour notre propos. Nous aborderons celles qui nous sont utiles ultérieurement.

Ligne de commande Sage. Si vous choisissez d'exécuter Sage depuis la ligne de commande, vous devez ouvrir un shell, également appelé invite de commande. Vous verrez une invite de commande, dont la forme peut varier considérablement ; lorsqu'il s'agit d'une invite de commande, nous utiliserons simplement un symbole bleu supérieur à : >. À l'invite, saisissez « sage », appuyez sur Entrée, puis attendez que Sage démarre. Cela peut prendre quelques secondes, mais vous verrez finalement quelque chose comme ceci :

```
> sage
SageMath Version 6.7, Date de sortie : 17/05/2015 Tapez
« notebook() » pour l'interface du bloc-notes basée sur un navigateur.
Tapez « help() » pour obtenir de l'aide.
sage: _
```

Le trait de soulignement (\_) peut ressembler à un petit cadre sur votre système. Une fois que vous le voyez, vous êtes prêt pour le chapitre suivant. Si vous ne le voyez pas, ou si vous constatez une erreur, contactez votre instructeur ou le support technique pour déterminer la cause du problème.

Si vous souhaitez exécuter une feuille de calcul Sage dans un navigateur, mais que vous ne souhaitez pas exécuter CoCalc et que vous n'avez pas accès à un autre serveur, saisissez notebook() et appuyez sur Entrée. De nombreux messages s'afficheront, par exemple :

```
sage: notebook()
Les fichiers du bloc-notes sont stockés dans : sage_notebook.sagenb Ouvrez votre
navigateur Web sur http://localhost:8080 Exécution de twistd --
pidfile="sage_notebook.sagenb/sagenb.pid" -ny "sage_notebook.sagenb/twistedconf.tac"

/Applications/sage-6.7-untouched/local/lib/python2.7/site-packages/ Crypto/Util/number.py:57:
PowmInsecureWarning : mpz_powm_sec n'est pas utilisé. Reconstruisez avec libgmp >= 5 pour
éviter une vulnérabilité aux attaques temporelles. _warn("MPz_powm_sec n'est pas utilisé. Reconstruisez avec
libgmp >= 5 pour éviter une
vulnérabilité aux attaques temporelles.", PowmInsecureWarning)

2016-05-27 14:30:49+0300 [-] Journal ouvert. 2016-05-27
14:30:49+0300 [-] Démarrage de twistd 14.0.2 (/Applications/sage-6.7-
untouched/local/bin/python 2.7.8)
en
haut. 2016-05-27 14:30:49+0300 [-] classe de réacteur :
twisted.internet.selectreactor.SelectReactor.

2016-05-27 14:30:49+0300 [-] Démarrage de QuietSite sur 8080 2016-05-27 14:30:49+0300 [-] Démarrage
de l'instance d'usine <__builtin__.QuietSite à 0x1181bb638>
```

La plupart du temps, vous n'avez pas à vous soucier de ces messages. Vous n'avez même pas besoin de suivre les instructions pour ouvrir votre navigateur web sur ce site ; sur de nombreux ordinateurs, le navigateur ouvre la page web automatiquement. De plus, le démarrage prend quelques secondes, alors détendez-vous. Si votre navigateur ne s'ouvre pas, pas de panique ! Ouvrez-le vous-même et vérifiez si l'adresse web indiquée par votre Sage fonctionne. Si c'est le cas, vous êtes prêt pour la suite.

Si ce n'est pas le cas,

## PANIQUE!

Oui, c'est tout à fait normal de paniquer de temps en temps. Expliquez-vous. Une fois terminé, examinez attentivement les messages et voyez s'il s'agit de messages d'erreur ; cela pourrait vous aider. Ensuite, consultez

<https://groups.google.com/forum/#forum/sage-support> et vérifiez si ce message d'erreur

a été abordé. Sinon, créez un nouveau message pour en savoir plus sur le problème.

### Obtenir de l'aide

Si vous lisez ceci dans le cadre d'un cours, votre instructeur devrait vous être utile. (Peut-être pas très utile, mais utile tout de même.) Nous avons déjà mentionné le forum de support sage dans le dernier chapitre. Outre ces options, Sage répondra lui-même à de nombreuses questions.

Docstrings. Pour comprendre le fonctionnement d'une commande, saisissez son nom suivi d'un point d'interrogation, puis exécutez-la. Sage vous fournira des informations utiles sur la commande, généralement accompagnées d'exemples.

---

Eh bien, peut-être les avertissements de sécurité concernant libgmp, si vous les voyez. Je devrais y jeter un œil.

```
sage : simplifier ?
Signature : simplifier(f)
Docstring :
    Simplifiez l'expression f.
```

**EXEMPLES :** Nous simplifions l'expression  $i + x - x$ .

```
sage : f = i + x - x ; simplifier(f)
```

je

En fait, l'impression de  $f$  donne la même chose, c'est-à-dire la forme simplifiée.

```
Docstring d'initialisation : x.__init__(...) initialise x ; voir help(type(x)) pour la signature Fichier : /Applications/
sage-6.7-untouched/
local/lib/python2.7/site-packages/sage/calculus/functional.py Type : fonction
```

Trouver des méthodes pour les objets. Une autre façon d'obtenir de l'aide est de voir quelles commandes un objet accepte comme méthodes. Une « méthode » est une commande très spécifique à un objet Sage particulier ; vous l'invoquez en saisissant le nom de l'objet, suivi d'un point, puis du nom de la méthode. Pour trouver le nom de toutes les méthodes acceptées par un objet, saisissez son nom, suivi d'un point, puis appuyez sur la touche Tab .

Par exemple, `simplify()` ne fonctionne pas très bien sur l'expression suivante :8

```
sage : rats = 1/x + 1/2 sage :
simplifier(rats)
1/x + 1/2
```

Nous souhaitons simplifier cette expression au maximum. Voyons donc si elle accepte une méthode permettant une simplification plus poussée. Tapez « `rats.` » (avec le point !) et appuyez sur Tab ; vous devriez voir plus de 200 méthodes possibles. Certaines ne sont pas vraiment adaptées à l'expression ; nous n'en détaillerons pas les raisons, mais en cherchant bien, vous devriez trouver au moins deux méthodes utiles. L'une d'elles est « `full_simplify()` ».

```
sage : rats.full_simplify() 1/2*(x + 2)/x
```

C'est une façon un peu compliquée d'écrire  $(x+2)/(2x)$ , mais au moins c'est simplifié !

<sup>7</sup>Un autre terme pour « méthode » est « message » ; les deux termes sont utilisés en informatique, mais « méthode » est le jargon de Sage.

<sup>8</sup>Cela fonctionne effectivement dans certaines versions de Sage, comme Sage 6.7, mais pas dans d'autres, comme Sage 8.1. Si cela simplifie votre version, faites comme si ce n'était pas le cas pour les besoins de l'argumentation et poursuivez, car ce n'est de toute façon pas le but.

L'idée de parcourir plus de 200 méthodes possibles peut paraître intimidante, mais de nombreux objets n'acceptent que très peu de méthodes. En pratique, cette technique n'est pas si difficile, car il existe de nombreuses façons de parcourir la liste.

### Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Ceci est juste un autre manuel de programmation.
2. Les mathématiques consistent à compter les nombres.
3. Bien que les élèves ne s'intéressent pas beaucoup aux fractions, les ordinateurs trouvent plus facile de travailler avec des fractions qu'avec des nombres à virgule flottante.
4. Les logiciels « libres » sont écrits par des fainéants au chômage qui vivent dans le sous-sol de leurs parents.
5. Tous les résultats mathématiques célèbres ont été appréciés dès le début pour leurs démonstrations claires et élégantes.
6. Le bytecode n'est techniquement pas un langage interprété.
7. Les langues interprétées sont appréciées avant tout pour leur rapidité.
8. Certains logiciels de taille significative sont exempts de bugs.
9. Les mathématiques abstraites sont inutiles dans le monde réel.
10. Il est facile de vérifier et d'améliorer les packages mathématiques propriétaires.
11. Bonus : La réponse à toutes ces questions Vrai/Faux est « Faux ».

Choix multiple.

1. Lequel des éléments suivants n'est pas un exemple de système d'algèbre informatique ?
  - A. Un package qui effectue des calculs arithmétiques sur les fractions sans aucune erreur d'arrondi.
  - B. Un package qui attribue des numéros aux notes de musique et utilise des fractales pour produire de la nouvelle musique.
  - C. Un package qui effectue une arithmétique polynomiale avec des valeurs approximatives pour les coefficients.
  - D. Un package qui se concentre sur des ensembles d'objets abstraits définis par des propriétés précises.
2. Laquelle des étapes suivantes n'est pas une étape du processus habituel de compilation d'un programme ?
  - A. Le compilateur lit la source à partir d'un fichier.
  - B. Le compilateur traduit chaque symbole en une combinaison de signaux marche et arrêt.
  - C. Le compilateur enregistre la traduction dans un fichier, appelé exécutable ou bibliothèque.
  - D. Le compilateur exécute les commandes traduites immédiatement avant de quitter.
3. Quel type de mathématiques computationnelles constitue l'objectif principal de Sage ?
 

A. calcul numérique avec des valeurs approximatives    B. calcul statistique  
     avec des valeurs probables    C. calcul symbolique avec des valeurs  
     exactes    D. calcul flou avec des valeurs incertaines
4. Lequel des systèmes d'algèbre informatique suivants utiliseriez-vous pour calculer une dérivée ?
 

A. Maxima    B.  
     PARI    C.  
     Schoonship    D.  
     SINGULAR
5. Lequel des langages informatiques bien connus suivants tente de combler le fossé entre les langages interprétés et compilés ?
 

A. C/C++    B.  
     Fortran    C.  
     Java    D.  
     Python

6. Laquelle des motivations suivantes est la principale motivation du mouvement pour les mathématiques « libres » logiciel?

- A. Antipathie envers la censure B.
- Collaboration internationale C. Manque de financement par subvention
- D. Vérification des résultats 7.

Lequel des mathématiciens suivants est célèbre malgré la popularisation d'un « fait » qui était très faux ?

- A. Pierre de Fermat B.
- Marin Mersenne C.
- William Stein D.
- Andrew Wiles 8.

Lequel des mathématiciens suivants est célèbre malgré son travail non scientifique ?

- A. Pierre de Fermat B.
- Marin Mersenne C.
- William Stein D.
- Andrew Wiles 9.

Lequel des éléments suivants n'est pas un avantage à travailler avec Sage ?

- A. Vous acquérez des compétences pratiques que les employeurs apprécient.
- B. Vous travaillez avec des logiciels de pointe.
- C. Vous n'avez plus à vous soucier d'obtenir la bonne réponse.
- D. Vous pouvez utiliser le matériel appris ici dans d'autres cours.

10. De laquelle de ces manières Sage fonctionne-t-il avec Python ?

- A. L'interface de Sage est essentiellement une interface Python.
- B. Sage est une bibliothèque Python que vous pouvez charger dans n'importe quel interpréteur Python.
- C. Python est l'un des systèmes d'algèbre informatique de Sage.
- D. Sage utilise le compilateur Cython pour compiler tous les programmes Python.

Bonus : Quelle réponse est correcte ?

- A. Le suivant.
- B. Le suivant.
- C. Le suivant.
- D. Le premier.

Réponse courte.

1. Expliquez comment la citation au début de ce chapitre est liée à sa thèse principale.

2. Décrivez une analogie du monde réel pour la différence entre le code compilé, interprété et le bytecode langues.

3. Un problème courant dans les manuels de mathématiques est de calculer la 1001e dérivée de  $\cos x$ .

La meilleure façon de trouver la réponse n'est pas de calculer les 1001 dérivées ; il suffit d'en calculer quelques-unes, d'observer une tendance et d'en déduire rapidement la 1001e dérivée. Expliquez comment cela se compare à ce que nous souhaitons vous apprendre de ce texte.

4. Tous les mathématiciens ne trouvent pas convaincants les arguments en faveur du logiciel libre, et l'utilisation L'utilisation de logiciels propriétaires est répandue. À votre avis, pourquoi ?

5. Même si Sage est moins populaire que des logiciels propriétaires comme Maple ou Mathematica, apprendre à utiliser Sage peut également faciliter l'utilisation de ces logiciels. Pourquoi ?

## CHAPITRE 2

## Calculs de base

Qu'y a-t-il dans un nom ? Ce que nous appelons une rose / Sous n'importe quel autre nom sentirait aussi bon...  
 (Shakespeare)

Créez une nouvelle feuille de calcul et appelez-la « Ma première feuille de calcul Sage ».

Pour décrire l'interaction avec Sage, nous utilisons le format suivant :

```
sage : une entrée, une
sortie
```

Le texte « some input » indique une commande que vous saisissez dans Sage. Si vous utilisez Sage en ligne de commande, vous saisissez cette commande dans une invite « sage: » bleue ; d'où la couleur et le libellé bleus.

Si vous utilisez Sage à partir d'une feuille de calcul, vous ne verrez pas d'invite ; à la place, vous saisissez la commande dans une « cellule », qui dans certaines versions de Sage est délimitée par une petite case.

Pour exécuter la commande :

- sur la ligne de commande, appuyez sur la touche Entrée ou Retour ; • dans une feuille de calcul, maintenez la touche Maj enfoncée et appuyez sur la touche Entrée ou Retour .

Sage interprétera et traitera ensuite votre commande. L'interface de la feuille de calcul affichera une barre colorée ou clignotante pendant cette opération ; l'interface de ligne de commande se mettra simplement en pause.

Une fois la sortie terminée, vous verrez le texte indiqué par « some output ». Si la commande a réussi, vous obtiendrez une réponse plus ou moins logique. En cas d'erreur, le texte contiendra de nombreuses informations, dont certaines en rouge (notamment la dernière ligne, qui est la seule que nous copierons), et n'aura probablement aucun sens si vous ne connaissez pas déjà Python. Par exemple :

```
sage : quelques mauvaises entrées
SomeError : un message expliquant l'erreur
```

Nous expliquerons plusieurs types d'erreurs au fur et à mesure de notre exploration de Sage. Pour plus de commodité, nous ajoutons des erreurs à l'index du manuel. Si vous rencontrez une erreur que vous ne reconnaissiez pas lors d'un calcul, essayez de la retrouver dans l'index ; si oui, les numéros de page référencés vous seront utiles.

Voici un exemple de calcul réussi ; vous devriez essayer vous-même maintenant et vous assurer d'obtenir le même résultat.

```
sauge : 2 + 3
5
```

Cela semble rassurant ; au moins Sage peut faire ça ! En voici un où l'utilisateur a oublié de relâcher la touche Maj avant d'appuyer sur le chiffre 3.

```
sage : 2 + #
SyntaxError : syntaxe non valide
```

Si vous exécutez cette commande dans Sage, vous remarquerez que le texte d'erreur est un peu plus long et que les couleurs ont beaucoup changé. Dans notre cas, cela ressemble à ceci :

```
Fichier « <ipython-input-3-7c2c726856a7> », ligne 1
Entier(2) + #
^

SyntaxError : syntaxe non valide
```

Vous remarquerez que nous n'avons copié que la dernière ligne, qui précise le type d'erreur ! Si nécessaire pour la discussion, nous inclurons parfois le reste des informations, mais la dernière ligne permet souvent de comprendre le problème.

Il peut arriver que la ligne à saisir soit trop longue pour tenir sur une seule ligne. Ce sera le cas des auteurs de ce texte, car nous manquons d'espace horizontal sur la page. Dans ce cas, vous pouvez continuer à saisir, ce qui peut rendre le code plus difficile à lire, ou indiquer à Sage que vous souhaitez continuer sur la ligne suivante en saisissant une barre oblique inverse \, interprétée par Sage comme un saut de ligne, puis en appuyant sur Entrée. Nous le ferons régulièrement pour améliorer la lisibilité du code. Il n'est pas nécessaire d'attendre la fin de la ligne pour cela, et il peut parfois être plus lisible d'ajouter le saut de ligne plus tôt. Par exemple :

```
sauge : 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 \
....: + 11 + 12 + 13 + 14 + 15
120
```

(Lors de l'utilisation d'une feuille de calcul Sage). L'un des avantages des feuilles de calcul Sage est que vous pouvez utiliser Commandes HTML pour ajouter une explication à votre travail. Saisissez simplement

```
sage: %html
```

... et tout ce qui suit sera considéré comme du texte HTML. La barre d'outils sera modifiée pour permettre le formatage HTML, mais si vous maîtrisez les balises HTML, vous pouvez les ajouter directement. Vous pouvez le formater de la même manière que pour une commande Sage : maintenez la touche Maj enfoncée et appuyez sur Entrée. Si nécessaire, vous pouvez ensuite modifier à nouveau la cellule HTML en double-cliquant dessus. Cette fonctionnalité est extrêmement utile pour diviser une longue feuille de calcul en sections, avec des en-têtes organisant les parties connexes du travail.

## Votre arithmétique de base

Comme vous pouvez vous y attendre, Sage propose les mêmes opérations arithmétiques de base que n'importe quelle calculatrice. En voici quelques-unes qui vous seront utiles.

---

<sup>1</sup>Vous pouvez également saisir  $a^b$  pour l'exponentiation, mais pour diverses raisons, nous ne le recommandons pas : dans certaines situations, Sage l'interprétera comme un opérateur différent.

a+b	additionne a et b
ab	soustrait a et b a*b
multiplie a et b	a/b trouve le
rapport de la division de a par ba	// b trouve
le quotient de la division de a par b	a%b trouve le
reste de la division de a par b	a**b élève a à la
puissance b	sqrt(a) la racine carrée
de a abs(a)	la valeur absolue de
a	

TABLEAU 1. Opérateurs Sage pour l'arithmétique de base

N'hésitez pas à essayer ces méthodes avec quelques chiffres, à la fois approximatifs et exacts.

Vos comparaisons de base. Sage peut également comparer des objets, dans une certaine mesure.

a>b	est-il strictement supérieur à b ?	a>=b
est-il supérieur ou égal à b ?	a==b	est-il égal à
b ?	a<=b	est-il inférieur ou
égal à b ?	a<b	est-il strictement inférieur à b

TABLEAU 2. Opérateurs Sage pour les comparaisons de base

Lors de l'utilisation de ces comparaisons, Sage renverra Vrai ou Faux, ce qui correspond évidemment à « oui » ou « non ».

```
sage : 2 < 3
Vrai
sage : 2 > 3
FAUX
```

Il faut être prudent avec la comparaison d'égalité, car le signe est doublé. Des problèmes surgiront si on l'oublie.

```
sage : 2 = 3
ValueError : le nom « 2 » n'est pas un identifiant Python valide.
```

Si vous voyez ce message d'erreur, le problème est presque certainement dû à l'utilisation d'un seul signe égal alors que vous en avez besoin de deux.

Outre la comparaison de nombres, de nombreux objets symboliques peuvent être comparés. Nous n'aborderons pas les ensembles avant un certain temps, par exemple, mais il existe un ordre naturel des ensembles basé sur les propriétés des sous-ensembles, et Sage comparera les ensembles sur cette base. Prenons par exemple les ensembles {2, 3}, {2, 4} et {2, 3, 4, 5} :

```
sage : {2,3} < {2,3,4,5}
Vrai
sage : {2,3,4,5} > {2,3}
Vrai
sage : {2,3} < {2,4}
FAUX
```

Valeurs exactes et approximatives. Sage propose des symboles pour représenter les valeurs exactes de certains nombres importants ; ils devraient être faciles à mémoriser.

pi	$\pi$ , le rapport entre la circonférence d'un cercle et son rayon
et	$1 - \frac{1}{x}$ , ou, $\text{et} = \lim_{x \rightarrow \infty} 1 + \frac{1}{x}$ , où, la valeur de $a$ telle que la dérivée de $x$ soit $a$ , le $x$
je	nombre « imaginaire » ( $i \infty, \text{infini}^2 = -1$ )
oo ou +Infini	(non borné ci-dessus)
-oo ou -Infini	$-\infty$ (illimité ci-dessous)

TABLEAU 3. Symboles Sage pour les constantes importantes

Nous mettons ces identifiants en italique dans le texte, à la fois pour faciliter la lisibilité et pour souligner qu'ils doivent représenter des valeurs fixes.<sup>3</sup> Sage ne les met pas en italien dans la sortie.

Nous avons déjà expliqué que la force d'un système de calcul formel comme Sage réside dans sa capacité à manipuler des valeurs exactes plutôt que des valeurs approximatives. Néanmoins, il est parfois nécessaire de calculer avec des valeurs approximatives, et Sage le permet également.

Pour utiliser l'arithmétique approximative, saisissez au moins un nombre sous forme décimale !

Pour voir comment cela fonctionne, considérez les commandes suivantes et leurs résultats.

```
sage : 2/3
2/3
sage : 2./3
0.6666666666666667
```

Dans le premier cas, vous avez saisi des valeurs exactes ; Sage vous donne donc le quotient exact de la division de 2 par 3, soit la fraction  $2/3$ . Dans le second cas, vous spécifiez au moins une décimale, ce qui vous donne le quotient approximatif de la division de 2 par 3. Cela peut ressembler à une virgule flottante, mais ce n'est pas tout à fait le cas ; il s'agit en fait d'un objet que Sage appelle un RealNumber. Ne vous souciez pas de ces détails pour l'instant.

<sup>2</sup>Il est également possible d'utiliser  $i$  pour le nombre imaginaire, mais comme il est courant d'utiliser  $i$  comme variable, nous essayons d'éviter cela.

<sup>3</sup>Comme nous le notons à la page 29, Sage n'a pas de véritables constantes, donc un utilisateur ou un programme peut modifier la signification des symboles.

## Constantes, variables et indéterminées

Une subtile distinction d'usage. Les symboles mathématiques  $e$ ,  $\pi$  et  $i$  représentent des constantes, ce qui signifie que leurs valeurs sont définies et fixes. Si quelqu'un écrit l'équation d'Euler,

$$\text{de } e^{i\pi} + 1 = 0,$$

Ainsi, tout mathématicien averti connaîtra les valeurs de  $e$ ,  $i$  et  $\pi$ , même s'il découvre l'équation. Grâce aux symboles décrits précédemment, cela est également vrai dans Sage :

```
dire : e**(I*pi) + 1 == 0
```

Bien sûr, il arrive que l'un de ces symboles ait une autre signification. Par exemple,  $x_i$  désigne le  $i$ ème nombre d'une liste, et non une manipulation de  $x$  par le nombre imaginaire.

En mathématiques, d'autres symboles d'une seule lettre représentent deux types de variables, ce qui signifie que leurs valeurs ne sont pas nécessairement fixes dans un problème. Le symbole  $x$ , par exemple, peut représenter une infinité de valeurs. Les mathématiciens travaillent généralement avec deux types de variables ; dans le langage courant, on a tendance à désigner les deux types de variables par le terme « variable ». Il existe en réalité une distinction importante, visible dans une expression aussi simple que le polynôme.

$$c^{2x^2} - .$$

- Dans de nombreuses situations, les valeurs spécifiques de  $x$  et  $c$  sont très importantes. L'équation

$$c^{2x^2} - = 2$$

n'est pas toujours vrai ; cela dépend des valeurs de  $x$  et  $c$ , et dans la pratique, nous essayons souvent de trouver des valeurs pour lesquelles cela est vrai.

- Dans d'autres situations, cependant,  $x$  et  $c$  représentent des valeurs arbitraires. L'équation

$$c^{2x^2} - = (x + c)(x - c)$$

est vrai quelles que soient les valeurs de  $x$  et  $c$ .

Ceci est très important pour les systèmes de calcul formel, car un symbole peut également désigner une valeur spécifique ou indéterminée, et il est important de tenir compte de cette distinction de temps à autre. Nous conviendrons de la convention suivante :

- Lorsqu'une valeur spécifique est attribuée à un symbole, nous l'appelons généralement une variable, car nous pouvons modifier cette valeur à tout moment. •  
Lorsque nous n'avons pas l'intention de modifier la valeur d'une variable, nous l'appelons une constante.
- Lorsqu'un symbole ne doit pas contenir de valeur spécifique, mais est purement symbolique pour une valeur arbitraire.  
valeur d'un ensemble, nous l'appelons indéterminée.

Sage ne fournit qu'une seule variable indéterminée au démarrage :  $x$ . Si vous souhaitez utiliser une autre variable indéterminée, telle que  $y$ , vous devez la créer.

Sage n'offre pas de moyen de définir vos propres constantes, comme le font certains langages de programmation. faire. Dans Sage, une constante est simplement une variable que vous essayez vraiment de ne pas modifier.

Réinitialisation d'une variable ou d'une valeur indéterminée. Sage ne permettant pas de définir de vraies constantes, vous pouvez, si vous le souhaitez, réaffecter la valeur de `I` à une autre valeur, et il est fort probable que vous le fassiez un jour. Si cela se produit, la commande `reset()` permet de corriger facilement ce problème . Il suffit d'inclure le symbole entre guillemets.

```
sage : I^2 + 1 0

sage : I = 2 sage :
I^2 + 1 5

sage : réinitialiser('I') sage :
I^2 + 1 0
```

Création de variables et d'indéterminées. Pour créer une variable, utilisez la construction d'affectation.

Identifiant = expression,

où « identifiant » est le nom légitime d'un identifiant de symbole et « expression » est une expression mathématique légitime. Par exemple :

```
sage : sqrt2 = sqrt(2)
```

Affecte la valeur 2 au symbole `sqrt2`. Après cette affectation, vous pouvez utiliser le symbole `sqrt2` dans n'importe quelle expression ; Sage le considérera comme 2. Par exemple :

```
sage : sqrt2**2 2
```

Contrairement aux langages de programmation qui ne permettent d'affecter des variables qu'à une seule variable à la fois, Sage hérite de la flexibilité d'affectation de Python, qui permet d'affecter des variables à plusieurs d'un coup. Par exemple :

```
sage : sqrt2, sqrt3 = sqrt(2), sqrt(3) sage : sqrt2**2 2
```

```
sage : sqrt3**4
9
```

La première ligne de cette séquence d'instructions attribue les valeurs 2 et 3 aux variables `sqrt2` et `sqrt3`, dans cet ordre. Les deux instructions suivantes montrent que les affectations étaient correctes. De ce fait, vous pouvez probablement constater que l'instruction

```
sage : sqrt2, sqrt3 = sqrt3, sqrt2
```

a pour effet d'échanger les valeurs de `sqrt2` et `sqrt3`.

Pour créer une variable indéterminée, Sage propose la commande `var()`. Saisissez le nom de la variable indéterminée entre parenthèses, entre guillemets. Vous pouvez également créer plusieurs variables indéterminées de ce type en les listant entre guillemets ; laissez simplement un espace entre chaque nom. En cas de succès, Sage affichera les noms des nouvelles variables indéterminées entre parenthèses.

Par exemple:

```
sage : var('y z') (y, z)
```

Vous pouvez ensuite manipuler ces variables à votre guise.

```
sage : (y + z) + (z - y) 2*z
```

Identifiants valides. Chaque variable ou indéterminé nécessite un identifiant valide. Sage accepte les noms d'identifiants utilisant les séquences de caractères suivantes :

- Le nom

- doit commencer par une lettre (majuscule ou minuscule) ou le trait de soulignement (`_`). • Le nom peut contenir n'importe quelle combinaison de lettres, de chiffres ou le trait de soulignement.
- Le nom ne peut pas être un mot réservé, également appelé mot-clé. Vous rencontrerez ces cas tout au long du cours, mais il est peu probable que vous les choisissiez.

Contrairement aux conventions mathématiques, les noms des variables et des indéterminées peuvent comporter plusieurs symboles ; on le constate déjà avec `pi`, proposé dès le départ par Sage. Dans de nombreux cas, un nom comportant plusieurs symboles est bien plus facile à comprendre qu'un nom composé d'un seul symbole ; comparez, par exemple, d à dérivée. En revanche, un nom trop long et répétitif devient fastidieux à saisir et peut même compliquer la compréhension d'un programme. Une personne ayant des connaissances en mathématiques, par exemple, comprendra aussi bien `ddx` que dérivée, et préférera peut-être le premier à la seconde. En général, nous recommandons un nom ne dépassant pas six caractères. Cependant, si un nom plus long est utile pour plus de clarté, vous pouvez ignorer cette règle, et vous devriez le faire, tout comme nous.

### Expressions et commandes pour les manipuler

Une expression mathématique est constituée de toute combinaison significative de symboles mathématiques. Une équation est une telle expression, et dans Sage, vous pouvez affecter des équations comme valeurs de variables. Vous devrez certainement le faire de temps en temps, mais le signe égal a déjà une signification : il attribue la valeur d'une expression à un symbole pour créer une variable ! Pour désigner une équation, vous utilisez donc deux signes égal consécutifs :

```
sage : équation = x**2 - 3 == 1
```

Comme précédemment, si vous oubliez de doubler les signes égal, Sage vous donnera une erreur :

```
sage : équation = x**2 - 3 = 1
SyntaxError : impossible d'attribuer à l'opérateur
```

Nous rappelons au lecteur que si vous voyez ce message d'erreur, le problème est presque certainement dû à l'utilisation d'un seul signe égal alors que vous en avez besoin de deux.

Il est souvent utile de réécrire des expressions de différentes manières, et un système de calcul formel n'est, en substance, rien de plus qu'un outil sophistiqué de réécriture d'expressions. Voici quelques commandes utiles pour réécrire des expressions.

factoriser(exp)	factoriser l'expression exp simplifier(exp)
simplifier un peu l'expression exp développer(exp)	effectuer une
multiplication et d'autres opérations sur l'expression exp arrondir(exp,n)	arrondir l'expression exp à n décimales

TABLEAU 4. Quelques commandes pour manipuler des expressions (il y en a beaucoup plus)

Quelques conseils. Le symbole mathématique traditionnel de la multiplication est le simple  $\times$  ou le point  $\cdot$ . Malheureusement, les claviers d'ordinateur ne proposent aucun de ces symboles par défaut ; il faut généralement utiliser une solution de contournement pour saisir ce symbole. La solution traditionnelle consiste à saisir l'astérisque, et c'est ce qu'utilise Sage.

Il est courant en mathématiques d'omettre le symbole de multiplication :  $2x$ , par exemple, ou  $ab cd$ . Vous ne pouvez pas faire cela dans Sage ; cela vous donnera diverses erreurs :

```
sage: 2x
SyntaxError : syntaxe non valide sage :
var('abc d') (a, b, c, d) sage :
abcd

NameError : le nom « abcd » n'est pas défini
```

Dans les deux cas, Sage pense que vous essayez de saisir le nom d'un identifiant, nom qu'il ne reconnaît pas : • Pour  $2x$ , vous ne pouvez pas

commencer un nom d'identifiant par un nombre, l'erreur est donc dans la syntaxe. • Pour  $abcd$ , Sage n'a aucun moyen pratique de savoir que vous voulez dire les produits de  $a$ ,  $b$ ,  $c$  et  $d$ , plutôt qu'un identifiant différent nommé  $abcd$ , donc l'erreur est en fait une erreur de nom. (Rappelez-vous que dans Sage, contrairement à la plupart des mathématiques, les variables et les indéterminées peuvent avoir des noms plus longs que d'un symbole.)

Ces deux expressions fonctionnent bien si vous tapez le symbole de multiplication là où vous le souhaitez.<sup>4</sup>

Fonctions transcendantes. Sage offre tout ce qu'une calculatrice scientifique peut offrir. Voici quelques fonctions transcendantes courantes qui vous seront utiles tout au long de ce manuel et dans votre programme de mathématiques :

---

<sup>4</sup>Dans certaines versions de Sage, il existe un moyen de faire fonctionner la multiplication sans écrire explicitement le symbole de multiplication dans ces circonstances, mais il n'est pas disponible par défaut, vous devrez donc exécuter une commande spéciale pour cela. C'est une mauvaise idée pour les débutants, et vous devrez quand même taper un espace entre les symboles dans tous les cas, nous ne décrirons donc pas cette technique.

cos(a), tan(a), évaluées en a cot(a), arccos(a), arctan(a), les fonctions	les fonctions trigonométriques, sin(a), sec(a), csc(a) arcsin(a),
trigonométriques inverses, arccot(a), arcsec(a), arccsc(a) évaluées en a a (synonyme de $e^{**}a$ ) le logarithme népérien de a le logarithme en base b de a	
exp(a)	et
ln(a), log(a) log_b(a, b ) ou log(a,b )	

TABLEAU 5. Commandes Sage pour les fonctions transcendantes courantes

Nous pouvons également calculer les fonctions trigonométriques hyperboliques et leurs inverses en ajoutant h à la fin du nom usuel et avant la parenthèse gauche.

Vous avez peut-être remarqué que log(a) est synonyme de ln(a). Ce n'est pas la pratique courante dans les manuels scolaires américains, où log(a) est synonyme de log10a . Pour calculer ce que les Américains appellent le logarithme « commun », nous devons utiliser la commande log\_b() :

```
sage : log(100) log(100)
sage :
log_b(100,10) 2
```

Soyez prudent lorsque vous utilisez ces fonctions dans des expressions plus longues. En mathématiques, on écrit souvent  $\sin^2 \pi/4$  pour signifier  $(\sin(\pi/4))^2$  . Sage ne fait pas cette distinction. Il n'y a qu'une seule façon de le faire correctement : écrire ce que l'on veut dire :

```
dire : sin^2(pi/4)
TypeError : l'objet « sage.rings.integer.Integer » n'est pas appellable sage : (sin^2)(pi/4)

TypeError : type(s) d'opérande non pris en charge pour ** ou pow() : « Function_sin » et « int »

dire : (sin(pi/4))^2 1/2
```

Lorsque vous voyez une erreur TypeError avec ces messages, il y a fort à parier que vous avez fait une erreur

de frappe.5 • Le premier type d' erreur TypeError devrait être facile à déboguer : recherchez un endroit où un nombre est immédiatement suivi d'une parenthèse ouvrante — dans notre cas, 2(pi/4). Lorsque cela se produit, Sage pense que vous essayez d'utiliser 2 comme une fonction. La façon dont Sage évalue les expressions, il voit sin2(pi/4) et, même s'il peut nous sembler illogique de considérer 2 ( pi/4) comme une fonction 2 évaluée au point pi/4, c'est ainsi que Sage la voit. • Le deuxième type d' erreur TypeError peut être plus difficile. En général, le problème est que vous essayez d'effectuer une opération avec deux objets pour lesquels Sage ne sait pas comment procéder.

---

Techniquement, Sage fait référence au « type » d'objet, et non à ce que vous avez « tapé », mais nous sommes suffisamment désespérés pour accepter le jeu de mots.

Effectuez l'opération. Neuf fois sur dix, vous avez fait une erreur de frappe ; réexaminez donc votre saisie. Dans ce cas, vous avez utilisé une abréviation pratique, que Sage ne comprend pas (et pour cause).

Fonctions mathématiques et substitution. Un aspect utile des indéterminées est la possibilité de substituer des valeurs. Contrairement à l'affectation d'une valeur à une variable, les indéterminées ne conservent pas cette valeur après le calcul. Nous utilisons trois méthodes de substitution dans Sage.

La première méthode consiste à utiliser la méthode `subs()`, qui est l'abréviation de la méthode `substitute()`. Rappelons qu'une méthode est une commande spécifique à un objet. Vous accédez à cette fonctionnalité de la manière suivante :

- Tapez le nom de l'expression, puis un point, puis `subs()` • Après la parenthèse, répertoriez les affectations. Il y a deux façons de procéder : – Répertoriez chaque affectation sous forme d'équation : indéterminé=valeur. 6 –
- Répertoriez chaque affectation sous forme de « dictionnaire ». Pour ce faire, ouvrez une paire d'accolades, répertoriez les affectations sous la forme indéterminé:valeur, en séparant chaque affectation par une virgule,
- puis fermez les accolades. • Fermez maintenant les parenthèses qui ont commencé après `subs` : `subs(affectations)`.

Voici un exemple :

```
sage : f = x**2 sage :
f.subs(x=2)
4
sage : f.subs({x:2}) 4
```

La première substitution illustre l'affectation par équations ; la seconde illustre l'affectation par dictionnaires. Cette dernière approche est la plus fiable, sans erreur ni avertissement, pour effectuer une substitution dans toute expression mathématique contenant des indéterminées.

Une deuxième façon est de remplacer sans spécifier le nom de la méthode !

```
sage : f(x=2) 4
sage : f({x:2}) 4
```

N'utilisez pas cette approche sans préciser le nom de l'indéterminé. Si vous n'avez qu'un seul indéterminé, comme pour `x` ci-dessus, vous risquez d'oublier de le nommer. Dans ce cas, Sage émettra un avertissement :

---

Cela ne fonctionne pas toujours. Nous y reviendrons plus tard.

<sup>7</sup>En particulier, la deuxième approche fonctionne même à l'intérieur des fonctions définies par l'utilisateur, également appelées procédure, lorsque vous Je souhaite passer une valeur indéterminée comme argument à la procédure. Nous en parlerons plus tard.

```
sage : f(2)
```

Avertissement : la substitution utilisant la syntaxe d'appel de fonction et les arguments sans nom est obsolète et sera supprimée d'une future version de Sage ; vous pouvez utiliser des arguments nommés à la place, comme EXPR(x=..., y=...) 4

Cet avertissement peut ne pas apparaître sur la dernière ligne. Ce n'est pas une erreur, et Sage parvient à deviner la bonne substitution et affiche la bonne réponse après l'avertissement. De plus, l'avantage des avertissements de dépréciation est qu'ils n'apparaissent qu'une seule fois, même si vous répétez la même erreur. Néanmoins, leur présence est un peu inquiétante, et il est fort possible que la menace de suppression de cette fonctionnalité de devinette se réalise. Essayez donc d'éviter cette erreur. Cela peut poser problème si vous fournissez trop de valeurs ; Sage ne saura pas quoi attribuer et se plaindra :

```
sage : f(3,2,1)
```

ValueError : le nombre d'arguments doit être inférieur ou égal à 1

Cela ne se produira pas lorsque vous spécifiez des affectations, car Sage sait ce qui va où :

```
sage : f = x^2 - y^2 sage :
```

```
f(x=3,w=2,z=1) -y^2 + 9
```

La troisième méthode de substitution est utile lorsque vous utilisez une expression mathématique appelée fonction. Rappelons qu'une fonction associe les éléments d'un ensemble, appelé domaine, aux éléments d'un autre ensemble, appelé plage, de telle sorte que chaque entrée possède une sortie bien définie. Définir et utiliser des fonctions est simple dans Sage :

```
sage : f(x) = x**2 sage : f(2)
```

Les fonctions ont également la propriété utile de définir toute valeur indéterminée figurant entre les parenthèses de leur définition. Vous pouvez ensuite utiliser ces valeurs indéterminées dans d'autres contextes sans utiliser au préalable la commande var() .

---

<sup>8</sup>L'avertissement est apparu depuis quelques années déjà.

<code>lim( f ,x=a)</code> ou <code>limite( f ,x=a)</code>	calculer la limite bilatérale de $f(x)$ à $x = a$
<code>lim( f ,x=a, dir=direction)</code> ou <code>limit( f ,x=a, dir=direction)</code> <code>diff( f ,x)</code> ou	calculer la limite unilatérale de $f(x)$ à $x = a$ , avec direction « gauche » ou « droite »
<code>derived( f ,x)</code> <code>diff( f ,x,m)</code> ou	calculer la dérivée de $f(x)$ par rapport à $x$
<code>derived( f ,x,m)</code> <code>integral( f ,x)</code> ou <code>integrate( f ,x)</code>	calculer la dérivée $m$ -ième de $f(x)$ par rapport à $x$
<code>integral( f ,x,a,b )</code> ou <code>integrate( f ,x,a,b )</code>	calculer l'intégrale indéfinie (primitive) de $f(x)$ par rapport à $x$
	calculer l'intégrale définie sur $[a, b]$ de $f(x)$ par rapport à $x$

TABLEAU 6. Commandes pour le calcul exact des limites, des dérivées et des intégrales

```
sage : f(w,z) = 4*w**2 - 4*z**2
sage : f(3,2)
20
sage : facteur(w**2 - z**2)
(entrée + sortie)*(entrée - sortie)
```

Notez que nous avons pu travailler directement avec  $w$  et  $z$ , même si nous ne les avons pas définis.

### Votre calcul de base

Nous passons maintenant à la question des offres de Sage pour le Calculus.

Calcul exact. Commençons par le calcul exact. Sage propose trois commandes

Effectuez le calcul exact pour le calcul intégral. Vous pouvez voir ces commandes dans le tableau 6, avec les deux synonymes et différentes options d'utilisation :

- Pour les limites, nous avons les synonymes `lim()` et `limit()`, que nous pouvons utiliser pour calculer soit Limites bilatérales (par défaut) ou unilatérales (spécifiez une direction). Vous devez spécifier les deux. l'indéterminé ( $x$ ) et la valeur ( $a$ ).
- Pour la différentiation, nous avons les synonymes `diff()` et `derivative()`, que nous pouvons utiliser soit pour différencier une fois, soit, en spécifiant un argument optionnel, pour différencier plusieurs fois.
- Pour l'intégration, nous avons les synonymes `integral()` et `integrate()`, que nous pouvons utiliser pour calculer soit l'intégrale indéfinie, également appelée primitive, soit si nous spécifions les limites de l'intégration, pour calculer l'intégrale définie, que vous apprenez probablement en premier comme l'aire sous une fonction.

Jetons un coup d'œil rapide à leur fonctionnement.

Les limites. Tout d'abord, les limites. En gros,

- $\lim_{x \rightarrow a^+} f(x)$  est la valeur y approchée par  $f$  lorsque  $x$  s'approche de  $a$  depuis la droite (c'est-à-dire depuis nombres plus grands que  $a$ );
- $\lim_{x \rightarrow a^-} f(x)$  est la valeur y approchée par  $f$  lorsque  $x$  s'approche de  $a$  depuis la gauche ; et

- $\lim_{x \rightarrow a} f(x)$  est la valeur  $y$  approchée par  $f$  lorsque  $x$  approche  $a$  des deux côtés.

Si une fonction est continue en  $a$ , on peut trouver la limite par substitution, ce qui est ennuyeux.

```
sage : f(x) = x^2 + 1 sage :
limite(f(x), x=1) 2

sage : f(1) 2
```

Les limites sont beaucoup plus intéressantes lorsque la fonction est discontinue en  $a$ . Voici quelques problèmes de calcul à retenir :

```
sage : limite((x**2 - 1)/(x - 1), x=1)
2

dire : limite(x/abs(x), x=0) et

sage : limite(x/abs(x), x=0, dir='gauche')
-1

sage : limite(1/x, x=0, dir='gauche')
-Infinity sage :
limite(sin(1/x), x=0) ind
```

Que signifient ces réponses ?

- Le premier exemple montre comment Sage détecte et contourne automatiquement la division par zéro, lorsque cela est possible.
- Le deuxième exemple montre ce qui peut mal se passer lorsqu'il n'y a aucun moyen de contourner cela : und est l'abréviation de « la limite est indéfinie ». Dans ce cas, il existe des limites unilatérales de gauche et de droite, et ce sont des limites finies, mais elles ne concordent pas entre elles.
- Les troisième et quatrième exemples montrent comment calculer des limites unilatérales dans le cas où les limites bilatérales n'existent pas.
- Dans certains cas, la limite n'existe pas car la fonction n'approche aucune valeur particulière. Lorsque cela se produit, mais que la fonction reste finie, Sage répond par ind, qui signifie « la limite est indéfinie, mais bornée » (c'est-à-dire non infinie). Nous le voyons dans le cinquième exemple. Si, en revanche, la fonction oscille entre des infinis, Sage répond par und.

En parlant de  $1/x$ , voici un résultat bilatéral auquel vous ne vous attendez peut-être pas :

```
sage : limite(1/x, x=0)
Infini
```

Si vous connaissez la bonne réponse, alors en voyant cela, vous pourriez être tenté de

**PANIQUE!**

... mais vous ne devriez pas. Ne vous méprenez pas sur cette réponse. Sage ne prétend pas que le résultat est  $\infty$  ; il possède en fait un symbole distinct pour cela.

```
sage : limite(1/x, x=0, dir='droite')
+Infini
```

Notez qu'il s'agit d'une infinité signée, alors que le résultat précédent était non signé. L'infini non signé indique que « la limite de la valeur absolue de l'expression est l'infini positif, mais la limite de l'expression elle-même n'est ni l'infini positif ni l'infini négatif ». [7]

Pour éviter cela, la meilleure solution est probablement d'évaluer manuellement vos limites de chaque côté ; dans ce cas, Sage affiche `+Infinity` ou `-Infinity`, selon le cas. Vous pouvez également vérifier si la limite produit `unsigned_infinity` :

```
sage : limite(1/x, x=0) == unsigned_infinity
Vrai
```

... mais dans ce cas, vous voudrez probablement vérifier les limites de gauche et de droite, de toute façon.

Dérivées. Rappelons que la dérivée de  $f(x)$  en  $x = a$  est

- la pente de la droite tangente à  $f(x)$  en  $x = a$  (si une telle droite existe) ; ou, de manière équivalente,
- la limite des pentes des droites sécantes reliant  $f$  en  $x = a$  et  $x = b$ , lorsque  $b \rightarrow a$  ; ou,
- de manière équivalente,
- la limite des pentes des droites sécantes reliant  $f$  en  $x = a$  et  $x = b$ , lorsque la distance entre  $a$  et  $b$  approche 0 ( $\Delta x \rightarrow 0$ ) ; ou, de manière équivalente,  $f(a + \Delta x) - f(a)$
- $\lim_{\Delta x \rightarrow 0} \frac{\Delta x}{\Delta x}$

Nous pouvons également parler de la dérivée comme d'une fonction ; c'est-à-dire

que  $f(x)$  est • la valeur de  $f(a)$  chaque fois que  $x = a$  ; ou, de  

$$(x + \Delta x) \frac{\text{manière équivalente, } f}{\Delta x \rightarrow 0 - f(x) + \lim_{\Delta x} \Delta x}$$
, que vous avez probablement passé beaucoup de temps à manipuler dans votre cours de calcul avec un  $h$  au lieu d'un  $\Delta x$ .

Les fonctions `diff()` et `derivative()` de Sage calculent la dérivée comme une fonction ; si vous souhaitez calculer la dérivée en un point, définissez une fonction et remplacez-la.

```
sage : diff(x**2, x) 2*x

sage : df(x) = diff(x**2, x) sage : df(1) 2

sage : diff(cos(x), x, 1042) -cos(x)
```

---

La citation suivante est tirée de la documentation sur Maxima, le sous-système utilisé par Sage pour évaluer les limites. Au moment de la rédaction de cet article, la documentation de Sage ne contient pas cette information et peut semer la confusion chez l'utilisateur en simplifiant (`Infinity == +infinity.full_simplify()` à `True`). Le problème est qu'`infinity` a une signification lorsqu'il apparaît dans la sortie de `limit()`, et une toute autre signification lorsqu'il est saisi en ligne de commande.

Ce dernier exemple nous a donné la dérivée 1042e de  $\cos x$ . Cela prendrait énormément de temps à faire à la main, à moins que vous ne remarquiez un modèle.

Intégrales. Le mot « intégrale » a deux significations différentes.

L'intégrale indéfinie de  $f(x)$  est sa primitive ; autrement dit,  $f(x)dx = F(x)$  où  $F$  est une fonction telle que  $F(x) = f(x)$ .

Il existe en réalité une infinité de telles primitives ; un résultat important de l'analyse est que deux d'entre elles ne diffèrent que par une constante. On résout généralement ce problème en calcul en ajoutant une « constante d'intégration » à l'intégrale ; par exemple,

$$\int \cos 2x dx = \sin 2x + C. \quad 2$$

Comme vous le verrez, Sage omet la constante d'intégration.<sup>10</sup>

L'intégrale définie de  $f(x)$  sur l'intervalle  $I$  est la limite des sommes pondérées sur  $n$  sous-intervalles de  $I$  lorsque  $n$  tend vers  $\infty$ . Plus précisément,

$$\int_a^b f(x) dx = \lim_{n \rightarrow \infty} \sum_{i=1}^n f(x_i) \Delta x, \text{ où } x_i \text{ est sur le } i\text{ème sous-intervalle de } I.$$

L'intervalle peut être fini —  $[a, b]$  — ou infini —  $[a, \infty)$  ou  $(-\infty, b]$  — à condition que l'intégrale converge et ne soit pas impropre. Si elle est impropre, il faut la décomposer en plusieurs parties ; par exemple,

$$\int_{-1}^1 \frac{1}{x^3} dx = \lim_{t \rightarrow 0^-} \int_{-1}^t \frac{1}{x^3} dx + \lim_{t \rightarrow 0^+} \int_t^1 \frac{1}{x^3} dx.$$

Sage, cependant, peut gérer de telles intégrales sans que vous ayez à les décomposer.

Vous avez vu dans le tableau que Sage vous permet de calculer des intégrales indéfinies et définies.

```
sage : intégrer(x**2, x)
1/3*x^3

sage : intégrer(x**2, x, 0, 1) 1/3 sage : intégrer(1/
x**2, x, -1, 1) 0

sage : intégrer(1/x, x, 1, infini)
ValueError : l'intégrale est divergente.
```

Cela semble simple : • le premier

nous donne  $x^2 dx$  ;  $x^2 dx$  ; • le

nous donne  $\int_0^1 \frac{1}{x^2} dx$  second

• le troisième nous donne  $\int_{-\infty}^1 \frac{1}{x^2} dx$ , ce qui est impropre en raison d'une asymptote à  $x = 0$ , bien que  $1/3$  Sage le gère facilement ; et • le quatrième

nous donne le  $\int_1^{\infty} \frac{1}{x^2} dx$ , qui diverge en fait ; il faut une puissance supérieure à 1 dénominateur pour converger.

Sage lui-même est conscient de cette dernière propriété ; nous le démontrons au cours d'une série d'étapes.

---

<sup>10</sup>Votre professeur de calcul serait consterné, absolument consterné.

```
sage : était('p')

p sage : intégrer(1/x**p, x, 1, infini)
ValueError : le calcul a échoué car Maxima a demandé des contraintes supplémentaires ; l'utilisation de la
commande « assume » avant l'évaluation *peut* aider (un exemple de syntaxe légale est « assume(q>0) »,
voir « assume ? » pour plus de détails)

Est-ce que p est positif, négatif ou nul ?
```

Oups ! C'est une requête parfaitement logique. Le message d'erreur est également utile : il introduit une nouvelle commande, la commande `assume()`. Nous ne l'utiliserons pas souvent, mais c'est un cas où elle s'avère utile. Supposons que  $p > 1$  :

```
sage : supposer(p > 1) sage :
intégrer(1/x**p, x, 1, infini) 1/(p - 1)
```

Sage affirme que

$$\int_1^{\infty} \frac{1}{x^p} dx = p \frac{1}{-1 + p}$$

Lorsque  $p > 1$ , un fait que vous devriez pouvoir vérifier manuellement. Si nous supposons que  $p$  est inférieur à 1, nous rencontrons deux problèmes. Le premier est peut-être inattendu :

```
sage : supposer(p <= 1)
ValueError : l'hypothèse est incohérente
```

Si vous voulez vraiment changer cela, vous pouvez utiliser la commande `forget()` pour oublier tout ce que vous avez supposé().<sup>11</sup> Nous voulons vraiment changer cela, donc,

```
sage : oublier() sage :
supposer(p <= 1) sage : intégrer(1/
x**p, x, 1, infini)
ValueError : le calcul a échoué car Maxima a demandé des contraintes supplémentaires ; l'utilisation de la
commande « assume » avant l'évaluation *peut* aider (un exemple de syntaxe légale est « assume(p>0) »,
voir « assume ? » pour plus de détails)

Est-ce que p est positif, négatif ou nul ?
```

Cela peut vous surprendre, mais pas pour longtemps : si  $p \leq 0$ , alors nous regardons  $x^q$  avec  $q \geq 0$ , alors que  $p > 0$  est ce que nous avions en tête. Puisque Sage ne peut pas lire dans nos pensées, ajoutons l'hypothèse suivante :

---

Vous pouvez également oublier() seulement une partie de vos hypothèses. Ainsi, outre l'utilisation de `forget()` comme nous l'avons montré, vous pouvez taper `forget(p <= 1)`. Cela est particulièrement utile si plusieurs hypothèses sont en jeu et que vous souhaitez n'en oublier qu'une.

```
sage : supposer(p>0)
sage : intégrer(1/x**p, x, 1, infini)
ValueError : l'intégrale est divergente.
```

Bien qu'il signale une erreur, nous devons considérer cette erreur comme un succès, car il vérifie ce que nous savions déjà.<sup>12</sup>

(N'oubliez pas de forget() lorsque vous avez terminé de supposer().)

Intégration numérique. Les intégrales présentent une particularité que les dérivées n'ont pas : on ne peut pas toujours calculer une « forme élémentaire » d'une intégrale.<sup>13</sup> Considérons, par exemple,  $e^{2x} dx$ .

```
sage : intégrer(e**(x**2), x)
-1/2*I*carré(pi)*erf(I*x)
```

Vous n'avez probablement jamais vu  $\text{erf}(x)$ , et ce n'est pas grave. <sup>14</sup> Le problème est que ce n'est pas une fonction élémentaire. Si vous essayez d'obtenir de l'aide à ce sujet dans Sage, vous verrez ce qui suit :

```
sage : héritage ?
...
erf(x) = frac{2}{sqrt{pi}} int_0^x e^{-t^2} dt.
...
```

Autrement dit,

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Il n'y a donc aucun moyen de simplifier davantage cela.

L'exemple suivant apparaît lorsque vous essayez de calculer la longueur de l'arc d'une ellipse centrée à l'origine, avec un axe horizontal de longueur 2 et un axe vertical de longueur 1.

```
sage : f(x) = sqrt(1 - x^2/4) sage :
df(x) = diff(f, x) sage :
intégrer(sqrt(1+(df(x)**2), x, -2, 2) intégrer(sqrt(-1/4*x^2/
(x^2 - 4) + 1), x, -2, 2)
```

---

<sup>12</sup> Il peut paraître surprenant que Sage nous demande de supposer que  $p > 0$  alors qu'en réalité, l'intégrale diverge également pour  $p \leq 0$ . D'un point de vue mathématique, la seule hypothèse nécessaire est  $p \leq 1$ . Cependant, l'intégration de tous les cas particuliers peut s'avérer trop complexe à mettre en œuvre en informatique ; l'utilisateur doit donc souvent résoudre certains de ces problèmes par lui-même.

<sup>13</sup> La définition précise d'une « forme élémentaire » dépasse le cadre de ce texte, mais vous pouvez essentiellement la considérer comme n'importe quelle combinaison algébrique des fonctions que vous avez étudiées en précalcul, y compris les fonctions trigonométriques, exponentielles, logarithmiques et hyperboliques.

Pour ceux qui sont intéressés, il s'agit de la fonction d'erreur gaussienne, et vous la verrez probablement dans un cours de probabilités.

Dans ce cas, la réponse de Sage à l'intégrale n'est qu'une autre intégrale. Cela ne semble pas particulièrement utile, mais Sage ne peut pas faire grand-chose.<sup>15</sup> L'intégrale ne se réduit pas à l'élémentaire.

termes.

Dans certains cas, une intégrale se réduit à des termes élémentaires, mais le résultat est tellement compliqué qu'il est préférable de ne pas l'utiliser. Voici un exemple.

```
sage : intégrer(x^10*cos(x), x) 10*(x^9 -  
72*x^7 + 3024*x^5 - 60480*x^3 + 362880*x)*cos(x) + (x^10 - 90*x^8 + 5040*x^6 - 151200*x^4  
+ 1814400*x^2 - 3628800)*sin(x)
```

... et ce n'est même pas si mal.

Dans de nombreux cas, on souhaite simplement obtenir une valeur numérique pour l'intégrale. On peut vouloir l'aire, une accumulation de valeurs ou une autre application. Dans ce cas, inutile de recourir à l'intégrale indéfinie ; on peut approximer l'intégrale définie grâce à une technique d'intégration numérique. La commande de Sage pour cela est « digital\_integral() ».

intégrale_numérique( f ,a,b )	estimation $\int_a^b f(x) dx$
intégrale_numérique( f ,a,b , max_points=n) estimation de n points	$\int_a^b f(x) dx$ n'utilisant plus

Son utilisation nécessite une petite explication, mais avant cela, considérons un exemple.

```
sage : intégrale_numérique(sqrt(1+(df(x))**2), -2, 2) (4.844224058045445,  
4.5253950830572916e-06)
```

Vous remarquerez immédiatement plusieurs différences. •

Nous ne spécifions pas la variable d'intégration pour la fonction intégrale\_numérique(). •

Le résultat est composé de deux nombres. Le second est en notation scientifique.

qui dans ce cas, vous devriez vous rappeler qu'il s'agit d'environ  $4,525 \times 10^{-6}$  •

La réponse que nous obtenons ici est une paire ordonnée. La première valeur est l'approximation de l'intégrale par Sage ; la seconde est une estimation de l'erreur. Autrement dit, notre réponse est certainement correcte jusqu'à la quatrième décimale, bien que l'arrondi imposé par la sixième décimale (où l'erreur peut se produire) crée une ambiguïté à la cinquième. Nous pouvons dire que l'intégrale se situe dans l'intervalle (4,8442195, 4,8442286) :

```
sage : A, err = intégrale_numérique(sqrt(1+(df(x))**2), -2, 2) sage : A - err, A + err  
(4.844219532650363,  
4.844228583440528)
```

---

Au moins un autre système de calcul formel propose une réponse similaire à EllipticE(...), ce qui semble prometteur jusqu'à ce que vous lisiez sa documentation. Comme erf(...), il ne fait que reformuler l'intégrale d'origine. Dans ce cas, le nom indique qu'il s'agit d'une intégrale elliptique.

## Structures mathématiques dans Sage

C'est l'occasion idéale de vous présenter l'une des fonctionnalités les plus puissantes d'un système de calcul formel : sa capacité à fonctionner dans différents contextes mathématiques. Dans ce cours, nous se concentrera particulièrement sur les anneaux et les champs :

Un anneau : est un ensemble où l'addition et la multiplication se comportent selon les propriétés que vous attendre:

Fermeture : L'addition ou la multiplication de deux éléments de l'anneau donne un autre élément de l'anneau.  
anneau.

Associatif : Le résultat de l'addition ou de la multiplication de trois éléments ne dépend pas  
lesquels additionnez-vous ou multipliez-vous en premier :  $a + (b + c) = (a + b) + c$  et  $a(bc) = (ab)c$ .

Identités : Vous pouvez trouver deux éléments « 0 » et « 1 », pour lesquels l'ajout de « 0 » ne change rien  
élément de l'anneau, et multiplier par « 1 » ne change aucun élément de l'anneau :  $a + 0 = a = 0 + a$  et  $a \times 1 = a = 1 \times a$ .

Distributif : Vous pouvez distribuer la multiplication sur l'addition :  $a(b + c) = ab + ac$ .

L'addition bénéficie de deux propriétés supplémentaires :

Addition commutative : Le résultat de l'addition de deux éléments de l'anneau ne dépend pas  
selon leur ordre dans la somme :  $a + b = b + a$ .

Inverses additives : Vous pouvez trouver le « contraire » de n'importe quel élément, de sorte qu'en les additionnant  
renvoie « 0 » :  $d + b = 0 = b + d$ . On note généralement ce contraire comme une valeur négative, donc  
que  $a + (-a) = 0 = (-a) + a$ .  
<sup>16</sup>

Notez qu'un anneau peut ne pas avoir de multiplication commutative ni d'inverses multiplicatives. Omettre cela permet d'organiser des ensembles de matrices sous forme d'anneaux, car les matrices satisfont aux propriétés énumérées.  
ci-dessus, mais ne satisfont pas les deux autres.

Un corps : est un anneau commutatif où l'on peut également « diviser », ce qui signifie que l'on peut trouver un « inverse multiplicatif » pour chaque élément non nul, c'est-à-dire pour chaque non nul  $a$ , vous pouvez trouver  $b$  dans le même anneau tel que  $ab = 1$ . L'analogie avec la division est seulement cela : une analogie ; nous n'utilisons généralement pas le symbole de division en dehors des systèmes vous êtes habitués ; nous les écrivons avec un exposant de  $-1$  : plutôt que  $a/b$ , nous écrivons  $a \cdot b^{-1}$ .

Les suspects habituels. Vous avez déjà passé beaucoup de temps à travailler dans les champs et les arènes,  
On ne vous l'a probablement pas dit. Sage vous propose directement plusieurs anneaux et champs :

Ensemble	Traditionnel symbole de structure mathématique		Symbole du sage
Entiers		anneau	ZZ
Nombres rationnels (fractions entières)		champ	QQ
Nombres réels		champ	RR (mais voir ci-dessous)
nombres complexes		champ	CC (mais voir ci-dessous)

TABLEAU 7. Anneaux et champs communs dans Sage

<sup>16</sup>Vous pouvez également considérer cette propriété comme une « fermeture sous soustraction », mais les gens n'en parlent généralement pas de cette façon. termes.

Lorsque vous saisissez une valeur, Sage fait une estimation éclairée du type d'objet que vous souhaitez.

Vous pouvez le trouver en utilisant une instruction type() :

```
sage : a = 1 sage :
type(a) <type
'sage.rings.integer.Integer' sage : b = 2/3 sage : type(b)
<type

'sage.rings.rational.Rational' sage : c = 2./3 sage : type(c)
<type

'sage.rings.real_mpfr.RealLiteral' sage : d = 1 + I sage : type(d)
<type

'sage.symbolic.expression.Expression'>
```

Sage ne semble pas signaler que le dernier est un nombre complexe, mais une expression symbolique.

Bien que  $1 + i$  soit en réalité une expression symbolique, ce qui suffira pour la plupart de nos besoins, nous pouvons forcer Sage à la considérer comme un élément de . Pour ce faire, nous saisissons CC, suivi de parenthèses, dans lesquelles nous saisissons le nombre complexe souhaité. L'inconvénient est que nous perdons la précision du calcul symbolique :

```
sage : d2 = CC(1+I) sage :
type(d2) <type
'sage.rings.complex_number.ComplexNumber' sage : d3 = d2 + (1 - I) sage : d3
2,000000000000000
```

Vous voyez ce qui s'est passé ? Au lieu d'obtenir la valeur exacte de 2, nous avons reçu une approximation de 2. Dans ce cas, l'approximation semble exacte, mais là n'est pas la question ; si vous travaillez dur, l'erreur commencera à s'infiltrer. Par exemple :

```
sauge : d3 - 1,99999999999
1,00008890058234e-12
```

Pourquoi cela s'est-il produit ? Pour effectuer des calculs efficaces dans les domaines réels et complexes, Sage a recours à des approximations des nombres concernés. Vous le constaterez immédiatement en observant d2 :

```
sauge : d2
1,000000000000000 + 1,000000000000000*I
```

Le processus même de conversion de d2 en nombre « complexe » implique une perte de précision. Il en sera de même si vous travaillez en RR. Parfois, c'est indispensable, mais il faut le garder à l'esprit. L'oublier peut conduire à des résultats déroutants, tels que :

```
sage : 2,99 - 2,9
0,09000000000000003
```

Oups !

Il est souvent utile de séparer les parties réelle et imaginaire d'un nombre complexe. Sage propose deux commandes utiles pour cela. Elles fonctionnent que le nombre complexe soit une expression symbolique ou qu'il soit dans CC.

partie_réelle(z)	partie réelle de z
image_part(z)	partie imaginaire de z

```
sage : partie_réelle(12-3*I) 12
sage : imag_part(CC(12-3*I))
-3,000000000000000
```

La norme d'un nombre complexe est également disponible.

norme(z)	norme de z
----------	------------

Si vous ne connaissez pas la norme d'un nombre complexe, sachez qu'elle est comparable à la valeur absolue d'un nombre réel, car elle donne une idée de sa taille. Plus précisément, la norme est

$$a + bi = \sqrt{a^2 + b^2}.$$

Si cela vous rappelle le théorème de Pythagore, c'est normal, car il est presque identique à la formule de distance euclidienne.

```
sage : norme(2+3*I) 13
```

Vous pouvez parfois convertir des nombres réels en nombres entiers, mais pas toujours.

```
sage : a = ZZ(1.0)
sage : un 1

sage : b = ZZ(1.2)
TypeError : tentative de conversion d'un nombre réel non entier en entier
```

Un autre symbole que vous rencontrerez de temps à autre est  $\mathbb{N}$ , l'ensemble des entiers non négatifs, également appelé nombre naturel. Il ne s'agit pas d'un anneau, car, par exemple,  $1 \in \mathbb{N}$  est  $-1 \notin \mathbb{N}$ .

Les suspects inhabituels. De nombreuses applications des mathématiques du dernier demi-siècle relèvent de l'arithmétique modulaire. Sans entrer dans les détails, l'arithmétique modulaire consiste à effectuer une opération, puis à en extraire le reste après division par un nombre fixe, appelé module. Un exemple très ancien de ce phénomène se produit lorsqu'on traite du temps :

```
sage : heure_courante = 8 sage :
heures_occupées = 20 sage :
temps_libre = heure_courante + heures_occupées sage : temps_libre 28
```

Le problème est évident : il n'existe pas de « 28e » heure. On peut trouver l'heure exacte en divisant par 12 et en prenant le reste :

```
sage : temps_libre = (heure_actuelle + heures_occupées) % 12 sage : temps_libre 4
```

Il est désormais évident que l'individu en question n'est pas libre avant 16 heures.

Demander explicitement l'arithmétique modulaire à chaque opération est fastidieux. Il s'avère que l'arithmétique modulaire permet d'obtenir un anneau parfaitement acceptable, ce qui nous permet d'ajouter une nouvelle ligne à notre tableau d'anneaux et de corps :

Ensemble	Structure mathématique traditionnelle	Symbol Sage	
Entiers modulo n ( $n > 1$ )	n	anneau	ZZ.quo(n)

TABLEAU 8. Anneaux et champs communs dans Sage

Voyons comment cela fonctionne en pratique. Nous allons tester notre problème précédent en travaillant dans  $\mathbb{Z}_{12}$ .

```
sage : Z_12 = ZZ.quo(12) sage :
heure_courante = Z_12(8) sage : heures_occupées
= Z_12(20) sage : temps_libre = heure_courante
+ heures_occupées sage : temps_libre 4
```

Les avantages de cette approche ne seront pas aussi évidents ici qu'en pratique, mais ils sont bien là : l'exponentiation par grands nombres, par exemple, est beaucoup plus rapide grâce à cette méthode. L'arithmétique modulaire étant un outil puissant, nous vous l'utiliserons régulièrement à partir de maintenant.

Les suspects les plus insolites : les structures en anneau que vous avez vues ici décrivent un « type » de données. Par exemple, Sage considère les éléments de  $\mathbb{Z}$  comme des types de sage.rings.integer.Integer, que nous pouvons abréger en Integer.

```
sage : type(Integer(3)) <type
'sage.rings.integer.Integer'>
```

Python, dans sa version la plus simple, ne comprend pas le type Integer de Sage , mais utilise un autre type pour représenter les entiers : int. Sage est suffisamment intelligent pour reconnaître lorsqu'un int et un Integer ont la même valeur, et la conversion entre les deux se fait sans trop de difficulté (du moins en usage « ordinaire »).

```
sage : type(int(3)) <type
'int'> sage :
int(3) == Entier(3)
Vrai
sage : type(int(3)) == type(Entier(3))
FAUX
sage : entier(int(entier(int(3)))) 3
```

Vous vous demandez peut-être pourquoi Sage implémente son propre type Integer . L'une des raisons est que les anciennes versions de Python imposaient une taille maximale aux entiers ; par exemple, les machines 64 bits ne fonctionnaient qu'avec les entiers  $-2^{64}, 2^{64} - 1$ . Les versions plus récentes de Python autorisent les objets de type int à n'importe quelle taille ; ce n'est donc plus si important, mais cela reste nécessaire. Les détails importent peu, mais sachez que cette représentation alternative des entiers existe.<sup>17</sup>

De la même manière, il existe au moins deux façons de représenter des approximations de nombres réels dans Sage. L'une consiste en des éléments de RR, qui sont à proprement parler de type sage.rings.real\_mpfr.RealLiteral. (Vous ne pouvez pas l'abréger facilement, mais vous pouvez utiliser RR dans les conversions.) Un deuxième est de type RealNumber. Sage hérite d'un troisième type de données, float, de Python. Ces trois types sont différents, mais la conversion entre eux fonctionne généralement sans problème.

```
sage : type(float(3)) <type
'float'> sage :
type(RR(3)) <type
'sage.rings.real_mpfr.RealLiteral'> sage : type(RR(3))
== type(3.0)
FAUX
sage : type(RR(3)) == type(float(3))
Faux
sage : RR(3) == float(3) == RealNumber(3)
Vrai
```

---

<sup>17</sup>En principe, Sage utilise des anneaux, et un élément d'un anneau est appelé Élément. Ceci étant spécifique à Sage, qui utilise Python mais n'est pas la même chose, un int Python ne peut pas être un Élément. En revanche, Sage peut « encapsuler » Python considère int comme un entier, mais pour des raisons plus complexes, Sage ne peut pas non plus le considérer comme tel . Quoi qu'il en soit, Sage implémente Integer grâce à un système appelé GMP .

## Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Bien que cela fonctionne pour l'exponentiation, vous devriez éviter d'utiliser le symbole  $\wedge$  dans ce contexte.  $2.6 // 4 == 1$  et  $6 \% 4 == 2$ . Bien que  $i$  soit une constante mathématique fixe et que Sage fournisse le symbole  $I$  pour la représenter, vous ne pouvez pas toujours compter sur l'équation  $I^{**2} == -1$  pour être vraie dans Sage.
4. Utilisez la commande `var()` pour créer de nouvelles variables. Vous créez de nouvelles variables indéterminées en... en leur signant des valeurs.
5. Si vous attribuez  $e = 7$  et que vous souhaitez ultérieurement restaurer  $e$  à sa valeur d'origine, de sorte que  $\ln(e) == 1$ , utilisez `réinitialiser('e')`.
6. Le nom `I337` satisfait aux exigences d'un identifiant Sage. (Le premier symbole est une minuscule `L`.)
7. Le nom `1337` satisfait aux exigences d'un identifiant Sage. (Le premier symbole est le numéro `1`.)
8. Dans Sage,  $\log(a) == \ln(a)$  pour toute valeur de  $a$ .
9. Avant de définir une fonction avec l'indéterminée  $w$ , vous devez définir l'indéterminée  $w$ .
- 10 Si une limite unilatérale () renvoie `+Infinity`, alors la limite réelle est  $\infty$ .
11. Si une limite bilatérale () renvoie `Infinity`, vous devez alors vérifier la limite des deux côtés.
12. Sage peut simplifier toutes les intégrales sous forme élémentaire à l'aide de la commande `integral()`.
13. Une intégrale indéfinie a une infinité de solutions, mais Sage n'en fournit qu'une seule.
14. Sage ne peut pas gérer les intégrales définies lorsqu'il y a une asymptote dans l'intervalle.
15.  $Z_{-12}(12) == 0$ , où  $Z_{-12}$  est défini comme ci-dessus.

Bonus : la réponse à toutes ces questions Vrai/Faux est « Faux ».

Choix multiple.

1. Si vous rencontrez une erreur `AttributeError` lorsque vous travaillez dans Sage, laquelle des actions suivantes n'est pas appropriée ?
  - A. Demandez à votre instructeur.
  - B. Regardez dans l'index pour voir si quelque chose de similaire est couvert quelque part dans le texte.
  - C. **PANIQUE !**
  - D. Lorsque tout le reste échoue, renseignez-vous sur le site Web `sage-support`.
2. Pour laquelle des expressions suivantes Sage semble-t-il renvoyer une valeur différente de zéro ?
  - A.  $e^{**(\text{i}*\pi)} + 1$
  - B.  $I^{**2} + 1$
  - C.  $(\cos(\pi/3) + \text{i} * \sin(\pi/3))^{**6} - 1$
  - D.  $\sin(\pi/4)^{**2} + \cos(\pi/4)^{**2} - 1$

ne propose-t-il pas, même si de nombreux langages de programmation

Les langues font-elles ?

- A. constantes B.
  - variables C.
  - identifiants D.
  - indéterminés
4. Lequel des noms d'identifiant suivants n'est probablement pas l'idée la plus judicieuse pour la ième valeur d'un dans une séquence ?
    - A. `a`
    - B. `a_curr`

- C.  
vous D. a\_i
5. Lequel des noms d'identifiant suivants pourrait ne pas être l'idée la plus judicieuse pour la valeur d'une fonction ?  
tion en un point  $x_0$  ?  
A. y0  
B. y\_0  
C. valeur\_fonction D.  
yval
6. Une façon de calculer une valeur approximative de  $\log_{10} 3$  dans Sage est de taper log\_b(3.,10.). Lequel des éléments suivants fonctionneraient également ?  
A. log(3.)  
B. log(3.)/log(10.)  
C. log(10.)/log(3.)  
D. log(10.^3)
7. Laquelle des méthodes suivantes de substitution de 2 à x dans une fonction mathématique f est garantie de fonctionner dans toutes les circonstances ?  
A. f(2)  
B. f(x=2)  
C. f.subs(x=2)  
D. f({x:2})
8. Laquelle des méthodes suivantes de substitution de 2 à x dans une expression mathématique f est garantie de fonctionner dans toutes les circonstances ?  
A. f(2)  
B. f(x=2)  
C. f.subs(x=2)  
D. f({x:2})
9. Lequel des résultats suivants attendez-vous de la commande  $\text{limit}(x/\text{abs}(x), x=0)$ ?  
A. Infini B.  
+Infini C. ind D.  
und
10. Lequel des résultats suivants attendez-vous de la commande  $\text{limit}(1/(x-1), x=1, \text{dir='right'})$  ?  
A. Infini B.  
+Infini C. ind D.  
und
11. Lequel des résultats suivants attendez-vous de la commande  $\text{limit}(\sin(1/x), x=0, \text{dir='right'})$  ?  
A. Infini B.  
+Infini C. ind D.  
und
12. Sage fera vos devoirs de calcul pour vous sur les intégrales suivantes :  
A. Intégrales exactes, car Sage peut simplifier chaque intégrale sous forme élémentaire.

- B. Intégrales approximatives, car Sage ne fournit qu'une seule réponse pour une intégrale indéfinie, plutôt que toutes les réponses possibles.
- C. Tout type d'intégrale, car Newton les a compris il y a des siècles, bien que les ordinateurs ce n'est que maintenant que nous sommes devenus suffisamment efficaces pour les réaliser.
- D. Aucun d'entre eux, car un bon professeur de calcul vérifie les étapes, pas seulement la réponse, et Sage ne montre pas les étapes.<sup>18</sup>
13. Laquelle des propriétés suivantes n'est pas garantie pour un anneau ?<sup>19</sup>
- A. Clôture de la multiplication B. Existence d'un plus petit élément C. Existence d'une identité additive D. Clôture de la soustraction
14. Laquelle des affirmations suivantes n'est pas une bonne raison d'effectuer des opérations arithmétiques sur des nombres entiers modulo n ?
- A. Dérouter les autres B. Modéliser des problèmes du « monde réel » C. Solution plus rapide à certains problèmes D. Curiosité mathématique
15. Si R est une variable qui fait référence à un anneau, quelle commande obligera Sage à considérer le nombre référencé par la variable a comme un élément de R, plutôt que comme un entier ?
- A. R(a) B. R a C. a: R D. R: a
- Bonus : Quelle réponse est correcte ?
- A. Tous. B. Certains d'entre eux. C. Au moins l'un d'entre eux. D. Tout ce qui n'est pas faux.
- Réponse courte.
1. Décrivez une autre situation dans laquelle l'arithmétique modulaire serait utile.
  2. Pour la question à choix multiples n° 2, la simplification totale est nulle pour les quatre réponses, mais Sage ne l'effectue pas automatiquement. Quelle(s) commande(s) pourriez-vous exécuter pour que Sage détecte ce problème ?
  3. Revenons à cette interaction Sage qui apparaît plus tôt dans les notes.

```
sage : d2 = CC(1+I) sage :
type(d2) <type
'sage.rings.complex_number.ComplexNumber' sage : d3 = d2 + (1 - I) sage : d3
2.0000000000000000

sage : d3 - 1,999999999999
1,00008890058234e-12
```

<sup>18</sup>... et tous les professeurs de calcul sont bons, n'est-ce pas ?

Soyez prudents sur ce coup-là ; nous sommes un peu rusés, mais seulement un peu.

Sage rapporte que la différence entre d3 et 1,99999999999 est de 1,00008890058234 ×

10–12. C'est légèrement faux ; la bonne réponse devrait être  $1 \times 10^{-12}$ .

- (a) Décrivez une séquence de commandes qui utilisent une arithmétique exacte pour nous donner la bonne réponse.
- (b) Pourquoi quelqu'un pourrait-il ne pas se soucier beaucoup du fait que l'approche utilisant l'anneau CC soit légèrement faux?

4. Dans un certain nombre de situations, vous devez éléver un nombre a à un grand exposant b, modulo un autre valeur n. Choisissez un nombre a pas trop grand (deux chiffres suffisent) et augmentez-le de plus en plus exposants b, modulo n en utilisant l' opérateur % , jusqu'à ce que vous constatiez un ralentissement notable. (Ne se laisser emporter, car une fois que ça ralentit, ça commence vraiment à ralentir.) Ensuite, comparez cela opération lorsque Sage l'exécute dans l'anneau. Notez ces valeurs pour a, b et n, et indiquez s'il est vraiment plus rapide d'utiliser l'anneau que d'utiliser l' opérateur % .

5. Nous avons seulement affirmé que c'est un anneau, mais pour certaines valeurs de n, c'est en fait un corps. Il n'y a pas question de savoir si la multiplication est commutative (elle l'est) mais si chaque élément a un L'inverse multiplicatif n'est pas très clair. Vérifiez ceci pour les valeurs n = 2, 3, 4, 5, 6, 7, 8, 9, 10 par tester autant de produits que nécessaire pour trancher la question.

- (a) Voyez-vous un modèle dans les valeurs de n pour lequel est un champ ?
- (b) Lorsque n'est pas un champ, voyez-vous un modèle dans les valeurs de a pour lequel vous pouvez trouver un inverse multiplicatif b ?

Indice : nous n'avons pas besoin d'un inverse multiplicatif pour 0, 1 est son propre inverse (après tout,  $1 \times 1 = 1$ ), et dans nous avons commodément n = 0. Avec la propriété commutative, cela signifie vous devez vérifier au moins  $\frac{(n-2)(n-1)}{2}$  produits ; avec n = 10, vous n'avez besoin de vérifier que 35. Donc, si vous êtes intelligemment à ce sujet, ce n'est pas un problème aussi lourd que vous pourriez le penser.

Bonus : Pourquoi avons-nous pu dire, avec assurance, qu'au point n°5, nous devons au plus déterminer s'il s'agit d'un champ ?

## CHAPITRE 3

## De jolies (et moins jolies) photos

La visualisation apporte souvent des informations que nous n'obtenons pas par d'autres méthodes. Cela ne s'applique pas seulement aux mathématiques ; d'où le dicton : « Une image vaut mille mots. » Nombre de nos exercices et travaux pratiques vous demanderont de produire des images, puis d'en tirer des conclusions.

Sage propose un ensemble d'outils très performants pour dessiner des graphiques, ainsi qu'une interface intuitive. Ce chapitre examine les objets bidimensionnels que vous pourriez être amené à utiliser. Tout ce que nous faisons ici relève de la géométrie cartésienne ; nous partons du principe que nous travaillons sur un plan cartésien.

À partir de ce chapitre, nous ne nous contenterons pas de lister les commandes avec une brève explication ; nous les détaillerons également avec leurs options. Veuillez noter que la description des options peut être incomplète :

- Nous nous concentrerons uniquement sur quelques aspects de la commande que nous pensons être les plus utiles pour les tâches à accomplir et pour ce dont vous auriez besoin à l'avenir.
- Sage est voué à changer à l'avenir, et il est peu probable que chaque commande Sage soit longue rester le même que ce qu'il est maintenant.

N'oubliez pas que chaque commande Sage vous offrira une explication plus complète et une liste d'options plus complète si vous saisissez simplement son nom, suivi d'un point d'interrogation. Par exemple :

```
sage : point ?
Signature : point(points, **kwds)
Docstring :
    renvoie un point ou une somme de points en 2 ou 3 dimensions.
```

... et ainsi de suite.

## objets 2D

Nous commençons par examiner quelques objets bidimensionnels fondamentaux.

Des choses « droites ». On peut illustrer de nombreux concepts mathématiques simplement en observant des points et des segments de droite. Sage propose trois commandes pour représenter graphiquement ces éléments : `point()`, `line()` et `arrow()`. À proprement parler, `line()` produit un segment de ligne courbé, pas nécessairement une ligne droite.

Chaque description commence par un modèle de commande, listant les informations obligatoires et facultatives (généralement appelées options), suivi d'une liste à puces décrivant ces informations. Vous n'êtes pas obligé de fournir toutes les informations ; pour chaque information, nous ajoutons donc entre parenthèses la valeur que Sage attribue à cette information si vous l'omettez.

Les commandes `point()` et `line()` font référence aux « collections ». Nous aborderons les collections plus en détail ultérieurement, mais pour l'instant, vous pouvez utiliser des tuples, une séquence d'objets entre parenthèses. Par exemple :

---

point(position,options)

- la position est
    - une paire ordonnée, ou –
    - une collection de paires ordonnées • les options incluent : –
- pointsize = size (10)
- 

**FIGURE 1. point()**

---

ligne(points,options)

- points est une collection de paires ordonnées • les options incluent : –
- épaisseur = épaisseur (1)
- 

**FIGURE 2. ligne()**

---

flèche(point de queue,point de tête,options) ou flèche(chemin,options) • le point de queue est une paire ordonnée ; la flèche commence ici • le point de tête est une paire ordonnée ; la flèche se termine ici • les options incluent : – taille de la flèche = taille de la pointe de la flèche (5) – tête = emplacement de la pointe de la flèche (1) 0 signifie « au point de queue » 1 signifie « au point de tête » 2 signifie « aux deux extrémités » – largeur = largeur de la tige (2)

---

**FIGURE 3. flèche()**

---

Les options suivantes sont communes à tous les objets 2D :

- alpha=transparence (1.0) • color=couleur, dont nous discutons dans une section dédiée (« bleu » ou (0,0,1)) • linestyle=style de dessin de la ligne, qui peut être « dashdot », « dashed », « dotted » ou 'solide' ('solide')
  - zorder=distance au spectateur, par rapport aux autres objets (dépend de l'objet)
- 

**FIGURE 4. Options communes à tous les objets 2D**

$(2,3)$  est un tuple d'entiers. Dans ce cas, il ne contient que deux éléments ; on parle donc de paire ordonnée. Les tuples peuvent être beaucoup plus longs :  $(x, x^{**}2, x^{**}3, x^{**}4)$  est un tuple d'expressions symboliques, mais n'est pas une paire ordonnée.

La figure 4 répertorie quelques options courantes pour tous les objets bidimensionnels. Dans cette section, nous aborderons toutes les options, à l'exception de la couleur ; une page lui est consacrée.

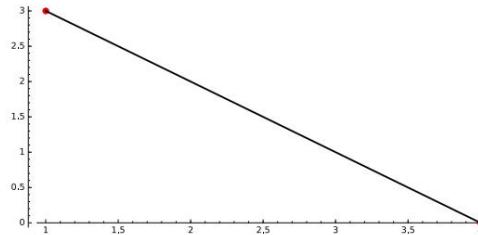
En plus d'illustrer les objets, notre premier exemple montrera l'intuitivité de la combinaison de plusieurs images : il suffit de les additionner avec l'opérateur `+`. Pour faciliter la lecture, nous assignerons plusieurs objets à des variables et les combinerons à la fin.

```
sage : p1 = point(((1,3),(4,0)), couleur='rouge', \ pointsize=60)
```

```
sage : p2 =
```

```
ligne(((1,3),(4,0)), couleur='noir', \ épaisseur=2)
```

```
sage : p1 + p2
```



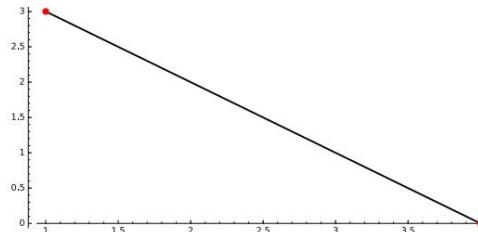
Nous avons une belle ligne noire reliant deux points rouges. (Remarquez l'utilisation du saut de ligne pour commencer une nouvelle ligne. N'oubliez pas que ce n'est pas obligatoire, surtout si vous disposez probablement de plus d'espace que nous.)

Si vous regardez attentivement, vous remarquerez la présence de points rouges sous la ligne noire. La plupart des gens trouvent cela peu esthétique, et il existe une solution miracle : l'option `zorder`. Nous avons déjà mentionné que l'ordre `z` d'un objet indique sa proximité par rapport aux autres objets de l'image. Plus le nombre est grand, plus il est proche de l'observateur. Le nombre 0 peut être considéré comme une position médiane ; les nombres positifs apparaîtront « devant » le zéro, et les nombres négatifs « derrière ».

Nous pouvons utiliser cela pour « soulever » les points au-dessus de la ligne.

```
sage : p1 = point(((1,3),(4,0)), couleur='rouge', \ pointsize=60,
zorder=5) sage : p2 = ligne(((1,3),
(4,0)), couleur='noir', \ épaisseur=2, zorder=0)
```

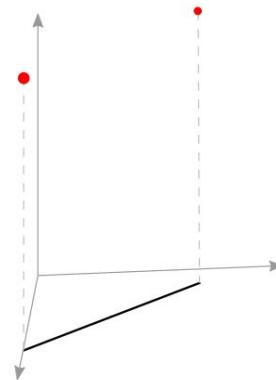
```
sage : p1 + p2
```



Sage considère désormais les points comme étant au niveau 5, plus proches du spectateur, tandis que la ligne reste au niveau 0, plus éloignée. C'est pourquoi les points se situent désormais au-dessus de la ligne.

Pourquoi cette option appelée `zorder`? Cela paraît logique si vous êtes familier avec les graphiques tridimensionnels : elle indique la valeur  $z$  de l'objet graphique, vu au-dessus du plan  $xy$ .

Nous illustrons cela avec une image 3D :

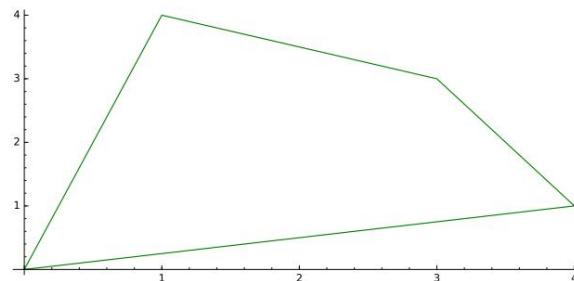


Imaginez-vous debout au-dessus de cela, regardant vers le bas : vous verriez les points rouges, à  $z = 5$ , devant la ligne, à  $z = 0$ . (Les lignes pointillées illustrent que les points rouges se trouvent au-dessus des extrémités de la ligne.)

Nous avons mentionné plus tôt que la commande `line()` produit en réalité un segment de ligne « tordu ».

Voici un exemple de cela en action.

```
sage : ligne(((0,0),(1,4),(3,3),(4,1),(0,0)), couleur='vert')
```



Dans ce cas, nous avons utilisé la ligne pour dessiner une figure fermée. Bien que cela soit possible, il serait plus pratique d'utiliser la commande `polygon()`, surtout si vous souhaitez remplir l'objet. Avec un polygone, vous n'avez pas besoin de spécifier à nouveau le premier point comme dernier point, car un polygone est implicitement une figure fermée.

```
sage : polygon(((0,0),(1,4),(3,3),(4,1)), couleur='vert')
```

---

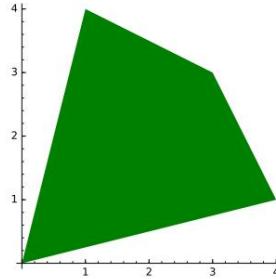
polygone(points,options) •

points est une collection de paires ordonnées

- les options incluent :

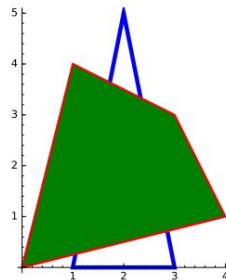
- couleur = couleur de remplissage ('bleu' ou  
(0,0,1)) notez que cela diffère de l'interprétation habituelle de la couleur – edgecolor = couleur du bord ('bleu' ou  
(0,0,1)) – remplissage = s'il faut remplir le polygone (Faux) – épaisseur = épaisseur du bord (1)
- 

FIGURE 5. polygone()



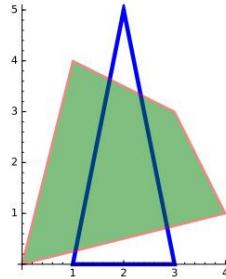
Vous n'êtes pas obligé de remplir un polygone. Ajoutons un autre polygone sous celui-ci. Tant qu'à faire, nous allons lui ajouter un bord rouge pour le faire ressortir.

```
sage : p1 = polygone(((0,0),(1,4),(3,3),(4,1)), couleur='vert', \
edgecolor='red', thickness=2) sage: p2 =
polygon(((1,0),(2,5),(3,0)), thickness=4, fill=False, \
zorder=-5)
sage : p1 + p2
```



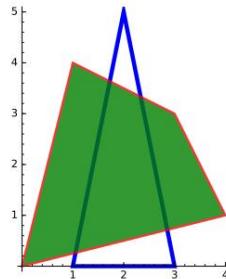
Dans certaines situations, nous souhaitons « voir à travers » la figure du haut pour voir celle du dessous. C'est là qu'intervient l' option alpha . Cette option contrôle la transparence d'un objet. Ses valeurs vont de 0 (invisible) à 1 (opaque).

```
sage : p1 = polygone(((0,0),(1,4),(3,3),(4,1)), couleur='vert', \
edgecolor='red', thickness=2, alpha=0.5) sage: p2 = \
polygon(((1,0),(2,5),(3,0)), thickness=4, fill=False, \
zorder=-5)
sage : p1 + p2
```



C'est un peu trop transparent. Nous voulons juste suggérer qu'il y a quelque chose en dessous ; nous ne voulons pas que le triangle bleu semble être au-dessus. Nous modifions l'alpha en conséquence.

```
sage : p1 = polygone(((0,0),(1,4),(3,3),(4,1)), couleur='vert', \
edgecolor='red', thickness=2, alpha=0.8) sage: p2 = \
polygon(((1,0),(2,5),(3,0)), thickness=4, fill=False, \
zorder=-5)
sage : p1 + p2
```



Soyez prudent avec la commande `polygon()`. Elle ne peut pas lire dans vos pensées ; elle suit les points dans l'ordre précis que vous spécifiez. Il suffit de peu pour transformer un pentagone en pentagramme.

```
sage: polygon(((1,0), (cos(2*pi/5),sin(2*pi/5)), \
(cos(4*pi/5),sin(4*pi/5)), \ (cos(6*pi/ \
5),sin(6*pi/5)), \ (cos(8*pi/5),sin(8*pi/ \
5))), épaisseur=2)
```

---

cercle(centre,rayon,options) • le centre

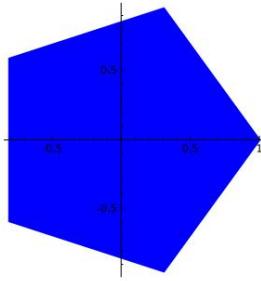
est une paire ordonnée • le rayon  
est un nombre réel • les options

incluent – fill = s'il faut

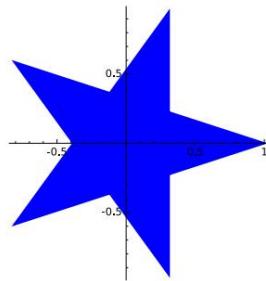
remplir le cercle (Faux) – edgecolor = couleur du  
bord du cercle ('bleu' ou (1,0,0)) – facecolor = couleur utilisée pour remplir  
le cercle ('bleu' ou (1,0,0)) – si vous spécifiez la couleur, alors Sage ignore  
edgecolor et facecolor même si vous spécifiez  
les explicitement

FIGURE 6. cercle()

---



```
sage: polygon(((1,0), (cos(4*pi/5),sin(4*pi/5)), \
(cos(8*pi/5),sin(8*pi/5)), \ (cos(2*pi/5),sin(2*pi/5)), \ (cos(6*pi/5),sin(6*pi/5))), épaisseur=2)
```



Nous n'illustrerons pas les flèches telles que définies dans cette section ; voir un exemple dans une section suivante.  
N'hésitez pas à les expérimenter par vous-même.

Des objets « courbés ». Nous considérons trois autres objets : les cercles, les ellipses et les arcs.

Le premier exemple est un arc, principalement destiné à illustrer la relation entre l'angle, le rayon,  $r^2$  et le secteur. À cette fin, nous ajouterons des lignes pointillées pour illustrer l' option de style de ligne .

---

`ellipse(centre,hradius,vradius,options)`

- le centre est une paire ordonnée • `hradius` et `vradius` sont des nombres réels (rayons horizontal et vertical)
- les options incluent
  - `fill` = s'il faut remplir le cercle (`False`) –
  - `edgecolor` = couleur du bord du cercle ('bleu' ou `(1,0,0)`) – `facecolor` = couleur utilisée pour remplir le cercle ('bleu' ou `(1,0,0)`) – si vous spécifiez la couleur, alors Sage ignore `edgecolor` et `facecolor` même si vous spécifiez les explicitement

---

FIGURE 7. `ellipse()`

---

`arc(centre,rayon,options)`

- le centre est une paire ordonnée • le rayon est un nombre réel • les options incluent – `angle` = angle entre l'axe de l'arc et l'axe des x ; la longueur de cet axe sera le rayon (0) – `r2` = un deuxième rayon, pour l'arc d'une ellipse ; perpendiculaire à l'angle (égal au rayon, obtenant l'arc d'un cercle) – `secteur` = une paire ordonnée, indiquant les angles qui définissent le début et la fin du secteur  $((0, 2\pi))$

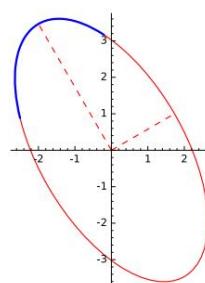
---

FIGURE 8. `arc()`

---

```
sage: p1 = arc((0,0), 2, angle=pi/6, r2=4, sector=(pi/4,5*pi/6), \
épaisseur = 2)
sage : p2 = ligne(((0,0), (2*cos(pi/6),2*sin(pi/6))), couleur='rouge', \ style de ligne='pointillé')
sage : p3 = ligne(((0,0),
(4*cos(pi/6+pi/2),4*sin(pi/6+pi/2))), \ couleur='rouge', style de ligne='pointillé')

sage : p4 = ellipse((0,0),2,4,pi/6,couleur='rouge',zorder=-5) sage : p1 + p2 + p3
+ p4
```



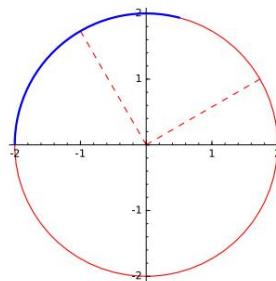
Notez que l'axe horizontal de l'ellipse est incliné selon l'angle  $\pi/6$ , comme l'illustre la ligne rouge en pointillés dans le premier quadrant. L'axe vertical est également incliné ; l'option d'angle a donc entraîné une rotation de l'ensemble de l'ellipse de  $\pi/6$ . Pour tracer l'arc, nous avons spécifié le secteur ( $\pi/4, \pi/6$ ). L'arc commence alors à l'angle  $\pi/4 + \pi/6 = 5\pi/12$ .

Vous vous demandez peut-être comment cette dernière affirmation peut être vraie alors que l'arc bleu commence clairement dans le deuxième quadrant, plutôt que dans le premier ; après tout,  $5\pi/12 < 6\pi/12 = \pi/2$ , et  $\pi/2$  est l'axe vertical. La réponse réside dans la distorsion causée par l'axe « vertical » de l'ellipse ; il tire le point final de l'arc vers le haut, mais comme l'ellipse a pivoté, « vers le haut » signifie en réalité vers le nord-nord-ouest, obtenant ainsi un point dans le deuxième quadrant. Ne nous croyez pas sur parole ; essayez avec un cercle en omettant toute spécification de  $r^2$ , et vous verrez si cela devient plus logique.

```
sage : p1 = arc((0,0), 2, angle=pi/6, secteur=(pi/4,5*pi/6), \ épaisseur=2)

sage : p2 = ligne(((0,0), (2*cos(pi/6),2*sin(pi/6))), couleur='rouge', \ style de ligne='pointillé')
sage : p3 = ligne(((0,0),
(2*cos(pi/6+pi/2),2*sin(pi/6+pi/2))), \ couleur='rouge', style de ligne='pointillé')

sage : p4 = cercle((0,0),2,couleur='rouge',zorder=-5) sage : p1 + p2
+ p3 + p4
```



Un aparté, pour ceux qui manquent de confiance en trigonométrie (et pour ceux qui n'en ont pas, mais devraient). Si vous vous demandez d'où viennent les paires ordonnées des droites, rappelons un peu de trigonométrie. Rappelons d'abord qu'en notation radian, un tour complet est considéré comme ayant un angle de  $2\pi$ , plutôt que de  $180^\circ$ . Nous divisons traditionnellement le tour complet à partir de là en un demi-tour ( $2\pi/2 = \pi$ ), un quart de tour ( $2\pi/4 = \pi/2$ ), un sixième de tour ( $2\pi/6 = \pi/3$ ) et un douzième de tour ( $2\pi/12 = \pi/6$ ).

Par ailleurs, les fonctions  $\cos\alpha$  et  $\sin\alpha$  représentent les valeurs  $x$  et  $y$  d'un point du cercle unité situé à un angle  $\alpha$  par rapport à l'horizontale. Par exemple, le point du cercle unité situé à un angle de  $\pi/3$  par rapport à l'horizontale serait

$$\text{donc } \frac{1}{2}, \frac{\sqrt{3}}{2}, \text{ donc } \frac{1}{2}, \frac{\sqrt{3}}{2} = \frac{1}{2}, \frac{\sqrt{3}}{2}.$$

Que faire si vous souhaitez un cercle de rayon différent ? Dans ce cas, la relation entre le cosinus, le sinus et le triangle formé par le rayon et les valeurs  $x$  et  $y$  implique que vous devez mettre à l'échelle le point en multipliant par le rayon du cercle. Par exemple, le point sur un cercle centré de rayon 2 à une

texte( chaîne de texte, position, options)

- la chaîne de texte est l'étiquette, entourée de guillemets simples ou doubles
- la position est une paire ordonnée • les options incluent – fontsize = taille du texte (10) – rotation = angle de rotation du texte, en degrés (0)

FIGURE 9. texte()

l'angle de  $\pi/6$  par rapport à l'horizontale serait

$$\frac{2 \cos \pi}{6}, \frac{2 \sin \pi}{6} = 3,1^\circ.$$

L'un des objectifs d'un système de calcul formel est de nous épargner la simplification. Plutôt que de calculer ( $\sqrt{3}, 1$ ) comme un point sur la droite, nous avons fourni à Sage l'expression ( $2*\cos(pi/6)$ ,  $2*\sin(pi/6)$ ) et l'avons laissé calculer le reste à notre place.

De même, lorsque nous avons voulu placer un point le long de l'angle de l'axe « vertical » de l'ellipse, plutôt que de déterminer la valeur simplifiée, nous avons simplement demandé à Sage de faire la rotation, puis de trouver les points correspondants :

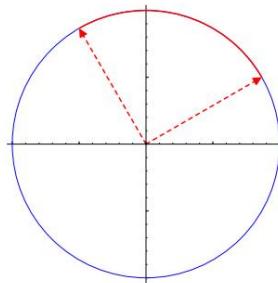
$$\frac{p}{6} + \frac{p}{2}, \quad , \quad \begin{matrix} p & p \\ 6 & 2 \end{matrix}$$

4 cos                          4 péchés                          4 cos                          ,                          4 cos  
commencer      tourner                          ,                          commencer      tourner

Nous pouvons illustrer cela dans Sage :

```
sage : p1 = cercle((0,0),2) sage : p2
= flèche((0,0),(2*cos(pi/6),2*sin(pi/6)), \ linestyle='dashed', color='red')
sage : p3 = flèche((0,0), (2*cos(pi/6+pi/2),
2*sin(pi/6+pi/2)), \ linestyle='dashed', color='red')

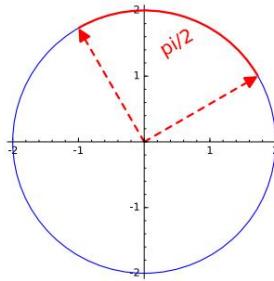
sage : p4 = arc((0,0), 2, angle=pi/6, secteur=(0,pi/2), couleur='rouge', \
épaisseur=2, zorder=5)
sage : p1 + p2 + p3 + p4
```



Notez l'utilisation de arrow() et de l' option linestyle .

Texte. Ne serait-il pas judicieux d'ajouter des étiquettes à cette dernière image ? La commande text() permet Nous insérons des étiquettes dans une image. Notre exemple reprend les images de l'exemple précédent.

```
sage : p5 = p1 + p2 + p3 + p4 sage : p6
= texte('pi/2', (.5,1.5), couleur='rouge', taille de police=18, \
rotation=30)
sauge : p5 + p6
```



Cela semble correct, mais ne serait-il pas plus agréable d'avoir un rendu un peu plus « professionnel » ? Ne serait-il pas plus agréable d'utiliser la lettre grecque  $\pi$  au lieu de quelques caractères latins ? Une solution serait de modifier la disposition du clavier de votre ordinateur afin de pouvoir saisir des lettres grecques, sans pour autant ressembler à une fraction.<sup>1</sup> Sage propose une solution mathématique élégante via LATEX.

Sans entrer dans les détails, LATEX est une sorte de « langage de balisage » développé dans les années 1970 et 1980. Il est très courant pour les mathématiciens de saisir des articles et des livres en LATEX (vous lisez un exemple), car il offre des commandes extrêmement intuitives et flexibles pour indiquer ce que vous souhaitez écrire, et organise automatiquement le texte d'une manière cohérente avec la tradition mathématique, avec un placement approprié des exposants, un étirement des symboles de regroupement, etc.

Sage ne propose pas l'intégralité du balisage LATEX ; il n'en propose qu'un sous-ensemble, mais celui-ci est suffisant pour les travaux pratiques en graphisme. Pour utiliser le balisage LATEX, deux tâches suffisent :

- entourez la sous-chaîne de mathématiques dans les délimiteurs LATEX \$ et \$ ; et • utilisez le balisage LATEX approprié , comme décrit au chapitre 12.

Parenthèse. Les utilisateurs de LATEX remarqueront que nous utilisons des doubles barres obliques inverses alors qu'ils s'attendraient normalement à des barres obliques inverses simples. En effet, la barre oblique inverse simple est une commande de contrôle dans les chaînes Sage2 : \n, \r, \t ont toutes des significations particulières (comme d'autres). Une simple barre oblique inverse peut parfois suffire, mais elle peut aussi semer le chaos aux moments les plus inopportunus. L'utilisation systématique d'une double barre oblique inverse permet d'éviter toute ambiguïté potentielle.

Et pourtant... Si vous regardez la figure 1, page 252, vous remarquerez que nous avons utilisé des barres obliques inverses simples. En effet, nous aborderons plus tard l'insertion de LATEX dans la feuille de calcul, et dans ce cas, une double barre oblique inverse est inappropriée. C'est une source de confusion regrettable pour l'apprenant, mais la règle de base est assez simple : le balisage nécessite une barre oblique inverse simple, mais dans une chaîne de texte, cela signifie qu'il faut une double barre oblique inverse.

De plus... Vous pouvez également utiliser LATEX dans les cellules HTML d'une feuille de calcul ! (Voir p. 26 pour une explication sur la création d'une cellule HTML.) Entourez le texte de signes dollar, entre barres obliques inverses.

<sup>1</sup>De plus, on a essayé. Ça ne marche pas.

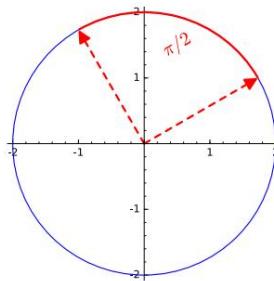
2C'est parce que Sage est construit sur Python.

parenthèses, ou entre crochets avec barre oblique inverse, et vous pouvez utiliser le balisage LATEX à votre guise.<sup>3</sup> Par exemple, le code suivant vous donnera une belle présentation de la définition de l'intégrale :

```
sage: %html
    La définition de la <b>dérivée</b> est <i>la limite de
    les pentes des droites sécantes lorsque la distance entre les points approche de
    0 ;</i> ou, $\frac{d}{dx}f(x) = \lim_{\Delta x \rightarrow \infty} \frac{f(x+\Delta x)-f(x)}{\Delta x}$ 
```

Revenons à notre programmation habituelle. Reprenons l'image précédente, en utilisant LATEX cette fois pour un rendu plus soigné. Si vous regardez la figure 1, vous trouverez  $\pi$  dans la ligne des lettres grecques ; pour obtenir le symbole grec, il vous faut le balisage  $\backslash\pi$ , qui, dans une chaîne Sage, s'écrit  $\backslash\pi$ .

```
sage : p7 = texte('$\backslashpi/2$', (.5,1.5), couleur='rouge', \
taille de police = 18, rotation = 30)
sauge : p5 + p7
```

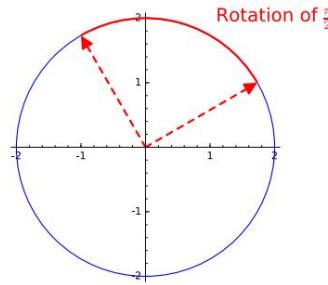


Cela semble bien mieux. Mais peut-être préféreriez-vous que cela ressemble à une fraction réelle, plutôt qu'à une simple division ? Regardez à nouveau la figure 1 et vous verrez un balisage  $\backslashfrac$ . Cela se traduit par  $\backslashfrac$  dans une chaîne Sage, mais cela illustre également un autre aspect du balisage LATEX : certaines commandes nécessitent des informations supplémentaires. Ces informations sont généralement transmises entre accolades, parfois entre plusieurs. Pour une fraction, cela devrait être logique : une fraction nécessite un numérateur et un dénominateur, donc deux informations, donc deux paires d'accolades. Nous illustrons cela dans l'exemple ci-dessous, qui diffère légèrement des précédents.

```
sage : p8 = texte('Rotation de $\backslashfrac{\backslashpi}{2}$', (.5,1.5), \
couleur='rouge', taille de police=18)
sauge : p5 + p8
```

---

<sup>3</sup>Eh bien, peut-être pas à votre goût, à plus d'un titre. Dans la limite du raisonnable, en tout cas.



Couleurs. Il existe plusieurs façons de définir les couleurs dans Sage. L'une d'elles est par nom, et Sage propose une un très grand nombre de couleurs par nom, avec des options intrigantes telles que « chartreuse » et « lavenderblush ». Vous pouvez obtenir une liste complète des noms en tapant ce qui suit :

```
sage : couleurs.clés()
dict_keys(['automatique', 'aliceblue', 'antiquewhite', 'aqua',
'aigue-marine', ...])
```

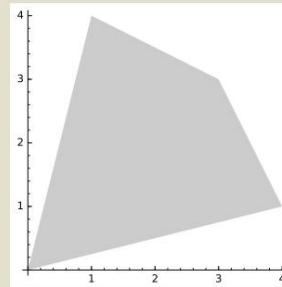
À l'heure actuelle, il existe près de 150 couleurs prédéfinies, nous en avons donc laissé de côté un grand nombre.

On peut en fait obtenir plusieurs millions de couleurs (au moins 16 millions environ) en manipulant ce qu'on appelle les « valeurs RVB ». La lumière visible peut être décomposée en trois composantes : le rouge, vert et bleu — et nous pouvons demander à Sage d'attribuer à chacun de ces composants, appelés canaux, une valeur de 0 à 1. Vous pouvez penser à « 0 » comme signifiant que le canal est complètement éteint, et « 1 » comme signifiant Le canal est complètement saturé. Ceci nous amène à ceci :

RVB		interprétation	résultat
1 0 0		tout rouge, pas de vert ni de	rouge
1 1 0	0,5	bleu tout rouge et vert, pas	jaune
0,5 0,5		de bleu demi-puissance rouge,	gris
vert, bleu	0,8 0,8 0,8	puissance huit dixièmes rouge, vert, bleu gris plus clair	

Pour utiliser les couleurs de cette manière dans Sage, transmettez simplement les trois nombres, entre parenthèses, comme couleur option.<sup>4</sup> Pour illustrer cela, nous revisitons notre pentagone d'avant.

```
sage : polygone(((0,0),(1,4),(3,3),(4,1)), couleur=(0,8,0,8,0,8))
```




---

<sup>4</sup>Il serait plus précis de le spécifier comme l' option `rgbcolor` , mais Sage interprète la couleur comme `rgb`, ce qui est assez En tout cas, c'est standard. Si vous connaissez le modèle de couleur Teinte-Saturation-Valeur, vous pouvez utiliser l' option `hsvcolor`. au lieu de cela, mais nous ne couvrons pas cela ici.

---

show(objet graphique , options) •

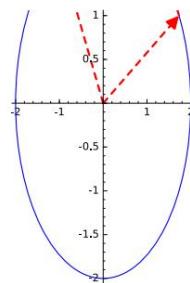
ymax = plus grande valeur y visible (déterminée par l'objet) • ymin  
= plus petite valeur y visible (déterminée par l'objet) • xmax = plus  
grande valeur y visible (déterminée par l'objet) • xmin = plus petite  
valeur y visible (déterminée par l'objet) • aspect\_ratio = rapport de 1  
unité y à 1 unité x (déterminé par l'objet) • axes = indique s'il faut afficher les axes  
(True)

---

**FIGURE 10.** show()

Autres options pour les objets 2D. Certaines options pour les objets 2D sont plus facilement manipulables après leur addition. Il existe plusieurs façons de les manipuler ; vous pouvez, par exemple, ajouter chaque option ci-dessous directement à une commande de tracé individuelle. Si vous avez plusieurs tracés, la commande `show()` est la plus pratique , car elle offre plusieurs options pour affiner une image. Nous listons celles qui nous intéressent le plus dans la figure 10. Nous les avons mises en pratique à la page 5, l'image ci-dessus du cercle avec rotation.

**sage :** afficher(p5, ymax=1, rapport\_aspect=2)



Vous pouvez voir que la plus grande valeur y visible est  $y = 1$ , et le cercle semble avoir été étiré verticalement, de sorte qu'une unité de 1 sur l'axe des y est deux fois plus longue qu'une unité de 1 sur l'axe des x.

Il est souvent pratique d'afficher des graphiques sans les axes ; `show` propose l' option `axes` pour cela.

Nous illustrerons cela plus tard, mais vous pouvez essayer de retravailler certains des graphiques listés ici pour voir le fonctionnement. Il existe d'autres options pour contrôler les axes que nous n'aborderons pas ici ; la plupart d'entre elles peuvent également être contrôlées via les méthodes d'un objet de tracé. Pour les découvrir, saisissez le nom d'un objet de tracé, suivi du point, puis de la touche Tabulation ; sélectionnez l'objet qui vous intéresse, saisissez le point d'interrogation et exécutez la commande pour afficher son aide (comme expliqué précédemment).

#### tracés 2D

À présent, vous devriez avoir appris trois manières de base de représenter les relations bidimensionnelles que nous Vous souhaitez tracer. Il

s'agit de : • coordonnées cartésiennes,

soit – avec y comme fonction de

x ; – sous forme d'équation en termes de x et y uniquement, mais pas nécessairement une

fonction ; ou – avec x et y en termes d'une troisième variable, t, appelée paramètre ; et

---

`plot( f(x),xmin,xmax,options)` •  $f(x)$  est

une fonction ou une expression dans une variable •  $x_{\min}$  est  
la plus petite valeur  $x$  à utiliser pour le tracé •  $x_{\max}$  est la  
plus grande valeur  $x$  à utiliser pour le tracé • les options  
incluent – `detect_poles`

= l'un des éléments suivants (`False`) :    `False` (dessiner  
directement)    `True`  
(détecter la division par zéro et dessiner de manière appropriée)  
`'show'` (identique à `True`, mais inclure une asymptote) – `fill =`  
l'un des éléments suivants (`False`)    `False`  
(ne pas remplir)    `'axis'`  
ou `True` (remplir jusqu'à l'axe des  $x$ )    `g`  
( $x$ ) (remplir jusqu'à la fonction  $g(x)$ )  
`'min'` (remplir de la courbe jusqu'à sa valeur minimale)    `'max'`  
(remplir de la courbe jusqu'à sa valeur maximale) – `fillcolor =`  
couleur utilisée pour le remplissage, ou `'automatic'` ('automatique') – `fillalpha =`  
transparence du remplissage, de 0 à 1 (0,5) – `plot_points =`  
nombre de points à utiliser lors de la génération d'un tableau  $xy$  (200)

---

FIGURE 11. `plot()`

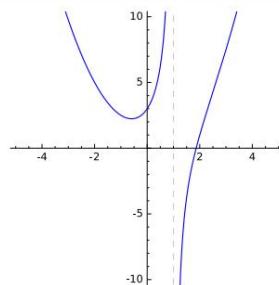
- coordonnées polaires.

Sage propose des commandes pour chacune de ces représentations.

Graphiques génériques. La commande la plus utilisée est la commande `plot()` , qui accepte une fonction, ou au moins une expression dans une variable, et produit un graphique où la valeur  $y$  est déterminée par les valeurs  $x$ , comme dans un tableau  $xy$ . Cette description est assez littérale ; la commande fonctionne en créant un certain nombre de paires  $(x, y)$  et en reliant les points. Ses options incluent celles listées dans la figure 4, [page 53](#). Il est important de noter que  $x_{\min}$  et  $x_{\max}$  n'ont pas la même signification ici et dans la commande `show()` . Ici, ils correspondent aux valeurs  $x$  les plus petites et les plus grandes pour lesquelles `plot()` génère une paire  $xy$ . Il est tout à fait possible de faire afficher() plus ou moins que ce montant en définissant `xmin` et `xmax` sur des valeurs différentes dans la commande `show()` .

Nous illustrons cette commande sur une fonction qui possède des pôles.

```
sage : p = plot(x^2 - 3/(x-1), -5, 5, detect_poles='show') sage : show(p,
ymin=-10, ymax=10, aspect_ratio=1/2)
```



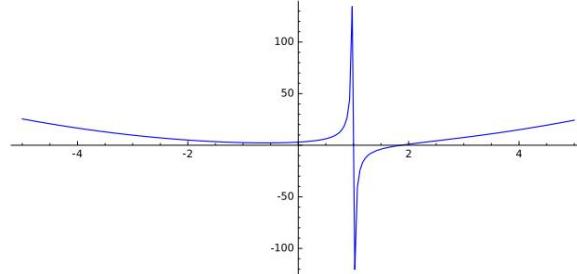
```
tracé_paramétrique( x(t), y(t) , (t,tmin,tmax) ,options)
```

- $x(t)$  et  $y(t)$  sont des fonctions ou des expressions en termes d'un paramètre  $t$ , qui définissent le Valeur  $x$  et  $y$  d'un point •  $t$
- est le paramètre •  $t_{\min}$
- est la plus petite valeur  $t$  à utiliser •  $t_{\max}$
- est la plus grande valeur  $t$  à utiliser • les options incluent –
  - `fill = True` (vers l'axe) ou `False` (aucun) (`False`) –
  - `fillcolor`, comme dans `plot()` –
  - `fillalpha`, comme dans `plot()`

FIGURE 12. paramétrique\_tracé()

Si nous ne contraignions pas les valeurs  $y$ , l'image serait très différente. (Vous pouvez essayer la commande `show()` sans les valeurs  $y_{\min}$  et  $y_{\max}$  pour vous en rendre compte.) De même, si vous omettez `detect_poles`, Sage reliera par erreur les deux points les plus proches du pôle, ce qui ressemble généralement à une asymptote.

```
sage : p = plot(x^2 - 3/(x-1), -5, 5) sage :
show(p)
```



Comme nous l'avons suggéré plus haut, c'est techniquement faux, car cette ligne apparemment verticale ne l'est pas ; elle relie deux points. Si vous voyiez cela se produire dans la réalité, vous pourriez conclure qu'il s'agit d'une asymptote, mais vous pourriez aussi vous tromper ; une caractéristique intéressante de la fonction pourrait bien s'y produire.

Tracés paramétriques. La commande `parametric_plot()` est étroitement liée à la commande `plot()`. Les tracés paramétriques sont fréquemment utilisés lorsque les valeurs de  $x$  et  $y$  dépendent d'un instant donné, comme c'est le cas en physique et en ingénierie. Dans de nombreux cas,  $t \in [0, 1]$ , où « 0 » indique la position de départ et « 1 » la position d'arrivée, mais il existe de bonnes raisons d'utiliser d'autres valeurs. L'exemple suivant utilise  $t \in [0, 2\pi]$ .

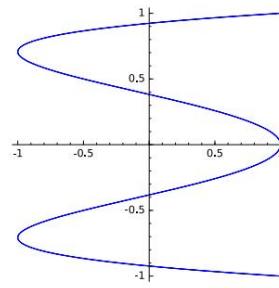
```
sage : var('t')
sage : tracé_paramétrique((cos(4*t),sin(t)), (t,0,2*pi))
```

---

```
polar_plot( r (θ), (θ, θmin , θmax) ,options)
```

- $r (\theta)$  est une fonction ou une expression définie en termes d'une autre variable  $\theta$ , pour laquelle vous peut utiliser  $x$
  - $\theta$  est la variable indépendante représentant l'angle •  $\theta_{\text{min}}$  n'est le plus petit angle à utiliser •  $\theta_{\text{max}}$  est le plus grand angle à utiliser • les options incluent –
    - `fill = True` (vers l'axe) ou `False` (aucun) (`False`) –
    - `fillcolor`, comme dans `plot()` –
    - `fillalpha`, comme dans `plot()`
- 

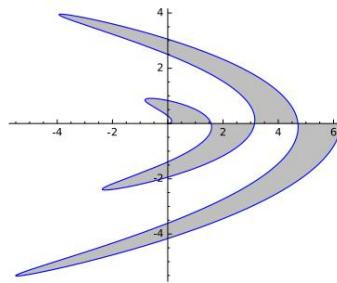
FIGURE 13. `polar_plot()`



Voici le graphique complet de cette fonction paramétrique. Jouez avec différentes valeurs de  $t_{\text{min}}$  et  $t_{\text{max}}$  pour déterminer le plus petit intervalle  $[a, b]$  permettant d'obtenir une onde complète.

L'exemple suivant illustre l'utilisation de l' option de remplissage .

```
sage : tracé_paramétrique((t*cos(4*t),t*sin(2*t)),(t,0,2*pi),fill=True)
```



C'est déjà bien, mais vous pouvez obtenir une image encore plus nette en doublant cette courbe de différentes manières. Essayez et voyez ce qui se passe.

Diagrammes polaires. Nous nous tournons vers les coordonnées polaires. Elles sont couramment utilisées pour décrire plus facilement l'évolution de la position d'un objet lors de sa rotation autour de son origine. Certaines images difficiles à décrire par paires ( $x, y$ ) deviennent plus faciles à décrire avec les coordonnées polaires, comme la fleur de lys abstraite ci-dessous.

---

animer( collection, options) • la  
collection peut être une liste, un ensemble ou un tuple  
de tracés • les  
options incluent – xmin, xmax, ymin, ymax et aspect\_ratio, comme dans show()

---

FIGURE 14. animate()

---

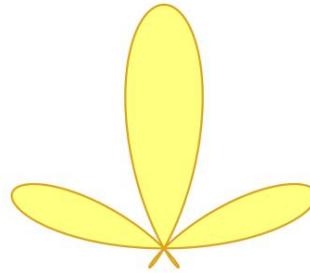
afficher( animation,options)

- l'animation est une animation générée à l'aide de la commande animate()
- les options incluent
  - delay = centièmes de seconde avant de passer à l'image suivante (20) –
  - iterations = nombre de fois pour répéter l'animation depuis le début, où 0 indique « pour toujours »
  - (0) – gif = s'il faut produire une animation GIF ou une animation WEBM (Faux)

---

FIGURE 15. show()

```
sage : polar_plot(cos(2*x)*sin(3*x), (x, 0, pi), fill=True, \
épaisseur=2, fillcolor='jaune', \ color='verge
d'or', axes=False)
```



Animation

L'animation, tant au cinéma qu'en programmation informatique, consiste à échanger plusieurs images plusieurs fois par seconde. Sage propose une commande `animate()` qui accepte une collection en entrée et crée une animation que nous pouvons ensuite afficher (`show()`). Dans ce cas, les options fournies à `show()` sont légèrement différentes.

Il est important de noter que nous ne spécifions pas les options `show()` traditionnelles `xmin`, `xmax`, etc. lors de l'affichage d'une animation ; elles sont plus appropriées dans la commande `animate()`, qui doit regrouper toutes les images et donc connaître les valeurs `x` et `y` maximales. Les options attendues par `show()` s'appliquent à l'affichage d'une animation et ne sont pas spécifiques à l'animation elle-même. Autrement dit, il est logique d'afficher la même animation avec un délai différent entre les images, car les images elles-mêmes ne changent pas.

cela n'a aucun sens de dire que la « même » animation aurait une valeur `xmin` différente ; les images elles-mêmes ont changé dans ce cas.

Pour illustrer le fonctionnement de la commande `animate()`, nous allons jouer avec notre fleur de lys de la section précédente.

```
sage: p1 = polar_plot(cos(2*x)*sin(3*x), (x, 0, pi), fill=True, \ thickness=2,
                     fillcolor='yellow', \ color='goldenrod',
                     axes=False) sage: p2 =
polar_plot(cos(3*x)*sin(4*x), (x, 0, pi), fill=True, \ thickness=2, fillcolor='yellow', \
           color='goldenrod', axes=False) sage: p3 =
polar_plot(cos(4*x)*sin(5*x), (x, 0, pi),
fill=True, \ thickness=2, fillcolor='yellow', \ color='goldenrod', axes=False) sage: p4 =
polar_plot(cos(5*x)*sin(6*x), (x, 0, pi),
fill=True, \ thickness=2, sage: p5 =
polar_plot(cos(6*x)*sin(7*x), (x, 0, pi), fill=True, \ thickness=2, fillcolor='yellow', \
           color='goldenrod', axes=False) sage : p6 =
polar_plot(cos(7*x)*sin(8*x), (x, 0, pi),
fill=True, \ thickness=2, fillcolor='yellow', \ color='goldenrod', axes=False) sage: panim
= animate((p1, p2, p3, p4, p5, p6), xmin=-1,
          xmax=1, \ ymin=-0.5, ymax=1.5,
          aspect_ratio=1)

sage : show(panim, gif=True)
```

Vous devriez obtenir une animation. Si vous consultez ce texte dans Acrobat Reader, vous devriez voir l'animation ci-dessous ; si vous consultez une version papier, vous devriez voir les images individuelles de l'animation :

Prenez le temps de tester les options. Par exemple, voyez ce qui se passe lorsque vous remplacez la dernière ligne de cette séquence par

```
sage : show(panim, gif=True, delay=10)
```

ou à

```
sage : show(panim, gif=True, delay=40)
```

Vous devriez remarquer un changement de comportement certain.

Vous pouvez également essayer de supprimer l' option `gif=True` . Nous l'avons utilisée car certains navigateurs web courants ne prennent pas en charge le format WEBM au moment de la rédaction de cet article. Voyez ce qui se passe après sa suppression.

L'approche présentée ici complique légèrement la création de tracés, sauf pour les animations les plus simples. Dans un chapitre ultérieur, nous verrons comment utiliser des boucles pour accélérer le processus.

### Intrigues implicites

Certains tracés cartésiens sont facilement décrits comme des relations en termes de  $x$  et  $y$ , mais sont difficiles à décrire comme des fonctions. Géométriquement, ils ne satisfont pas au « test de la droite verticale » qui détermine une fonction, et il est difficile de les réécrire sous forme paramétrique ou polaire.

Prenons l'exemple du cercle. Une relation qui n'illustre pas bien cette difficulté est celle du cercle où  $a$  est une constante  $2x^2 + y^2 = a$ , quelconque (dans la limite du raisonnable). On peut exprimer cette courbe  $+ y$  avec un paramètre  $t$  :

$$(x, y) = (\text{un coût}, \text{un sin } t)$$

ou en utilisant les coordonnées polaires :

$$r = a.$$

Néanmoins, l'équation traditionnelle d'un cercle est utile à cet égard : elle n'admet pas de fonction  $y$  en fonction de  $x$  ; après tout, si nous résolvons pour  $y$ , nous avons

$$y = \pm \sqrt{a^2 - x^2},$$

et avoir le choix entre deux valeurs  $y$  est un peu décevant pour une fonction. C'est pour le moins gênant.

Une solution consiste à la tracer à l'aide de diagrammes paramétriques ou polaires, mais cette méthode n'est pas toujours disponible, ou difficile à utiliser. Une autre solution consiste à tracer la courbe implicitement ; dans ce cas, vous spécifiez une équation, ses variables et leur étendue. Sage transforme ensuite l'équation.

$f(x, y) = g(x, y)$  à  $f(x, y) - g(x, y) = 0$ , construit une grille le

long de la fenêtre que vous avez spécifiée, vérifie quels points de la grille se rapprochent raisonnablement de zéro, puis les connecte de manière intelligente pour obtenir le graphique correct de la fonction.

```
sage: var('y') sage:
implicit_plot(x**2 + y**2 == 6, (x, -3, 3), (y, -3, 3),
              (fill=True))
```

---

« Dans la limite du raisonnable ? » vous murmurez-vous probablement. « Qu'est-ce que ça veut dire ? » Nous voulons dire  $a = 0$ . Et fini. Et Réel, bien réel. Complex seraient mauvais, du moins pour nos besoins. Donc, fondamentalement,  $a \neq \{0\}$ .

---

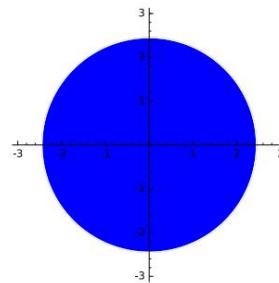
```
implicit_plot( relation, (xvar,xmin,xmax) , (yvar,ymin,ymax) ,options)
```

- la relation est une équation en termes de deux variables •
- xvar et yvar sont les indéterminées de la fonction pour laquelle vous souhaitez tracer le long des axes x et y, respectivement
- xmin et xmax sont les valeurs x les plus petites et les plus grandes à utiliser pour générer des points • ymin et ymax sont les valeurs y les plus petites et les plus grandes à utiliser pour générer des points • les options incluent – fill = s'il faut remplir « l'intérieur » de la relation ; c'est-à-dire les paires (x, y) où la relation transformée devient négative –
- plot\_points = nombre de points à utiliser sur chaque axe lors de la génération du tracé (150)

Notez que fillcolor, fillalpha et thickness ne sont pas des options valides pour implicit\_plot().

---

FIGURE 16. implicit\_plot()



Expliquons comment Sage a déterminé les points situés à l'intérieur. Nous lui demandons de représenter graphiquement l'équation.

$$2x + et^2 = 6,$$

donc

$$f(x, y) = x^2 + et^2 \quad \text{et} \quad g(x, y) = 6.$$

Sage réécrit cela comme

$$f(x, y) - g(x, y) = 0,$$

ou plus précisément

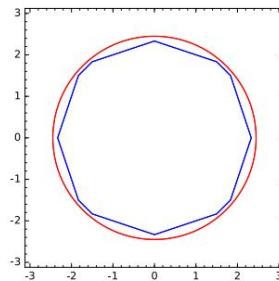
$$2x + et^2 - 6 = 0.$$

Sage divise ensuite les intervalles x et y (-3, 3) en 150 sous-intervalles, ce qui crée une grille sur le plan, et substitue chaque paire (x, y) de la grille dans le côté gauche de l'équation. Certains de ces points peuvent correspondre exactement à 0, mais la plupart ne le seront pas ; par exemple, (1,5, 1,92) et (1,5, 1,96) sont proches du cercle, mais le premier donne une valeur négative (-0,0636) lorsqu'il est substitué dans  $x^2 + y^2 - 6$ , tandis que le second donne une valeur positive (0,0916). Cela indique trois choses à Sage : • Le cercle doit passer entre ces deux

points ; après tout, ses points ont la valeur 0 lorsque  $x^2 + y^2 = 6$ . • De ces deux points, le point (1,5, 1,96) se trouve plus substitué dans  $x^2 + y^2 - 6$  près de 0. • Parce que sa valeur était négative, le point (1,5, 1,96) se trouve à l'intérieur du cercle et devrait être rempli si fill=True.

Pour voir cela en action, nous pouvons tester l' option `plot_points` . Nous allons tracer deux cercles vides, chacun avec un nombre de points de tracé très différent.

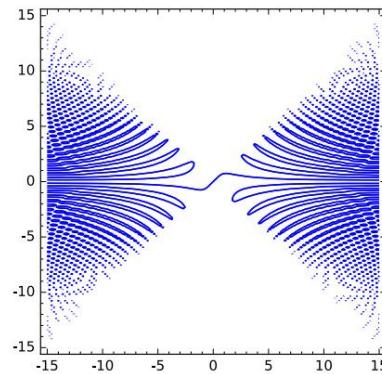
```
sage : p1 = implicit_plot(x^2+y^2-6, (-3,3), (-3,3), \
                           plot_points=5, zorder=5)
sage : p2 = implicit_plot(x^2+y^2-6, (-3,3), (-3,3), \
                           couleur='rouge', zorder=-5, \
                           plot_points=300)
sage : p1 + p2
```



Le graphique bleu, dont les axes ne comptaient que six points (les extrémités du sous-intervalle ajoutent 1 à l' option `plot_points` ), semble irrégulier et irrégulier, tandis que le graphique rouge semble lisse, malgré sa génération identique. Cette illusion de lissage est due au nombre beaucoup plus élevé de points connectés pour former le graphique.

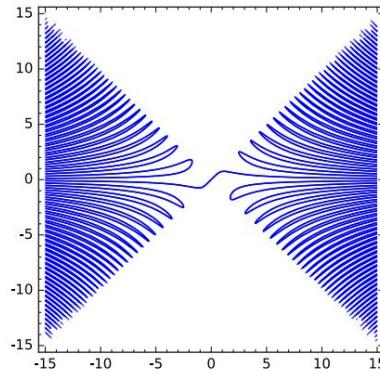
Pour bien comprendre l'importance du nombre de points de l'intrigue dans des situations pratiques, examinons un autre exemple.

```
sage : implicit_plot(x*cos(x*y)-y, (x,-15,15), (y,-15,15))
```



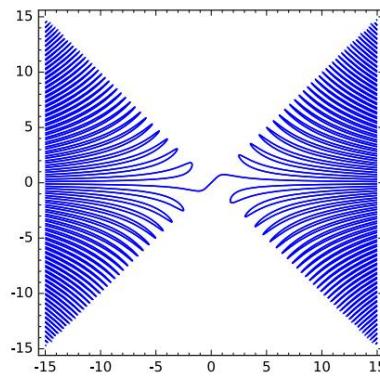
Cette courbe a l'air tout simplement... impressionnante. Mais s'agit-il d'une courbe unique et connectée ? Elle semble comporter de nombreux éléments irréguliers. Certaines parties sont également déconnectées ; ce n'est pas un problème en soi, mais, avec l'irrégularité, cela suggère un problème. Dans ce cas, on pourrait s'attendre à ce qu'une augmentation du nombre de points de tracé soit extrêmement utile, et c'est le cas. Si vous augmentez le nombre de points de tracé d'environ 50, jusqu'à ce que les choses deviennent claires, vous obtenez finalement quelque chose comme ceci :

```
sage : implicit_plot(x*cos(x*y)-y, (x,-20,20), (y,-20,20), \
plot_points=500)
```



Il y a une sacrée différence dans cette image, n'est-ce pas ? Et surtout, il y a une sacrée différence dans le temps qu'il a fallu pour les représenter, n'est-ce pas ? Certains points restent flous, alors allons plus loin :

```
sage : implicit_plot(x*cos(x*y)-y, (x,-20,20), (y,-20,20), \
plot_points=1000)
```



Notez le compromis : plus de points d'intrigue résolvent plus de divergences, mais conduisent également à un temps plus long. À ce stade, il n'y a aucune raison d'augmenter `plot_points` plus haut, car il n'y a aucune raison de penser que nous avons perdu des fonctionnalités intéressantes.

Nous concluons par quelques avertissements. Il peut arriver que vous oubliez de spécifier `xvar` et `yvar` lors de la création de votre tracé implicite ; c'est une erreur assez naturelle, d'autant plus que `plot()` ne nécessite pas la spécification d'une variable. Dans ce cas, vous recevrez un avertissement de dépréciation :

```
sage : implicit_plot(x**2 + y**2 - 6, (-3,3), (-3,3))
```

Avertissement : les plages sans nom pour plusieurs variables sont obsolètes et seront supprimées d'une future version de Sage ; vous pouvez utiliser des plages nommées à la place, comme `(x,0,2)`

L'avertissement ici est à la fois explicite et utile : il indique le problème et propose une solution. L'utilisation correcte est celle décrite ci-dessus, et nous la répétons ici pour plus de clarté :

```
sage : implicit_plot(x**2 + y**2 - 6, (x,-3,3), (y,-3,3))
```

Vous rencontrerez également des problèmes si vous négligez de définir  $y$  comme une valeur indéterminée, ou si vous négligez de spécifier les plages pour  $x$  et  $y$  ; cela conduit à des avertissements NameError (car  $y$  est inconnu) et TypeError (car Sage attend un nombre différent d'arguments).

### Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Nous avons répertorié toutes les options possibles pour les commandes de traçage.
2. Vous n'êtes pas obligé de fournir des valeurs pour tous les arguments que nous listons pour une commande.
3. Les valeurs des couleurs RVB vont de 0 à 255.
4. Peu importe ce que vous faites, vous êtes coincé avec des axes dans chaque intrigue Sage.
5. Les options zorder permettent de placer les objets « au-dessus » ou « devant » les autres.
6. Une des raisons d'utiliser la commande `polygon()` au lieu de la commande `line()` est que vous ne  
il faut re-spécifier le point de départ comme point d'arrivée.
7. La transparence d'un remplissage peut différer de la transparence de la courbe à remplir.
8. Sage génère des tracés de la même manière que nous apprenons à le faire : en reliant les points entre les paires  $xy$ .
9. Comme une calculatrice graphique, Sage est incapable de détecter les endroits où se produisent des asymptotes.  
$$\frac{2}{x} = 3 = x$$
10. La meilleure façon de représenter graphiquement la courbe  $y = \sqrt{x+1}$  consiste à résoudre  $y$  en prenant la racine carrée, à  
tracer les branches positives et négatives séparément à l'aide d'une commande `plot()`, puis à les combiner à l'aide de  
l'addition, comme vous le faites sur une calculatrice graphique.
11. Bonus : Lorsque vous ne connaissez pas la réponse à une question Vrai/Faux, la meilleure stratégie est de  
répondez « Discutable », puis priez pour un crédit partiel.

Choix multiple.

1. Si les coordonnées  $x$  et  $y$  d'un objet sont des fonctions du temps, le meilleur choix pour tracer un graphique de son mouvement est probablement :
  - A. `plot()`
  - B. `implicit_plot()`
  - C. `paramétrique_tracé()`
  - D. `polar_plot()`
2. Si la distance d'un objet par rapport à un point central varie lorsqu'il tourne autour du centre, la meilleure  
le choix pour tracer un graphique de son mouvement est  
probablement : A. `plot()`  
 B. `implicit_plot()`  
 C. `paramétrique_tracé()`  
 D. `polar_plot()`
3. Si une fonction est bien définie par une variable, le meilleur choix pour tracer son graphique est probablement :
  - A. `plot()`
  - B. `implicit_plot()`
  - C. `paramétrique_tracé()`
  - D. `polar_plot()`
4. Pour modifier le degré de transparence d'un objet graphique, nous utilisons cette option :

- A. alpha B.
- opacité C.
- transparence D.
- zorder

5. Pour changer quel objet semble « au-dessus » ou « devant » un autre, nous utilisons cette option :

- A. alpha B.
- profondeur
- C. niveau
- D. ordre z

6. Quel aspect d'une intrigue pourrait suggérer que vous devez augmenter la valeur de `plot_points` ?

- A. Il semble qu'il y ait une asymptote.
- B. Des lignes dentelées ou des courbes « tordues » apparaissent lorsque vous attendez à une douceur.
- C. Une équation produit deux courbes ou plus.
- D. Nous avons une courbe approximative au lieu d'une courbe exacte.

7. Si un `plot()` produit une image dont une partie ressemble à une ligne verticale, quelle option devez-vous modifier pour tester une asymptote ?

- A. `detect_poles` B.
- `draw_asymptotes` C.
- `plot_points` D.
- zorder

8. Si vous obtenez une `NameError` lors de la création d'un `implicit_plot()`, alors, selon toute vraisemblance, vous avez créé le erreur suivante : A.

- Vous avez oublié de spécifier une équation en x et y, plutôt qu'une expression en x seul B. Vous avez mal orthographié le nom d'une option de `implicit_plot()`
- C. Vous avez oublié de définir l'indéterminé y en utilisant `var()`
  - D. Vous avez oublié de spécifier x et y comme `xvar` et `yvar`

9. Si vous obtenez une `TypeError` lors de la création d'un `implicit_plot()`, alors, selon toute vraisemblance, vous avez créé le erreur suivante : A.

- Vous avez mal orthographié le nom, `implicit_plot()`
- B. Vous avez mal orthographié le nom d'une option de `implicit_plot()`
  - C. Vous avez oublié de définir l'indéterminé y en utilisant `var()`
  - D. Vous avez oublié de spécifier les plages pour x et y

10. Si nous enseignons le calcul et que nous voulons illustrer comment trouver l'aire entre les courbes 2 et x sur [0, 1],

X quelle commande afficherait la région que nous voulons ? (Le tracé devrait également

inclure les deux courbes qui définissent la région.)

- A. `plot(x**2-x, 0, 1, fill=True)`
- B. `plot(x**2, 0, 1, fill=x)`
- C. `plot(x**2, 0, 1, fill=True) - plot(x, 0, 1, fill=True)`
- D. `plot(x**2, 0, 1, fill=x) + plot(x, 0, 1)`

Bonus : Vous êtes bloqué sur une île déserte. Lequel de ces objets préféreriez-vous avoir avec vous ?

- toi ?
- A. une serviette
  - B. un ouvre-boîte C.
  - une bouteille de crème solaire
  - D. un ordinateur portable fonctionnel avec Sage installé

Réponse courte.

- Le professeur Proofrock affirme que nous pouvons visualiser la courbe  $y = x^2$  comme une courbe paramétrique  $y = t^2$ ,  $x = t$ . Tracez les deux courbes en utilisant `plot()` pour la première et `parametric_plot()` pour la seconde. Êtes-vous d'accord avec cette moyenne ? Outre les tracés, qu'est-ce qui vous rend d'accord ?
- Pourquoi pensez-vous que les valeurs par défaut des options `show()` `ymax`, `ymin` et `aspect_ratio` dépendent de l'objet affiché plutôt que de valeurs spécifiques ? Donnez une brève explication pour chaque option.

Programmation.

- Utilisez Sage pour représenter graphiquement les fonctions suivantes sur l'intervalle  $[1, 5]$ . Attribuez une couleur différente à chacune d'elles et nommez-les avec un objet `text()`.

$$x^2, x, \log x, \text{ et } x$$

Classez les fonctions selon celle qui croît le plus rapidement à long terme. Utilisez ensuite les fonctions de calcul de Sage pour vérifier que votre classement est correct à  $x = 10$  et à  $x = 100$ . Indice : la dérivée indique le taux de variation en un point.

- En utilisant vos connaissances en calcul, procédez comme suit :

(a) Choisissez une fonction transcendante  $f(x)$  et un point  $x = a$  où la fonction est définie. (b) Calculez la dérivée de  $f$  en  $x = a$ . Si la dérivée est 0, revenez à l'étape (a) et choisissez un autre point. (c) Tracez  $f$  dans un voisinage de  $a$ . Rendez cette

courbe noire, d'épaisseur 2. (d) Ajoutez à cela un tracé de la droite tangente à  $f$  en  $x = a$ .

Rendez cette droite bleue et en pointillés. (e) Ajoutez à cela un tracé de la droite normale en  $x = a$ . Rendez cette droite verte et en pointillés.

Astuce : La droite normale passe par  $x = a$  et est perpendiculaire à la tangente. (f) Ajoutez un gros point rouge à  $(a, f(a))$ . (g) Ajoutez des étiquettes pour le point, la courbe et la droite. Utilisez LATEX. La couleur de chaque étiquette doit correspondre à celle de l'objet qu'elle étiquette. (h) affichez() l'image avec un rapport hauteur/largeur approprié pour un résultat agréable.

- Le théorème de la valeur moyenne du calcul stipule que si  $f$  est une fonction continue sur l'intervalle

$[a, b]$ , alors nous pouvons trouver un  $c \in (a, b)$  tel que

$$f(c) = \frac{1}{b-a} \int_a^b f(x) dx,$$

et cette valeur de  $y (f(c))$  est en fait une valeur « moyenne » de la fonction sur l'intervalle. Dans ce problème, vous illustrerez ce résultat. (a) Choisissez une fonction quadratique  $f(x)$ . Choisissez un intervalle raisonnable  $[a, b]$ . Tracez le graphique de  $f$  sur  $[a, b]$ , en remplaçant jusqu'aux valeurs minimale et maximale de  $y$ . (Techniquement, cela nécessite de créer deux tracés, puis de les combiner.) (b) Trouvez la valeur de  $c$  qui satisfait le théorème de la valeur moyenne. (Vous devrez le faire à la main, car nous n'avons pas encore abordé l'utilisation de Sage pour résoudre des problèmes. Il s'agit d'une équation quadratique, donc cela ne devrait pas être si difficile.)

(c) Ajoutez la droite  $y = f(c)$  à votre tracé rempli de  $f(x)$ . Ajoutez également une droite reliant  $(c, 0)$  à  $(c, f(c))$ .

Ces lignes doivent avoir une couleur différente de celle du tracé ou de son

remplissage. (d) Expliquez comment cette image finale illustre le théorème de la valeur moyenne.

- Un nombre complexe a la forme  $a + bi$ . Le plan complexe est une représentation du corps dans le plan cartésien, le point  $(a, b)$  correspondant au nombre  $a + bi$ .

(a) Choisissez trois nombres complexes  $a + bi$ , chaque point correspondant du plan complexe étant situé dans un quadrant différent. Créez des flèches reliant l'origine à chaque point, chaque flèche étant d'une couleur différente. (b) Utilisez Sage pour calculer le produit  $i(a + bi)$

pour chacun des points que vous venez de choisir. Créez trois nouvelles flèches reliant l'origine à chaque point, d'une couleur correspondant à celle du point initial. (c) Tracez les six flèches simultanément. Quel est l'effet géométrique de la multiplication de  $(a + bi)$  par

je?

5. Une grande partie des mathématiques supérieures s'intéresse aux courbes elliptiques, qui sont des courbes lisses, non auto-sécantes, de forme générale

$$\begin{aligned} 2x^3 &= y \\ \text{et } & \quad + b \end{aligned}$$

Représentez graphiquement plusieurs courbes de cette forme, en utilisant différentes valeurs pour  $a$  et  $b$ . Quelles valeurs produisent des courbes elliptiques ? Quelles valeurs ne produisent pas de courbes elliptiques ? Décrivez le schéma général de ce qui se produit lorsque les valeurs de  $a$  et  $b$  varient.

6. Nous poursuivons sur le problème précédent. De nombreuses mathématiques supérieures considèrent les courbes elliptiques sur un corps fini, par exemple lorsque  $p$  est premier. Choisissez une des courbes elliptiques du problème précédent. Soit  $p = 5$  et remplacez toutes les valeurs possibles de  $x$  et  $y$  dans l'équation.

(Il n'y en a que 25, donc ce n'est pas trop compliqué, et vous pouvez probablement en faire beaucoup dans votre tête. N'oubliez pas de les faire modulo 5.) Tracez les points dans le plan. Cette courbe ressemble-t-elle à celle que vous aviez auparavant ?

Bonus : Pourquoi avons-nous pu affirmer avec certitude que dans l'équation n° 6, il y avait 25 valeurs possibles de  $x$  et  $y$  à substituer ?

Et si nous avions choisi  $p = 7$  ? Quel aurait été le nombre de points ?

## CHAPITRE 4

## Définir vos propres procédures

Résoudre des problèmes plus complexes nécessite bien plus qu'une simple liste de commandes. Il est souvent utile d'organiser de nombreuses tâches souvent réutilisées en une seule commande, ce qui permet de les maîtriser toutes. Cela nous permet également de traiter les tâches de manière plus abstraite, en nous aidant à voir la forêt plutôt que les arbres. Cela facilite la réutilisation du même code à différents endroits, et permet également à quelqu'un d'autre de le lire et, si nécessaire, de le modifier à ses propres fins. Cette idée de modularité nous rend plus « productifs », dans le sens où nous devons de meilleurs résolveurs de problèmes.

Techniquement, vous l'avez déjà fait : chaque commande Sage que vous utilisez pour simplifier des expressions et dessiner des images est en réalité un ensemble d'autres commandes conçues (généralement avec soin) pour cette tâche spécifique. Vous pouvez d'ailleurs visualiser une partie du code de la plupart des commandes Sage à l'aide d'un double point d'interrogation :

```
sage : simplifier ???
Signature : simplifier(f)
Source:
def simplify(f): r"""

```

Simplifiez l'expression « f ».

**EXEMPLES** : Nous simplifions l'expression «  $i + x - x$  ».

::

```
sage : f = i + x - x ; simplifier(f)
je
```

En fait, l'impression de « f » donne la même chose, c'est-à-dire la forme simplifiée.

essayer:

```
    renvoie f.simplify()
sauf AttributeError :
    retour f
```

Fichier : /Applications/sage-6.7-untouched/local/lib/python2.7/site-packages/sage/calculus/functional.py Type : fonction

En fait, cela nous donne un peu plus que le code. Il répertorie d'abord la signature de la commande, un modèle d'utilisation. Il répertorie ensuite le « code source », la liste des commandes qui définissent la commande `simplify()`. Ces commandes commencent après la ligne `Source:` et continuent jusqu'à la ligne `File:` ; cette dernière ligne indique le fichier sur le disque où se trouve la commande. Si vous ouvriez ce fichier, vous y trouveriez la même source que celle indiquée ci-dessus. Notez que la majeure partie du code source dans ce cas est constituée de la même documentation que celle affichée lorsque vous , ce qui indique que la documentation de Sage est intégrée au code lui-même. type `simplify?`

Une grande partie du reste de ce texte est consacrée à vous aider à comprendre la signification de tout ce code source et à l'utiliser à vos propres fins. Nous n'attendons pas de vous que vous deveniez un développeur Sage à part entière à la fin de votre formation (cela dépasserait largement son objectif), mais nous attendons de vous que vous soyez capable d'écrire de nouvelles commandes simples permettant d'accomplir des tâches spécifiques utiles à la recherche et à l'enseignement. Cela exige un certain dévouement de votre part, mais rassurez-vous : cela exige également le nôtre, et nous nous y attelons maintenant.

### Définir une procédure

Les commandes définies dans Sage sont souvent appelées fonctions, ce qui correspond à leur utilisation mathématique : elles fournissent des arguments et un résultat. Cependant, une fonction Sage et une fonction mathématique sont deux choses différentes. Par conséquent, pour plus de clarté, nous utiliserons le terme « procédure » pour désigner une nouvelle commande définie dans Sage.

Format de base. Pour définir une procédure, nous utilisons le format suivant :

```
sage : def nom_procédure(arguments) : première_instruction
                    deuxième_déclaration
                    ...

```

On peut considérer `def` comme l'abréviation de « `define` ». C'est un exemple de mot-clé ou de mot réservé, une séquence de caractères ayant une signification particulière dans un langage de programmation, et à laquelle le programmeur ne peut attribuer aucune autre signification. Ce qui suit est un nom d'identifiant valide (revoyez les règles p. 31) : c'est le nom de votre nouvelle procédure. Vous devez ensuite ouvrir et fermer les parenthèses. Vous pouvez éventuellement inclure des identifiants supplémentaires entre les parenthèses ; bien que ce ne soit pas obligatoire, c'est le moyen le plus efficace et le plus fiable de transmettre des informations à votre procédure. Nous appelons ces identifiants des <sup>1</sup> et je les examinerai plus attentivement dans un instant.

arguments. Il convient ensuite de noter que les « instructions » sont indentées. Il s'agit d'un principe important que Sage hérite de Python, dont vous vous souviendrez qu'il s'agit du langage de programmation que nous utilisons pour interagir avec Sage : chaque fois que nous arrivons à un point dans un programme où nous devons organiser des instructions « à l'intérieur » d'autres instructions, ou lorsque certaines commandes « dépendent » d'autres, nous indiquons quelles instructions dépendent des autres en utilisant l'indentation.<sup>2</sup> Cette configuration est appelée structure de contrôle ; ici, la structure de contrôle est l' instruction `def` , qui définit une procédure, qui, pour nos besoins, peut être considérée comme un raccourci pour une liste d'autres instructions.

1. Un nom bien choisi, car écrire un programme ressemble souvent à une dispute avec l'ordinateur. Textes d'informatique appellent souvent les arguments « paramètres ».

La plupart des langages informatiques ne nécessitent pas d'indentation, mais utilisent des mots-clés ou des caractères spéciaux pour indiquer le début et la fin d'une structure de contrôle. Par exemple, de nombreux langages suivent le C dans l'utilisation des accolades { ... } pour ouvrir et fermer une structure ; d'autres utilisent des variantes de BEGIN... END.

Pour l'instant, examinons une procédure assez simple. Elle ne nécessite aucun argument, mais n'est pas totalement inutile.

```
sage : def salutations():
    print('Salutations !')
```

Vous pouvez utiliser des guillemets simples ou doubles autour du mot « Salutations », mais assurez-vous de le faire de manière cohérente.

Si vous avez saisi correctement ce message, Sage l'acceptera automatiquement. Si vous saisissez une erreur, Sage le détecte et signale une erreur. Par exemple, si vous oubliez d'indenter, vous verrez ceci :

```
sage : def greetings():
    print("Salutations !")
Fichier "<ipython-input-33-33dcc1adcd0e>", ligne 2
    print("Salutations !")
^
IndentationError : un bloc indenté était attendu

Si vous souhaitez coller du code dans IPython, essayez les fonctions magiques %paste et
%cpaste.
```

Il est peu probable que vous voyiez une erreur d'indentation si vous tapez votre programme dans une feuille de calcul ou sur la ligne de commande, car Sage effectue automatiquement l'indentation pour vous lorsque vous devriez le faire.

Voici une autre erreur que vous pourriez voir :

```
sage : définition salutations()
Fichier « <ipython-input-34-583f0b909fc6> », ligne 1
    salutations def()
^
SyntaxError : syntaxe non valide
```

Voyez-vous le problème ici ? Sinon, prenez quelques minutes pour comparer cette tentative avec celle qui a réussi. Observez particulièrement la fin de la ligne, là où pointe le symbole du carat ^ . C'est à cet endroit que Sage rencontre ses problèmes.

Revenons maintenant à l'hypothèse selon laquelle vous avez défini la procédure avec succès. Vous pouvez utiliser votre nouvelle procédure en la tapant, suivie d'une paire de parenthèses, et en exécutant la ligne.

```
sage : salutations()
Salutations!
```

Sage a exécuté votre demande. Il a recherché la procédure greetings(), a constaté qu'elle devait imprimer (« Greetings ! ») et a exécuté la demande.

Au passage. La signification de la commande print est, je l'espère, évidente : elle imprime ce qui suit.

Dans les anciennes versions de Sage, printwas était le mot-clé ; vous ne pouviez pas lui attribuer une nouvelle valeur :

```
sage : print = 2
Fichier « <ipython-input-38-7c34f307b821> », ligne 1 print = Integer(2)

^

SyntaxError : syntaxe non valide
```

Mais aujourd'hui, l'impression est l'une des « procédures standard » de Sage ; un nom que vous pouvez réattribuer à des fins plutôt malheureuses si vous ne faites pas attention :

```
sage : imprimer = 2
sage : imprimer 2

sage : imprimer(3)
TypeError Traceback (dernier appel le plus récent) <ipython-input-8-
fd56b57ba354> dans <module>() ----> 1 print(Integer(3))

TypeError : l'objet « sage.rings.integer.Integer » n'est pas itérable
```

Si vous redéfinissez par inadvertance une procédure standard, n'oubliez pas que la commande de réinitialisation la ramènera à sa signification d'origine :

```
sage : réinitialiser("imprimer")
sage : imprimer(3) 3
```

Malheureusement, l'impression ne fonctionne pas avec LATEX, vous ne pouvez donc pas générer de belles sorties de cette façon. pour ce faire, il faut utiliser LATEX à partir d'une cellule HTML (voir p. 26), ou créer un tracé.

Autre remarque : les programmeurs laissent souvent des commentaires dans le code pour expliquer le fonctionnement d'une procédure. Dans Sage, la principale méthode pour ce faire est d'utiliser le symbole # ; lorsque Sage voit ce symbole, il ignore tout ce qui suit. Par exemple, l'implémentation suivante de greetings() est équivalente à celle ci-dessus.

```
sage: def greetings(): # saluer
    l'utilisateur print("Salutations !")
```

Sage ignore simplement la ligne qui commence par le symbole # et passe immédiatement à la suivante un.

C'est une excellente idée de laisser des commentaires dans les procédures, surtout au début d'une séquence de lignes complexes. Les auteurs peuvent témoigner qu'il leur est arrivé une année d'écrire du code qui semblait évident au moment de la programmation, puis d'y revenir des semaines, des mois, voire des années plus tard, en se demandant : « Mais pourquoi ai-je fait ça ? » Il faut alors plusieurs minutes, heures, voire jours, à analyser le code pour en comprendre le fonctionnement. La plupart des procédures

Les passages que nous écrivons dans ce texte ne sont pas très longs, mais nous inclurons néanmoins des commentaires dans bon nombre d'entre eux afin d'illustrer le besoin.

### Arguments

Un argument est une variable qu'une procédure utilise pour recevoir des informations nécessaires ou simplement utiles. On peut considérer un argument comme un espace réservé aux données ; son nom n'existe qu'à l'intérieur de la procédure, et l'information est oubliée une fois celle-ci terminée. Un argument ne contient qu'une copie de l'information envoyée, et non l'information originale elle-même. On peut illustrer l'idée de base en modifiant la procédure de la section précédente :

```
sage : def salutations(nom) :
    print('Salutations,', nom)
```

Ici, name est un argument de greetings(). C'est également un argument obligatoire, car nous n'avons spécifié aucune valeur par défaut. Nous aborderons plus loin la différence entre arguments obligatoires et facultatifs ; pour l'instant, il suffit de comprendre que vous devez fournir une valeur pour les arguments obligatoires. Par exemple, l'utilisation suivante de greetings() est autorisée dans Sage :

```
sage : salutations('Pythagore')
Salutations, Pythagore
```

Que s'est-il passé ? Lorsque Sage lit la commande, il constate que vous souhaitez appeler greetings() avec un argument, la chaîne « Pythagoras ». Il copie cette information dans la variable, puis exécute toutes les commandes indentées sous nom , la définition de greetings().

D'autre part, maintenant que nous avons redéfini greetings(), l'utilisation suivante donne une erreur, même si cela fonctionnait bien auparavant :

```
sage : salutations()
TypeError Traceback (dernier appel le plus récent)
<ipython-input-44-deb3f0bd6096> dans <module>() ----> 1
greetings()

TypeError : greetings() prend exactement 1 argument (0 donné)
```

Prenons un exemple précis : l'équation d'une tangente. Nous allons nous concentrer sur un usage particulier : supposons que vous souhaitez écrire une procédure qui calcule automatiquement l'équation en un point  $x = a$  d'une droite tangente à une courbe définie par une fonction  $f(x)$ . Rappelons que l'équation d'une droite de pente  $m$  et passant par un point  $(a, b)$  est de la forme

$$y - b = m(x - a),$$

que nous pouvons réécrire comme

$$y = m(x - a) + b.$$

Dans ce cas,  $m$  est la dérivée de  $f$  en  $x = a$  ; ou,  $m = f(a)$ . Nous pouvons donc réécrire l'équation comme  $y = f(a) \cdot (x - a) + b$ .

Nous voulons une procédure qui produit une telle équation, ou au moins le côté droit de celle-ci.

Pour définir cette procédure, nous devons répondre à plusieurs questions :

- De

quelles informations la procédure a-t-elle besoin ?

Nous avons besoin de la fonction  $f(x)$  et de la valeur  $x = a$ .

- 

Comment utilisons-nous ces informations pour obtenir le résultat souhaité ?

– Il faut d'abord calculer  $f(a)$ . Cela nécessite de calculer  $f(x)$ , puis de substituer  $x = a$ .

– Il faut aussi calculer  $b$ . C'est la valeur de  $y$  à  $x = a$ , il suffit donc de calculer

$f(a)$  ; c'est-à-dire en substituant  $x = a$  dans  $f(x)$ .

- 

Comment communiquons-nous le résultat au client ?

Pour l'instant, nous allons simplement l'imprimer, mais nous discuterons éventuellement d'une meilleure façon de procéder.

En rassemblant ces informations, nous arrivons à la définition suivante :

```
sage : définition de la ligne_tangente(f, a) :
# forme point-pente d'une droite b = f(a)
df(x) =
diff(f,x) m = df(a) result
= m*(x - a)
+ b print('La droite tangente à',
f, 'en x =', a, 'est',) print(result, '.)')
```

Mettons cela en pratique. Un exemple simple, vérifiable à la main, est l'équation de  $x = 1$ .  
une droite tangente à  $y = x^2$

```
sage : tangente_ligne(x**2, 1)
/Applications/sage-6.7-untouched/src/bin/sage-ipython:1: Avertissement : la
substitution utilisant la syntaxe d'appel de fonction et des arguments sans nom est
obsoète et sera supprimée d'une future version de Sage ; vous pouvez utiliser des
arguments nommés à la place, comme EXPR(x=..., y=...)
```

Voir <http://trac.sagemath.org/5930> pour plus de détails. #!/usr/bin/

env python La ligne tangente

à  $x^2$  à  $x = 1$  est  $2*x - 1$ .

Oups ! Nous avons de nouveau rencontré cet avertissement d'obsolescence. Ce problème se produit une fois par session Sage, et nous obtenons finalement la bonne réponse ( $2*x - 1$ ).

Malgré tout, il faudrait le modifier. Mais où cela se produit-il ? Il faut vérifier chaque énoncé de la procédure :

- $df(x) = diff(f,x)$

Cela ne peut pas être le problème, car cela définit une procédure.

- $m = df(a)$

Cette syntaxe d'appel de fonction utilise  $df$  ; elle est donc susceptible de poser problème. Cependant, elle ne peut pas être le problème, car nous avons défini  $df$  comme une fonction. Il est donc approprié d'utiliser la syntaxe d'appel de fonction avec les fonctions.

- $b = f(a)$

Cette fonction utilise  $f$  dans sa syntaxe d'appel ; elle est donc susceptible de poser problème. Comme  $f$  est un argument dans lequel Sage a copié la valeur fournie lors de l'appel de `tangent_line()`, nous devons vérifier si c'est bien le problème. Nous l'avons appelée avec la commande :

```
tangente_ligne(x**2, 1)
```

Cela doit être le problème, car  $x^{**2}$  est une expression, pas une fonction.

Une façon d'éviter cet avertissement est de définir une fonction avant d'appeler `tangent_line()`, comme suit :

```
sage : f(x) = x**2 sage :
tangente_ligne(f, 1)
La droite tangente à x |--> x^2 à x = 1 est 2*x - 1 .
```

Ce n'est cependant pas très pratique, car cela nécessite beaucoup plus de saisie. De plus, cela impose une condition au client.

Une bien meilleure solution consiste à ajuster le programme lui-même, afin qu'il ne suppose pas qu'il reçoit une fonction alors qu'en fait, ce n'est pas le cas.<sup>3</sup> Nous pouvons le faire en modifiant la ligne  $b = f(a)$  pour utiliser une substitution explicite,  $b = f(x=a)$ . Essayons cela.

```
sage : définition de la ligne_tangente(f, a) :
# forme point-pente d'une droite b =
f(x=a) df(x) =
diff(f,x) m = df(a) result
= m*(x - a)
+ b print('La droite tangente à',
f, 'en x =', a, 'est',) print(result, '.')
```

Maintenant, nous essayons de l'exécuter :

```
sage : tangente_ligne(x^2, 1)
La droite tangente à x^2 à x = 1 est 2*x - 1 .
```

Comme un avertissement d'obsolescence n'apparaît qu'une seule fois, il n'est pas toujours évident que nous ayons résolu le problème, car Sage pourrait le supprimer à nouveau. Vous pouvez le vérifier en redémarrant Sage : en mode feuille de calcul, cliquez sur le bouton Redémarrer ; sur la ligne de commande, saisissez `exit`, puis redémarrez Sage.

Arguments optionnels. Sage autorise également les arguments optionnels. Dans ce cas, il est souvent judicieux de fournir une valeur par défaut pour un argument. Ainsi, vous n'avez pas besoin de fournir de valeurs « évidentes » en général ; vous pouvez les fournir uniquement lorsque cela est nécessaire.

---

Si vous êtes familier avec les langages typés tels que C, Java ou Eiffel, vous remarquerez qu'il s'agit d'un cas où l'utilisation de types, souvent trop contraignante, est d'une grande aide pour le programmeur. Non seulement Sage n'exige pas de types, mais il n'offre aucune fonctionnalité pour les utiliser, sauf si vous utilisez Cython. Nous abordons Cython à la fin du texte.

Vous avez déjà rencontré ce problème. Consultez, par exemple, les figures 1 et 4 de la page 53. Elles parlent toutes deux d'options, qui sont en réalité des arguments optionnels des procédures. Par exemple, vous pouvez saisir

```
sage : point((3,2))
```

ou vous pouvez taper

```
sage : point((3,2), pointsize=10, couleur='bleu', alpha=1, zorder=0)
```

car les deux signifient la même chose.

Cet exemple illustre également comment les arguments facultatifs permettent au client de se concentrer sur le problème. Souvent, la couleur, la taille, etc. d'un point importent peu ; on souhaite simplement savoir où il se situe, généralement par rapport à un autre élément. Si cela s'avère utile, vous pouvez spécifier certaines de ces options. Par exemple, vous pouvez observer plusieurs points sur une courbe et les colorier différemment pour les distinguer. Mais cela peut être totalement inutile, et même si vous souhaitez spécifier la couleur, la taille précise, la transparence et l'ordre Z peuvent rester sans importance. Rendre ces arguments facultatifs signifie que vous n'avez pas besoin de les spécifier.

Comment nommer un argument optionnel lors de l'écriture de vos propres procédures ? Après le nom de la variable, placez un signe égal, suivi de la valeur par défaut. Voyons comment procéder dans notre procédure pour saluer quelqu'un : nous souhaiterions par exemple que la valeur par défaut soit « Leonhard Euler ».

```
sage : def greetings(name='Leonhard Euler'):
    print('Salutations,', name)
```

Le nom de l'argument est désormais facultatif. Comme dans la version précédente, nous pouvons désormais fournir un nom :

```
sage : salutations('Pythagore')
Salutations, Pythagore
```

... et vous pouvez également nommer explicitement l'argument facultatif (ceci est utile lorsqu'il existe plusieurs arguments facultatifs et que vous ne voulez pas vous soucier de les lister dans le bon ordre) :

```
sage : salutations(nom='Pythagore')
Salutations, Pythagore
```

... mais contrairement à la version précédente, nous pouvons utiliser la procédure sans valeur par défaut :

```
sage : salutations()
Salutations, Leonhard Euler
```

Fournir des indéterminés comme arguments par défaut. Il est souvent utile de fournir une indéterminée comme argument par défaut. Pour un exemple, revenons à notre exemple avec les tangentes. Cela fonctionne très bien si l'expression est définie en termes de  $x$ , mais de nombreuses expressions sont plus appropriées.

défini en termes d'une autre indéterminée, telle que  $t$ . Notre procédure calculera-t-elle toujours les lignes tangentes pour ces expressions ?

```
sage : var('t') sage :
tangent_line(t**2, 1)
La droite tangente à  $t^2$  à  $x = 1$  est  $t^2$ .
```

Ce n'est absolument pas correct. La réponse aurait dû être  $2t - 1$ , et non  $t$ . Plusieurs raisons : •  $df(x)$

=  $diff(f, x)$  différencie  $f$  par rapport à  $x$ , et non par rapport à  $t$ . Le résultat est que  $df(x)$  est égal à 0, et non à  $2t$ . •  $m = df(a)$  fonctionne avec une valeur incorrecte de  $df(x)$ , ce qui donne  $m = 0$ .

de  $m = 2$ . •  $b = f(x=a)$  demande à Sage de substituer  $a$  à la place de  $x$ , et non à la place de  $t$ . La substitution plutôt ne fait effectivement rien, avec pour résultat que  $b$  finit par être  $t^2$ , que 1. • Assemblez le tout avec  $m*(x - a) - b$  et il devient clair pourquoi Sage renvoie  $t^2$  à la place de  $2*t - 1$ .

Une solution serait d'écrire une nouvelle procédure qui fonctionnerait par rapport à  $t$  plutôt qu'à  $x$ . C'est une idée déplorable : le nombre d'identifiants possibles est considérable ; outre les 52 lettres majuscules et minuscules de l'alphabet latin, on peut aussi combiner ces lettres avec des chiffres et le trait de soulignement. Tous les noms d'indéterminés suivants pourraient être utilisés dans un contexte mathématique parfaitement légitime :

- $x$
- $x_0$
- $x_{\_0}$
- $x_i$
- $x_{\_i}$

... et ce ne sont que des indéterminés qui commencent par  $x$ . Vous pouvez jouer à ce jeu avec  $y, t, a, \dots$  vous voyez l'idée.

Cette approche est donc peu pratique. La meilleure façon de procéder est de spécifier l'indéterminé comme argument de la procédure elle-même, par exemple  $tangent_line(f, 1, t)$ . Il serait encore mieux de faire de l'indéterminé un argument facultatif, avec la valeur par défaut  $x$ , afin que le client n'ait à le spécifier que lorsque cela est nécessaire. Ainsi, les instructions  $tangent_line(f, 1)$  et  $tangent_line(f, 1, x)$  ont la même signification. Une approche naïve serait la suivante :

```
sage: def tangent_line(f, a, x=x): # forme point-
    pente d'une droite b = f(x=a) df(x) =
    diff(f,x) m =
    df(a) result = m*(x - a)
    + b print('La
    droite tangente à', f, 'à', x, '=', a, 'est',) print(result, '!')
```

L'instruction `x=x` peut paraître étrange, mais elle est parfaitement logique : le premier `x` spécifie le nom de l'argument de `tangent_line()`, tandis que le second `x` spécifie sa valeur par défaut. En résumé, elle indique que `tangent_line()` possède un argument optionnel nommé `x`, dont la valeur par défaut est `x`.<sup>4</sup>

Alors que se passe-t-il quand on l'essaie ?

```
sage : tangente_ligne(t**2, 1, t)
La droite tangente à t^2 à t = 1 est t^2 + 2*t - 2 .
```

## PANIQUE!

... Eh bien, pas de panique. Analysons plutôt ce qui s'est passé et tentons de résoudre le problème. Pour ce faire, nous introduisons une technique de débogage classique, appelée « désespoir ».

## imprimer

Qu'est-ce que cela signifie ? Nous imprimons tous les calculs de Sage dans la procédure et tentons d'identifier ce qui a mal tourné. Réécrivons la procédure comme suit :

```
sage: def tangent_line(f, a, x=x): # forme point-pente d'une
    droite b = f(x=a) print(b) df(x) = diff(f,x) print(df)
    m = df(a) print(m)
    result = m*(x
    - a) + b print('La droite
    tangente à', f,
    'à', x, '=', a,
    'est')
    print(result, '!')
```

Vous pouvez constater que nous avons une instruction `print` après presque chaque instruction originale. Nous avons laissé de côté le résultat, car il est déjà affiché sur la ligne suivante. Que se passe-t-il lorsque nous exécutons cette version de `tangent_line()` ?

```
sage : tangente_ligne(t**2, 1, t) t^2
t |--> 2*t^2
La droite tangente à t^2 à t = 1 est t^2 + 2*t - 2 .
```

Nous voyons que :

- `df` est correctement calculé comme  $2t$  ; •
- `m` est correctement calculé comme  $2$  ; mais
- `b` est incorrectement calculé comme  $t^2$  — comme si la substitution n'avait pas eu lieu ! Encore !

---

Ce type de construction se produit assez fréquemment dans le code source de Sage.

Notez la subtile différence cette fois : une substitution a effectivement fonctionné. L'autre, non. La différence réside dans le fait que l'une est une fonction mathématique définie dans notre procédure Sage ; l'autre est une expression définie en dehors de notre procédure Sage.

La solution apparente consiste à redéfinir `f` dans notre procédure, comme une fonction mathématique. Nous nous appuyons alors exclusivement sur la notation fonctionnelle. Cela ne présente aucun danger ; grâce aux règles de portée, modifier la valeur de `f` dans `tangent_line()` ne modifie pas sa valeur d'origine à l'extérieur. La procédure réussie ajoute une ligne :

```
sage: def tangent_line(f, a, x=x): # forme point-
    pente d'une droite f(x) = fb = f(a) df(x) =
    diff(f,x) m
    = df(a)
    result = m*(x - a) + b
    print('La
    droite tangente à', f, 'à', x, '=', a, 'est',) print(result, '!')
```

Cette première ligne redéfinit l'argument `f` comme une fonction. Si `f` est une expression, nous avons évité l'avertissement d'obsolescence ; s'il s'agit d'une fonction, Sage se l'attribuera sans difficulté. Essayez :

```
sage : tangente_ligne(t**2, 1, t)
La droite tangente à t |--> t^2 à t = 1 est 2*t - 1 .
```

Fin des dysfonctionnements de communication.

Jusqu'à présent, nous avons affiché le résultat d'un calcul à l'aide de la commande `print`. Bien que cela soit utile pour démontrer la commande `print` et afficher la sortie, cela constitue un obstacle à une automatisation efficace. Vous ne voulez pas que votre code attende que vous voyiez le résultat imprimé d'une procédure, puis le saisisse dans une autre procédure. Plutôt que de vous épuiser de cette façon, il serait bien mieux que les procédures communiquent directement. De cette façon, vous pourriez appeler une procédure à l'intérieur d'une autre et vous esquiver complètement. Vous pouvez démarrer un calcul, vous éloigner, prendre une tasse de caféine, puis revenir et constater que vos procédures ont terminé leurs tâches et vous attendent joyeusement, la queue qui remue.

Portée locale ou globale. Supposons, par exemple, que vous souhaitiez écrire une procédure qui trace une fonction mathématique  $f(x)$  et la tangente à  $f$  en ce point. Vous pourriez, bien sûr, lui demander de calculer la tangente en copiant-collant le code de `tangent_line()` dans la nouvelle procédure, mais cela gaspille des ressources et rend la nouvelle procédure plus difficile à lire. Vous pourriez plutôt utiliser la modularité en laissant `tangent_line()` calculer la tangente et en laissant votre procédure se concentrer uniquement sur le tracé de la fonction et de la droite. Mais comment la nouvelle procédure utiliserait-elle le résultat de `tangent_line()`? D'une part, `tangent_line()` possède une variable appelée `result`, donc peut-être pourrions-nous procéder de cette façon ?

---

5. À l'origine, cet article faisait référence à une boisson cafénée particulière, tout en attaquant une autre boisson plus populaire, car l'un des auteurs est un peu immature. Un deuxième auteur a commenté ZOMGPONIES et a ensuite convaincu le premier de modifier son discours pour adopter le discours générique actuel sur les boissons cafénées.

```
sage : def plot_tangent_line(f, a, x=x) : # utilisez
    tangent_line() pour trouver la ligne tangent_line(f, a,
    x) plot(result, a-1, b-1)
```

Cela ne fonctionnera pas, et si nous essayons, nous rencontrerions l'erreur suivante :

```
sage : plot_tangent_line(x^2, 1)
NameError : le nom global « result » n'est pas défini
```

Cette erreur indique que si le résultat existe quelque part, il n'existe pas dans l'environnement Sage « global ». Nous utilisons donc son nom inutilement. Nous avons défini le résultat uniquement dans la procédure `tangent_line()`, il est donc resté local à cette procédure.

Il existe deux solutions. La première consiste à déclarer `result` comme variable globale dans la procédure `tangent_line()` à l'aide du mot-clé `global` ; le résultat serait alors également accessible en dehors de cette variable.

```
sage : def tangent_line(f, a, x=x) : # forme point-
    pente d'une droite résultat global f(x) =
    fb = f(a) df(x) =
    diff(f,x) m
    = df(a)
    résultat = m*(x - a) + b
```

Cependant, c'est une très mauvaise idée, pour plusieurs raisons. Premièrement, « résultat » est le nom le moins informatif possible. Si chaque procédure plaçait son résultat dans une variable nommée « résultat », cela pourrait fonctionner, mais ce serait plutôt chaotique. De plus, il existe une meilleure solution.

Retour du chaos. Sage propose une commande pratique pour cela avec un autre mot-clé : `return`. Chaque procédure Sage renvoie une valeur, même si vous omettez ce mot-clé. Vous pouvez vérifier cela avec les procédures que nous avons définies, à l'aide d'une instruction `type()` :

```
sage : rien = tangent_line(f, 1, t)
La droite tangente à t |--> t^2 à x = 1 est 2*t - 1 . sage: type(rien)
'NoneType'
```

Dans ces cas, Sage renvoie un objet appelé `None`. Les programmeurs utilisent souvent `None`<sup>6</sup> pour indiquer qu'un élément n'a pas été initialisé, mais c'est un autre sujet.

Contrairement à d'autres langages informatiques, Sage ne vous contraint pas à renvoyer un seul résultat ; il est possible d'en renvoyer plusieurs simultanément. Il suffit de les lister après le mot-clé `return`, en les séparant.

<sup>6</sup>Ou l'analogue dans d'autres langues : `NULL`, `NIL`, ...

Séparez-les par des virgules. Par exemple, le programme suivant renverra la dérivée et la primitive d'une procédure, en supposant qu'elles existent toutes les deux :

```
sage : def deriv_and_integ(f, x=x):
    # BOOOO-RING
    df(x) = diff(f, x)
    F(x) = intégrale(f, x) renvoie
    df, F
```

Si cela vous rappelle comment Sage permet d'affecter plusieurs variables dans une même instruction (voir p. 30), vous avez raison ! Vous pouvez utiliser la procédure ci-dessus de la manière suivante :

```
sage : f(x) = x**2 sage :
df, F = deriv_and_integ(f) sage : df x |--> 2*x
sage : F x
|--> 1/3*x^3
```

Si vous avez déjà utilisé des langages qui ne le permettent pas, vous comprenez à quel point il peut être compliqué et peu intuitif de renvoyer plusieurs valeurs à la fois.<sup>7</sup>

Ce que nous voulons faire, c'est modifier la procédure `tangent_line()` pour renvoyer la ligne, plutôt que l'imprimez -le. C'est assez simple :

```
sage : def tangent_line(f, a, x=x) : # forme point-
        pente d'une droite f(x) = fb = f(a) df(x) =
        diff(f,x) m
        = df(a)
        result = m*(x - a) + b
```

résultat de retour

C'est aussi simple que ça. Que se passe-t-il quand on l'utilise ?

```
sage : tangente_ligne(t**2, 1, t)
2*t - 1
```

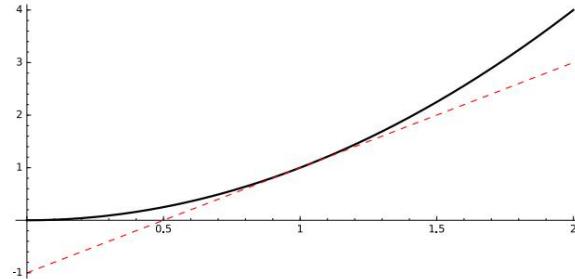
---

<sup>7</sup>Par exemple, en C++, on peut soit passer des arguments par référence, ce qui donne la signature  
void deriv\_and\_integ(Function f, Function & df, Function & F, Indeterminate x=x); ou créez une structure ou une classe qui contient des champs pour les deux réponses :

```
struct Dndl_result { Fonction df; Fonction F; };
Dndl_result deriv_and_integ(Fonction f, Indéterminé x=x);
mais l'approche Python semble plus élégante.
```

Le résultat peut paraître moins informatif qu'avant, mais c'est parce que nous l'avons réduit à son essence. Vous pourriez modifier l' instruction `return` pour inclure les informations textuelles dont nous disposions auparavant, mais l'essentiel est bien plus utile. Pourquoi ? Nous pouvons désormais effectuer les opérations suivantes :

```
sage : p1 = plot(t**2, 0, 2, couleur='noir', épaisseur=2) sage : par_line =
tangent_line(t**2, 1, t) sage : p2 = plot(par_line, 0, 2,
couleur='rouge', style de ligne='pointillé') sage : p1 + p2
```



Remarquez ce que nous faisons : nous prenons le résultat de `tangent_line()`, attribuons sa valeur à une variable nommée `par_line`, puis la passons en argument à `plot()`. Vous pouvez le faire même si vous n'aviez pas envisagé de tracer le résultat de `tangent_line()` ! En effet, vous remarquerez que nous n'avons pas du tout évoqué la possibilité de tracer la tangente avant ce point. L'intérêt des procédures réside en grande partie dans leur utilisation dans des situations inédites lors de leur conception ; les clients bénéficient toujours de l'avantage d'avoir le code déjà écrit.

### Pseudocode

Les personnes qui travaillent sérieusement sur ordinateur doivent communiquer leurs idées. Tout le monde n'utilise pas le même langage de programmation, et pour des raisons d'abstraction et de communication, il est préférable d'éviter de s'appuyer sur un langage particulier pour décrire la solution d'un problème. C'est pourquoi les mathématiciens utilisent souvent du pseudo-code pour décrire un programme informatique, ou plus précisément l'algorithme utilisé. Un algorithme est une séquence d'étapes avec des entrées (informations utilisées) et des sorties (informations renvoyées) bien définies.

Il existe différentes manières d'exprimer un pseudo-code ; quelques exemples apparaissent dans la figure 1. Vous il faut remarquer plusieurs propriétés qu'ils partagent tous :

- Ils précisent tous le nom de l'algorithme, les informations requises (« entrée ») et les informations promises résultat (« sortie »).
- Ils utilisent tous une forme d'indentation pour montrer comment certains codes dépendent d'autres. • Ils s'appuient tous sur un anglais « simple » et des symboles mathématiques pour communiquer ce qui est à faire.
- Aucun d'entre eux ne ressemble à du code informatique dans un langage informatique particulier.

Ce texte vous fournira du pseudo-code à implémenter et vous demandera occasionnellement d'écrire le vôtre. Nous devons adopter une norme, et nous procéderons comme suit : • Les identifiants des variables ou les noms des algorithmes apparaissent en italique. • Les mots-clés des structures de contrôle et des informations fondamentales apparaissent en gras.

Nous introduisons maintenant les mots-clés suivants :

- `algorithm` apparaît au début et déclare le nom d'un algorithme

<pre><b>Algorithm InvolutiveNormalForm:</b> <b>Input:</b> <math>p, F</math> <b>Output:</b> <math>h = NF_L(p, F)</math> <b>begin</b>     <math>h := p</math>     <b>while</b> exist <math>f \in F</math> and a term <math>u</math> of <math>h</math> such that         <math>lm(f) _L(u/cf(h, u)) \neq 0</math>         choose the first such <math>f</math>         <math>h := h - (u/l(f))f</math>     <b>end</b> <b>end</b></pre> <p style="text-align: center;">Apparaît dans [6]</p>	<p><b>Input:</b> <math>I = (T_1, \dots, T_s)</math> with <math>T_i</math> power-products in <math>A = k[X_1, \dots, X_N]</math></p> <p><b>Output:</b> <math>\langle I \rangle</math></p> <pre><b>Function</b> HPNum(<math>I</math>) <b>begin</b>     <b>if</b> <math>I</math> is a base-case <b>then return</b> <math>\langle I \rangle</math>;     <b>else if</b> <math>I</math> is a splitting-case <b>then return</b> HPNum(<math>I_1</math>) + ··· + HPNum(<math>I_r</math>);     <b>else</b>         choose a pivot <math>\mathcal{P}</math>;         <b>return</b> HPNum(<math>I, \mathcal{P}</math>) + <math>t^{\deg \mathcal{P}}</math> HPNum(<math>I : \mathcal{P}</math>);     <b>end.</b></pre> <p style="text-align: center;">Apparaît dans [3]</p>
<pre><b>Input:</b> <math>F, G \in \mathbb{H}[q] \setminus \{0\}</math> <b>Output:</b> <math>GCD(F, G)</math> <b>Initialization:</b> <math>a := F, b := G</math> • <b>While</b> <math>b \neq 0</math> <b>Do</b>     <math>t := b</math>     <math>b := mod_l(a, b)</math>     <math>a := t</math> • <b>Return</b> <math>a</math>.</pre> <p style="text-align: center;">Apparaît dans [5]</p>	<p><b>Theorem 17</b> (Body of Buchberger's Algorithm). Let <math>u_1, \dots, u_s</math> be non-zero vectors in <math>F</math> (homogeneous non-zero vectors in <math>\bar{F}</math>) and let <math>M</math> be the submodule of <math>F</math> (graded submodule of <math>\bar{F}</math>) generated by <math>\{u_1, \dots, u_s\}</math>.</p> <ol style="list-style-type: none"> <li>(1) (<b>Initialization</b>) Pairs = <math>\emptyset</math>, the pairs; Gens = <math>(u_1, \dots, u_s)</math>, the generators of <math>M</math>; <math>G = \emptyset</math>, the <math>\sigma</math>-Gröbner basis (<math>\bar{\sigma}</math>-Gröbner basis) of <math>M</math> under construction.</li> <li>(2) (<b>Main loop</b>) While Gens <math>\neq \emptyset</math> or Pairs <math>\neq \emptyset</math> do             <ol style="list-style-type: none"> <li>(2a) choose <math>w \in</math> Gens and remove it from Gens, or a pair <math>(v_i, v_j) \in</math> Pairs, remove it from Pairs, and let <math>w = S(v_i, v_j)</math>;</li> <li>(2b) compute a remainder <math>v := \text{Rem}(w, G)</math>;</li> <li>(2c) if <math>v \neq 0</math> add <math>v</math> to <math>G</math> and the pairs <math>\{(v, v_i) \mid v_i \in G\}</math> to Pairs.</li> </ol> </li> <li>(3) (<b>Output</b>) Return <math>G</math>.</li> </ol> <p>This is an algorithm which returns a <math>\sigma</math>-Gröbner basis (<math>\bar{\sigma}</math>-Gröbner basis) of <math>M</math>, whatever choices are made in step (2a) and whatever remainder is computed in step (2b).</p> <p style="text-align: center;">Apparaît dans [4]</p>

FIGURE 1. Pseudo-code issu de diverses publications mathématiques

- les entrées apparaissent immédiatement après l'algorithme, et en dessous se trouve une liste en retrait de les informations requises, ainsi que les ensembles dont elles proviennent
- les sorties apparaissent immédiatement en dessous de la liste des entrées, et en dessous se trouve une liste en retrait des informations promises, ainsi que la façon dont elles se rapportent aux entrées
- do apparaît immédiatement en dessous de la liste des sorties, et en dessous se trouve un retrait liste d'instructions
- return apparaît comme la dernière instruction dans les instructions<sup>8</sup>

Nous pouvons maintenant décrire notre procédure tangent\_line() en pseudo-code comme suit :

<sup>8</sup>Il est courant de voir return placé à d'autres endroits dans la liste d'instructions, mais nous adopterons la convention selon laquelle return est toujours la dernière instruction. Bien qu'il soit parfois difficile d'organiser le code autour de cela, convention, cela peut aider à la fois au débogage et à la lisibilité, ce qui est important pour ceux qui apprennent pour la première fois.

```

algorithme tangente_line
entrées
    • un
    •  $x$ , une indéterminée •  $f$ ,
    une fonction dans  $x$  qui est différentiable en  $x = a$  produit • la
droite
    tangente à  $f$  en  $x = a$ 

soit  $m = f(a)$  soit  $b$ 
     $= f(a)$  renvoie
     $m(x - a) + b$ 

```

Les observations suivantes de ce code sont de mise.

- Nous effectuons des affectations avec l'instruction mathématique traditionnelle « let ».<sup>9</sup>
- Non seulement cela ne ressemble pas beaucoup à notre code Sage, mais nous omettons certaines tâches spécifiques à Sage, comme l'affectation de  $f(x)$  à une variable.<sup>10</sup>
- Les entrées et certaines commandes sont répertoriées dans un ordre différent.

Au fur et à mesure que nous parcourons le texte, nous ajouterons des mots-clés de pseudo-code supplémentaires et discuterons de la manière d'accomplir des tâches supplémentaires.

En pratique, nous formulons presque toujours du pseudo-code avant de l'implémenter en code. Nous avons inversé cette pratique ici, principalement pour vous familiariser avec le sujet dès le départ, mais un aspect important du travail informatique est de réfléchir à la manière de résoudre le problème avant de l'écrire. En réalité, un programme n'est rien d'autre que la rédaction d'une solution au problème : « Comment faire ceci, cela et autre chose, étant donné ceci et cela ? » Si vous avez déjà suivi un cours de programmation, vous avez peut-être été élevé dans la mauvaise tradition de ne pas réfléchir à ce qu'il faut faire avant de le faire. Essayez de résister à cette tentation ; nous vous aiderons dans les exercices en vous demandant de formuler du pseudo-code en plus d'écrire un programme.

#### Scripting :

De nombreux mathématiciens doivent répéter un ensemble de tâches à chaque fois qu'ils travaillent sur quelque chose. Écrire des procédures Sage peut simplifier cela, mais si vous les saisissez dans une session particulière, elles ne s'appliquent qu'à cette session, que ce soit en ligne de commande ou dans une feuille de calcul. Si vous quittez la session ou en ouvrez une autre, Sage les oublie et vous devez les redéfinir. Copier-coller ou (pire) retaper ces procédures à chaque démarrage d'une nouvelle session n'est pas très pratique ; il est donc pratique de conserver une bibliothèque de fichiers qui les enregistrent et que nous pouvons charger et utiliser à volonté.

C'est là qu'interviennent les scripts Sage. Un « script » est une séquence d'instructions Sage enregistrée dans un fichier. Charger ce fichier dans Sage est pratique, évitant ainsi de devoir tout retaper. Le format d'un script Sage est identique à celui d'une cellule Sage : il peut s'agir d'une séquence d'instructions simples, mais il peut aussi définir une ou plusieurs procédures. En effet, un grand nombre

---

Comme le montrent les exemples de la figure 1, la diversité est grande. Là où l'on trouve « soit  $m = f(a)$  », certains écrivent «  $m := f(a)$  » et certains langages informatiques ont adopté cette convention pour que l'affectation soit de la forme  $m := df(a)$  (par exemple, Pascal, Eiffel, Maple). Certains écrivent même simplement «  $m = f(a)$  ».

Techniquement, nous n'avons pas besoin de le faire dans Sage non plus, mais cela rend les choses beaucoup plus faciles.

La quantité de Sage est constituée de scripts Sage que les gens ont écrits en raison d'un besoin et, en raison de l'utilité de la tâche, le script a ensuite été intégré à Sage lui-même.

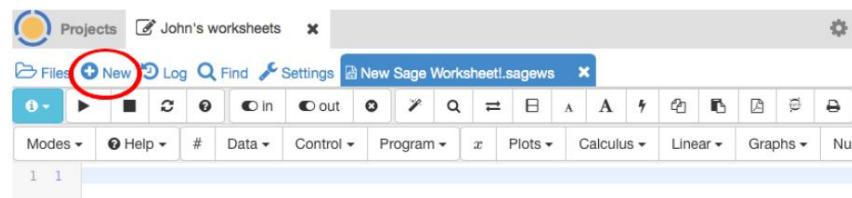
Pour montrer comment créer un script Sage, nous rappelons notre programme sur le calcul de la ligne tangente à une courbe:

```
def tangente_ligne(f, a, x=x) :
    # forme point-pente d'une droite f(x) = fb = f(a)
    df(x) =
    diff(f,x) m =
    df(a) résultat = m*(x - a) +
    b résultat de
    retour
```

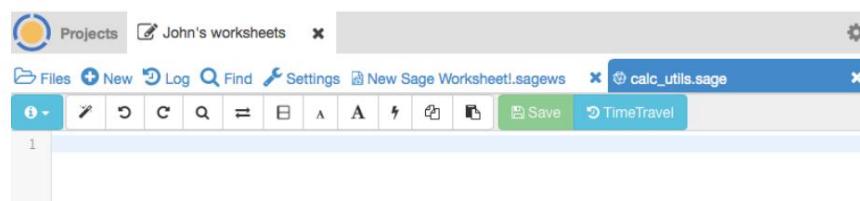
Pour créer un script, effectuez l'une des opérations suivantes.

- Si vous exécutez Sage comme une feuille de calcul gérée par un serveur indépendant, vous devrez contacter votre instructeur ou l'administrateur du serveur. Ce n'est pas difficile à faire, mais c'est un peu compliqué ; nous leur laissons donc le soin de s'en occuper.
- Si vous exécutez Sage en ligne de commande, ouvrez un éditeur de texte sur votre ordinateur, saisissez le programme ci-dessus dans l'éditeur, puis enregistrez le fichier sous le nom calc\_utils.sage. Nous vous recommandons de l'enregistrer dans un répertoire spécial de votre répertoire personnel, intitulé SageScripts, mais choisissez un emplacement facile à mémoriser et à accéder.
- Si vous exécutez Sage via le serveur CoCalc, ouvrez un projet, puis sélectionnez-le dans le menu supérieur.

Nouveau.



Dans la zone de texte sous les instructions « Nommez votre fichier, dossier ou collez un lien », saisissez le nom calc\_utils.sage. Cliquez ensuite sur « Fichier » (à droite) et Sage ouvrira un nouvel onglet portant ce nom.



Saisissez la procédure. Si le texte vous semble trop petit, cliquez sur « Agrandir ». Cliquez sur « Paramètres » dans la capture d'écran ci-dessous. Cela agrandira le texte.

---

<sup>11</sup>L'un des auteurs clique régulièrement sur ce bouton six à huit fois, et ce n'est pas parce qu'il est vieux. — Enfin, pas à ce point-là.

```

1 def tangent_line(f, a, x=x):
2     f(x) = f
3     df(x) = diff(f, x)
4     m = df(a)
5     b = f(a)
6     result = m*(x - a) + b
7     return result

```

Enregistrez-le. (Il devrait en fait enregistrer automatiquement à intervalles réguliers, mais il est toujours utile de cliquer une fois de plus sur le bouton.)

Une fois que vous avez écrit et enregistré votre script, vous pouvez vous tourner vers une session Sage (soit sur la ligne de commande, soit dans la feuille de calcul) et, à ce stade, taper ce qui suit (ne tapez pas réellement avant d'avoir lu ce qui apparaît ci-dessous) :

```
sage : %attach /chemin/vers/fichier/calc_utils.sage
```

Avant de procéder, nous souhaitons décrire deux erreurs potentielles et indiquer une convention dont vous devez être conscient.

Tout d'abord, la convention. Nous avons écrit `/chemin/vers/fichier/` comme « modèle » à remplir. Il s'agit d'un champ à compléter, couramment utilisé pour indiquer que vous devez saisir la valeur correcte :

- Si vous utilisez Sage à partir d'une feuille de calcul provenant d'un serveur indépendant, vous devez demander à votre instructeur ou à l'administrateur du serveur.
- Si vous utilisez Sage à partir de la ligne de commande, vous devriez essayer `'~/SageScripts'`, car cela correspond au répertoire que nous avons suggéré précédemment.
- Si vous utilisez le serveur CoCalc et que vous avez suivi nos instructions ci-dessus, vous n'avez pas besoin de saisir `/chemin/vers/fichier` du tout ! Sinon, si vous avez créé un répertoire spécial pour conserver vos scripts Sage, vous devez saisir le chemin d'accès à ce répertoire.

Une fois cette convention dépassée, nous passons aux erreurs possibles.

Si vous êtes dans le mauvais répertoire, ou si vous avez mal saisi le nom du fichier, ou si vous n'avez pas suivi nos instructions conseil ci-dessus d'attendre pour taper cela, vous verrez probablement une erreur du genre :

```
sage : %attach /path/to/file/calc_util.sage IOError :
fichier u'calc_util.sage' à joindre introuvable
```

Cela indique une erreur dans le chemin d'accès au fichier ou dans le nom du fichier lui-même. Dans cet exemple, les deux sont erronés : nous n'avons pas correctement modifié le chemin d'accès au fichier et nous avons omis le « `s` » dans le nom de fichier `calc_utils.sage`.

Passons maintenant à la deuxième erreur. Il nous arrive de faire des fautes de frappe. Ce n'est pas grave ; c'est la nature humaine ; pas besoin de s'en faire. Après tout, l'ordinateur s'énerve tout seul. Il est possible de faire une faute de frappe, et dans ce cas, vous obtiendrez une erreur de syntaxe.

Un exemple courant pour les débutants sera probablement celui-ci :

---

Le caractère tilde (~) est un moyen standard de référencer le répertoire personnel d'un utilisateur. Nous vous avions suggéré de créer un répertoire nommé `SageScripts` dans votre répertoire personnel, ce qui devrait suffire.

```
sage : %attach /path/to/file/calc_utils.sage IndentationError : la désindentation
ne correspond à aucun niveau d'indentation externe
```

Dans ce cas, prêtez attention à la ligne où Sage s'est plaint et assurez-vous que l'indentation correspond à un bloc de code précédent. Une erreur similaire se produira :

```
sage : %attach /chemin/vers/fichier/calc_utils.sage b = f(a)
^
SyntaxError : syntaxe non valide
```

Cela ne devrait pas vous arriver, mais si cela vous arrive et que tout semble parfait, le problème est que vous avez copié-collé du texte. L'apparence d'un éditeur de texte peut être trompeuse, et lorsque vous copiez d'un fichier à un autre, vous pouvez détecter des caractères de formatage cachés, ou quelque chose de similaire. Dans ce cas, la solution consiste à éliminer le caractère de formatage caché en supprimant ces espaces, puis en les retapant.

Maintenant que ces précisions sont apportées, essayez de charger le fichier. Si tout se passe bien, Sage Gardez le silence. Vous devriez alors pouvoir exécuter votre procédure de la manière habituelle :

```
sage : %attach /chemin/vers/fichier/calc_utils.sage sage : tangent_line(x**2,
2) 4*x - 4
```

Une fois que vous avez correctement attaché calc\_utils.sage à une session Sage, vous pouvez y apporter des modifications. et Sage intégrera automatiquement les modifications, en laissant un message du type :

```
### rechargement du fichier joint calc_utils.sage modifié à 08:39:00
###
```

#### Fiches de travail interactives

Si vous travaillez dans une feuille de calcul Sage, vous avez accès à une fonctionnalité permettant de créer une démonstration facile à manipuler. Nous appelons cette fonctionnalité « feuilles de calcul interactives ». Une feuille de calcul interactive comporte une ou plusieurs procédures interactives permettant à l'utilisateur de manipuler ses arguments de manière pratique grâce à des objets d'interface tels que des zones de texte, des curseurs, etc. Outre la procédure elle-même, la création d'une feuille de calcul interactive comprend deux étapes clés :

- Sur la ligne précédant la procédure, mais dans la même cellule, saisissez `@interact .` • Choisissez parmi les objets d'interface suivants ceux que vous souhaitez utiliser et fournissez à la procédure des arguments « facultatifs » dont les valeurs par défaut sont initialisées sur ces objets : – une zone de saisie (pour fournir une valeur) – un curseur (pour sélectionner une valeur dans une plage) – une case à cocher (pour activer ou désactiver une propriété) – un sélecteur (pour sélectionner une propriété parmi plusieurs) – un sélecteur de couleur (pour... eh bien, j'espère que la raison est évidente)

boîte_d'entrée()	une boîte permettant à l'utilisateur de saisir des valeurs, généralement des nombres ; les options spécifiques incluent • width= largeur de la boîte (104)
curseur(a,b)	une ligne avec un bouton que l'utilisateur peut faire glisser vers la gauche et la droite, allant de a à b avec des valeurs intermédiaires
curseur(a,b, par défaut=c)	similaire à slider(a,b), mais l'utilisateur ne peut faire glisser que par incrément de c
curseur (L)	similaire à slider(a,b), mais l'utilisateur ne peut sélectionner que des options dans la collection L (nous ne vous disons toujours pas ce qu'est une collection, mais vous pouvez toujours utiliser un tuple, comme avant)
range_slider(a,b)	similaire à slider(a,b), mais l'utilisateur sélectionne un sous-intervalle [c, d] de [a, b]
range_slider(a,b, par défaut=(c,d))	similaire à range_slider(a,b), mais le sous-intervalle par défaut est [c, d]
case à cocher()	une case sur laquelle l'utilisateur peut cliquer pour activer ou désactiver
sélecteur()	un menu déroulant ou une barre de boutons, les options spécifiques incluent • valeurs = collection de valeurs à sélectionner • boutons = s'il faut utiliser des boutons (Faux) • nrows = nombre de lignes si vous utilisez des boutons (1) • ncols = nombre de colonnes si vous utilisez des boutons (dépend du nombre) • largeur = largeur du bouton, si les boutons (dépend de valeur)
Couleur(nom)	un bouton qui fait apparaître un sélecteur de couleur ; la couleur par défaut est celle fournie par le nom, comme « rouge »
Couleur (r, v, b)	identique à Couleur(nom), mais la couleur par défaut est la combinaison rouge-vert-bleu donnée par r, g, b

FIGURE 2. Modèles pour les objets d'interface dans une procédure interactive

label=string une étiquette, placée à gauche ; elle peut inclure LATEX entre les signes dollar default=value la valeur initiale de l'objet	
---	--

FIGURE 3. Options communes à tous les objets d'interface d'une procédure interactive, à l'exception de Color()

Vous trouverez des modèles pour ces objets dans la figure 2, et quelques options communes à tous les éléments dans la figure 3. Cette liste n'est pas exhaustive, car nous avons omis certaines options qui, selon nous, vous seront probablement moins utiles. D'autres objets d'interface peuvent également être disponibles, mais nous n'en avons pas connaissance (ils ont peut-être été ajoutés après la rédaction de cet article).

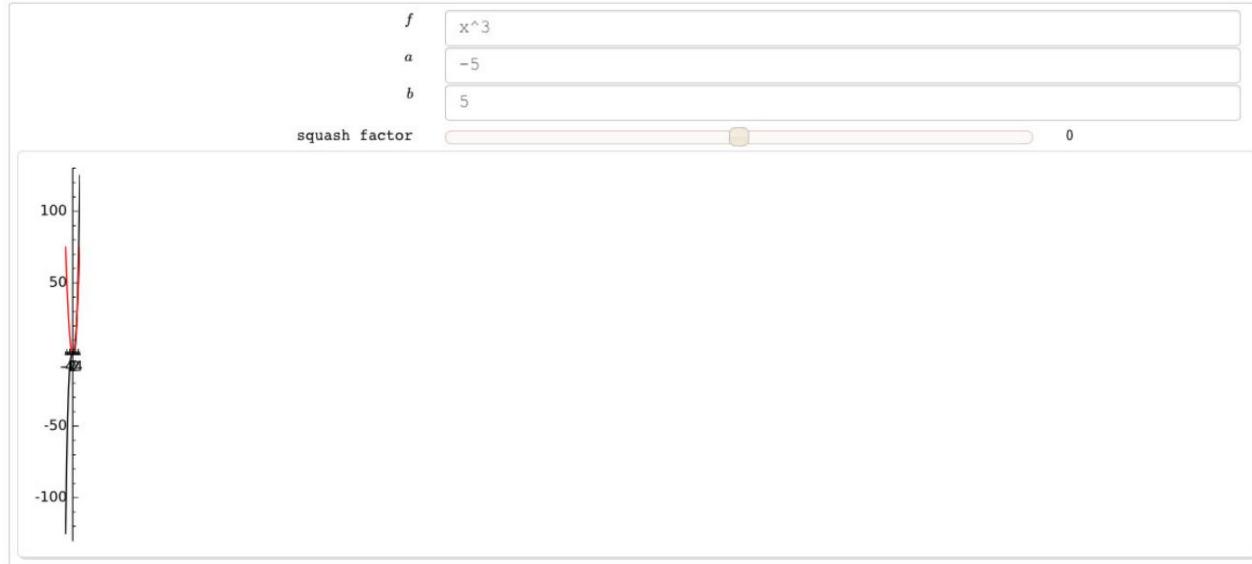


FIGURE 4. Un écrasement malheureux

Les curseurs et les sélecteurs sont similaires : ils permettent de choisir parmi une petite plage de valeurs ; par exemple, `slider((0, 0,25, 0,5, 0,75, 1))` et `selector((0, 0,25, 0,5, 0,75, 1))` permettent tous deux de choisir dans la même liste. Dans ce cas, cependant, un curseur est préférable, car il permet d'ordonner les valeurs de gauche à droite, ce qui est plus clair. Si les valeurs ne s'intègrent pas bien dans un paradigme de gauche à droite, un sélecteur est plus adapté. De plus, les curseurs n'étiquettent pas les valeurs individuellement ; si vous jugez important que chaque valeur soit étiquetée, il est préférable d'utiliser un sélecteur, même si les valeurs s'intègrent bien dans un paradigme de gauche à droite.<sup>13</sup>

Nous illustrons cela avec deux exemples. Le premier est assez simple, simplement pour illustrer le principe : il permet à l'utilisateur de saisir une fonction  $f$  et de tracer  $f$  et sa dérivée sur un intervalle  $[a, b]$ .

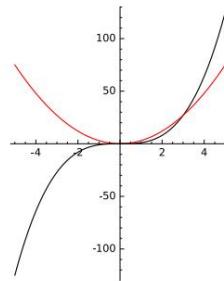
```
sage : @interact
def plot_f_and_df(f=input_box(default=x**3,label='f'), a=input_box(default=-5,label='a'),
                   b=input_box(default=5,label='b'), w=slider(-19,19,1,default=0, label='squash factor'):

    p = plot(f, a, b, couleur='noir') df = diff(f) p = p +
    plot(df, a, b,
         couleur='rouge') show(p, rapport_aspect=(20-w)/20)
```

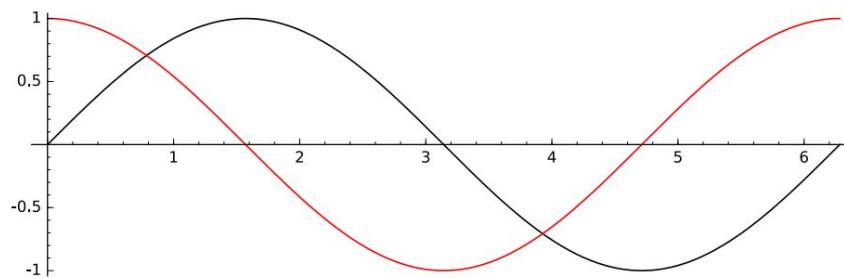
Si vous saisissez correctement ces informations dans une feuille de calcul Sage, vous devriez obtenir un résultat similaire à la figure 4. Vous pouvez constater que les champs de saisie ont des libellés bien formatés. (Comme vous l'avez peut-être deviné grâce à l'utilisation des symboles dollar, le code repose sur LATEX.) Malheureusement, le tracé n'est pas très clair ; il est un peu trop comprimé horizontalement. Nous pouvons corriger ce problème en déplaçant le facteur de compression vers la droite ; si vous faites glisser

<sup>13</sup>Bien sûr, quelqu'un trouvera une exception à chacune de ces lignes directrices ; ce ne sont que des recommandations.

le bouton sur le curseur « facteur d'écrasement » complètement à droite, vous obtenez une image plutôt sympa dès que vous le lâchez :



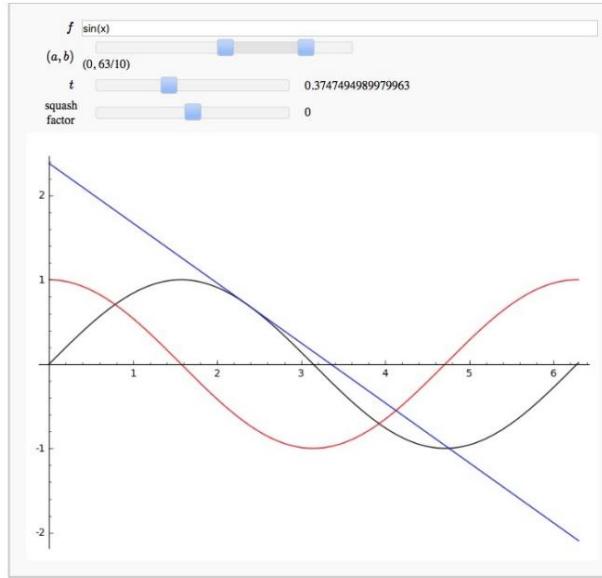
Vous pouvez également modifier  $f$ ,  $a$  et  $b$ , mais ce sont des champs de saisie ; au lieu de faire glisser quoi que ce soit, vous devez donc modifier le texte. Comme  $a < b$ , vous devriez obtenir un graphique correct dès qu'un changement se produit. Essayez une autre fonction, comme  $\sin x$ , avec  $a = 0$  et  $b = 2\pi$ , et le graphique devrait se modifier comme suit (veillez également à remettre le facteur de compression à 0) :



Vous pouvez bien sûr appeler les procédures que vous avez définies depuis une procédure interactive. Écrivons une procédure interactive qui trace  $f$  et sa dérivée sur  $[a, b]$ , ainsi qu'une tangente à  $f$  en un point  $c$  situé entre  $a$  et  $b$ . Nous pouvons appeler la procédure `tangent_line()` que nous avons déjà écrite pour générer la tangente. Tant qu'à faire, nous allons changer la valeur par défaut à  $\sin(x)$  sur  $[0, 2\pi]$ .

```
sage : @interact
def plot_f_and_df(f=input_box(default=sin(x),label='$f$'), \
                   subint=range_slider(-10,10,par défaut=(0,2*pi), \ label='$(a,b)$', \
                   t=slider(0,1,default=0.375,label='$t$'), \
                   w=slider(-19,19,1,default=0,label='squash factor')): (a,b) = subint p = plot(f, a, \
                   b, color='black') df = \
                   diff(f) p = p + plot(df, a, b, color='red') # c nous \
                   indique jusqu'où \
                   se déplacer le long de l'intervalle [a,b] # 0 signifie a ; \
                   1 signifie b ; valeurs entre proportionnelles c = a + t*(ba) p = p + plot(tangent_line(f, \
                   c), a, b) show(p, aspect_ratio=(20-w)/20)
```

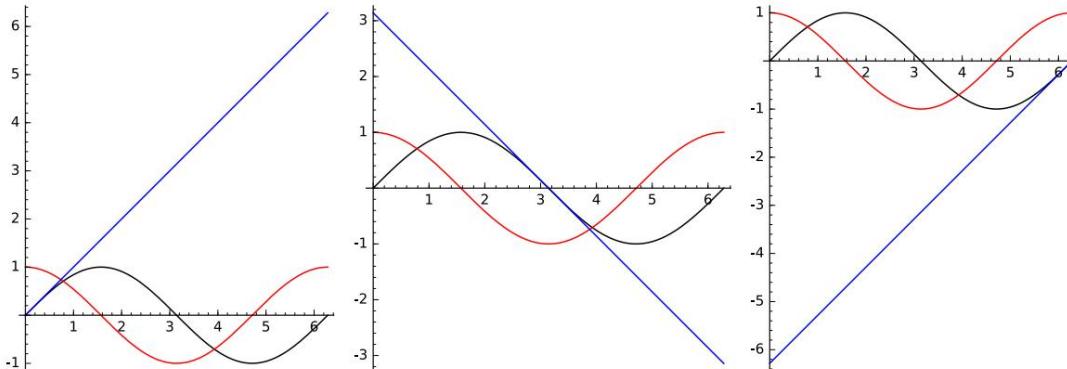
Lorsque vous essayez ceci dans Sage, vous devriez voir ce qui suit :



Vous avez peut-être remarqué que l'utilisateur n'a pas spécifié directement le point  $c$ . Il l'a spécifié indirectement, en choisissant une valeur  $t$  comprise entre 0 et 1. Le code calcule ensuite une valeur de  $c$  comprise entre  $a$  et  $b$  en utilisant la paramétrisation suivante :  $c = a + t(b - a)$ .

Cette paramétrisation est un outil extrêmement utile, en partie parce qu'elle est relativement simple :

- lorsque  $t = 0$ , alors  $c = a + 0 = a$  ; • lorsque  $t = 1$ , alors  $c = a + (b - a) = b$  ; • lorsque  $t = 0,5$ , alors  $c = a + 0,5(b - a) = 0,5a + 0,5b = a+b/2$ , à mi-chemin entre  $a$  et  $b$  ; • et ainsi de suite.



Expérez avec le curseur pour voir comment différentes valeurs de  $t$  permettent naturellement à l'utilisateur de sélectionner un bon point entre  $a$  et  $b$ .

Nous n'avons pas montré le fonctionnement de tous les objets de l'interface, mais nous avons présenté ceux qui nous semblent les plus importants, ainsi qu'une méthode élégante permettant à l'utilisateur de sélectionner un point dans un intervalle sans avoir à vérifier si ce point se trouve réellement dans cet intervalle. Nous vous encourageons à tester les autres objets, notamment la sélection d'une couleur.

### Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Les procédures nous permettent de réutiliser des groupes de commandes communs.
2. Si un argument contient la valeur d'une variable en dehors de la procédure, la modification de l'argument  
La valeur de `ment` modifie également la valeur de cette variable.
3. Sage vous permet de spécifier des arguments facultatifs.
4. Si une procédure Sage ne renvoie pas explicitement de valeur, elle renvoie `None`.
5. Lorsque nous passons des indéterminés comme arguments aux procédures Sage, nous devons redéfinir toutes les fonctions mathématiques qui en dépendent.
6. Les procédures définies par l'utilisateur ne peuvent pas utiliser les procédures Sage.
7. Personne n'utilise de pseudo-code dans la vraie vie.
8. Le pseudocode communique des idées en utilisant un « anglais simple » et des symboles mathématiques, plutôt que que les symboles et les formats d'une langue particulière.
9. La meilleure façon de réutiliser un groupe de procédures Sage dans différentes sessions est de copier-coller le code.
10. La commande `%attach` permet à Sage de lire et d'exécuter un script, ainsi que de recharger automatiquement quand tu le changes.
11. Vous pouvez utiliser des procédures interactives à partir de la ligne de commande Sage.
12. Si l'utilisateur doit sélectionner parmi 100 valeurs équidistantes sur une droite numérique, il est préférable d'utiliser un curseur.
13. Si l'utilisateur doit sélectionner une fonction dans l'ensemble  $\{\sin x, \cos x, \ln x, e^x\}$ , il est préférable d'utiliser un sélecteur.
14. Pour trouver le nombre un quart de la distance de 2 à 10, remplacez  $t = 0,25$  dans le expression  $2 + 8t$ .
15. L'inconvénient des procédures interactives est que vous ne pouvez pas diviser leurs tâches en tâches exécutées par d'autres procédures.  
  
Choix multiple.
  1. Nous commençons la définition d'une procédure dans Sage en utilisant : A. un algorithme B. son type C. son nom
  - D. `def`

2. Le format d'un argument facultatif est : A. de ne pas le lister B. de préfacer son identifiant avec le mot-clé `optional` C. de préfacer le nom de la procédure avec le mot-clé `def` D. de placer = après son identifiant, suivi d'une valeur par défaut

3. Laquelle des affirmations suivantes n'est pas une description précise d'un argument ?

  - A. un substitut à une variable extérieure à la procédure, dont la procédure peut modifier la valeur
  - B. une copie de certaines données extérieures à la procédure
  - C. une variable dont vous ne connaissez peut-être pas la valeur à l'avance
  - D. des informations dont la procédure peut ou non avoir besoin pour effectuer ses tâches

4. La meilleure façon de signaler le résultat d'une procédure au client est de

  - A. lister le résultat dans une instruction de retour
  - B. attribuer la valeur à l'un des arguments
  - C. l'attribuer à une variable globale appelée `résultat`
  - D. attribuer une valeur au nom de la procédure

5. Une procédure qui n'a pas d'instruction de retour :

  - A. génère une erreur

B. renvoie l'ensemble vide C.  
renvoie None D.

ne fait rien de spécial

6. Lorsqu'une liste de commandes est soumise à un mot-clé qui démarre une structure de contrôle, telle que def,  
vous devriez :

- A. indenter les commandes en question B. ajouter deux points à la fin de l'instruction de la structure de contrôle C. inverser l'indentation lorsque la liste est terminée D. tout ce qui précède 7.

Laquelle des propriétés

suivantes notre norme de pseudo-code partage-t-elle avec le code Sage ?

- A. Les déclarations soumises à des structures de contrôle sont en retrait.
- B. Nous pouvons utiliser def à la fois dans le pseudo-code et dans le code Sage.
- C. Nous spécifions les types d'entrées dans Sage, tout comme nous spécifions de quel ensemble provient une entrée dans pseudo-code.
- D. tout ce qui précède 8.

La meilleure façon dans Sage de remplacer une valeur indéterminée qui a été passée en argument est

par : A. substitution d'appel de fonction

B. substitution de dictionnaire C.

affectation de mot-clé D.

substitution d'appel de fonction après redéfinition de la fonction 9. À quels

identifiants suivants ne pouvons-nous pas attribuer de valeurs ?

A. un identifiant valide B.

un mot-clé C. une  
constante

D. le nom d'une procédure déjà définie

10. Lorsque l'utilisateur modifie un paramètre dans une procédure interactive, le seul effet garanti est  
changer quelle partie de la procédure ?

A. une variable globale B.  
une variable locale C. un

argument de la procédure D. le nom de la  
procédure

Réponse courte.

1. Expliquez la modularité et pourquoi c'est une bonne pratique lors de l'écriture de programmes.
2. La modularité et l'arithmétique modulaire sont-elles la même chose ?

Programmation.

1. Sage n'ajoute pas automatiquement de pointes de flèche à une courbe. Puisque la dérivée indique la pente de la courbe en un point donné, nous pouvons déterminer la direction des flèches à l'extrémité en calculant les dérivées aux extrémités et en y insérant des flèches. Écrivez une procédure arrowplot() qui prend comme argument une fonction f, deux extrémités a et b, et une longueur de flèche  $\Delta x$ , puis trace f sur  $[a, b]$ , en ajoutant des pointes de flèche selon le principe selon lequel  $(a, f(a))$  et  $(a - \Delta x, f(a) - f(a)\Delta x)$  nous donneraient deux points sur une droite tangente à f en a (et qui spécifieraient une flèche appropriée). (Nous vous laissons le soin de modifier cette définition pour une pointe de flèche en b.)

2. Écrivez le pseudo-code d'une procédure qui calcule et renvoie la ligne normale à  $f$  à  $x = c$ .

Ensuite : (a) Implémentez ceci dans le

code Sage. (b) Choisissez une fonction transcendante  $f$  et un point  $c$  où  $f$  est transcendant. (c)

Utilisez ce code et la procédure `tangent_line()` pour tracer les lignes normales et tangentes.

à  $f$  à  $x = c$ .

(d) Écrivez une procédure interactive pour tracer le graphique d'une fonction, d'une tangente en un point  $x = c$  et de la normale en  $x = c$ . Outre les éléments d'interface évidents, incluez au moins un curseur pour contrôler le rapport hauteur/largeur du graphique obtenu.

Votre implémentation échouera parfois ; par exemple, si vous utilisez  $f = (x - 1)$ , vous devriez<sup>2</sup> + 2 et  $c = 1$ , rencontrer une `ZeroDivisionError`.

## NE PAS PANIQUER!

Vous devriez vous attendre à cette erreur, car le calcul d'une perpendiculaire nécessite une réciproque, c'est-à-dire une division, ce qui ouvre la possibilité d'une division par zéro. Nous abordons ce problème au chapitre 7 sur la prise de décision. Assurez-vous que votre code fonctionne avec des fonctions et des points qui fonctionnent correctement.

3. Écrivez le pseudo-code d'une procédure nommée `avg_value` dont les entrées sont une fonction  $f$ , une fonction indéterminée  $x$  et les extrémités  $a$  et  $b$  d'un intervalle  $I$ . La procédure renvoie la valeur moyenne de  $f$  sur l'intervalle. (Vous devrez peut-être revoir quelques notions de calcul pour résoudre ce problème.)
4. Définissons l'opération  $a \star b = ab + (a + 2b + 1)$ . (a) Écrivez le pseudo-code d'une procédure nommée `star()` qui accepte deux entiers  $a$  et  $b$  et renvoie  $a \star b$ .
- (b) Implémentez ce pseudo-code en tant que procédure Sage.
  - (c) Testez votre procédure sur plusieurs valeurs différentes de  $a$  et  $b$ . (d) Existe-t-il une valeur de  $a$  telle que  $a \star b = b$ , quelle que soit la valeur de  $b$  ?
5. Écrivez une procédure Sage nommée `dotted_line_segment()` qui accepte sept arguments nommés  $x1$ ,  $y1$ ,  $x2$ ,  $y2$ , couleur, `pointsize` et `pointcolor`. Les valeurs par défaut des couleurs sont « noir » et la couleur par défaut de `pointsize` est 60. La procédure renvoie la somme de :
- un segment de droite reliant ces points et de couleur `color` ;
  - deux points, de couleur `pointcolor` et de taille `pointsize`, dont les emplacements sont  $(x1, y1)$  et  $(x2, y2)$ .
- (a) Utilisez cette procédure pour tracer un segment de ligne reliant les points  $(0, 0)$  et  $(1, 4)$  dont la couleur et la couleur du point sont toutes deux noires.
  - (b) Utilisez cette procédure pour tracer un segment de ligne noir reliant les points rouges  $(0, 6)$  et  $(2, 0)$ .
- Vous n'avez pas besoin d'écrire de pseudo-code pour cette procédure.
6. Implémentez le pseudo-code suivant dans Sage.

```
algorithme Taylor_Truncated4
entrées
    • un
    • x, une indéterminée • f,
      une fonction dans x qui est intégrable en x = a produit
    • la série
      de Taylor tronquée pour f (x) autour de x = a

soit le résultat = f (a)
ajouter f (a)·(x - a) au résultat
ajouter f (a)·(x - a)2/2 au résultat
ajouter f (a)·(x - a)3/6 au résultat
ajouter f (4) (a)·(x - a)4/24 au résultat
renvoyer le résultat
```

Voici un indice : pour « ajouter... au résultat », utilisez la construction résultat = résultat +...

Vérifiez-le en trouvant la série de Taylor tronquée pour x à x = 1 et en comparant la série tronquée

Valeur de la série 1.1 à x = 1.1 avec la valeur de e 7. Expliquez comment le .

« facteur d'écrasement » utilisé dans nos procédures interactives a été utilisé pour définir le rapport hauteur/largeur du graphique. Votre explication doit expliquer pourquoi la valeur la plus à gauche écrase le graphique au maximum à gauche, pourquoi la valeur la plus à droite l'étire au maximum à droite, et pourquoi la valeur médiane se situe quelque part entre les deux.

## CHAPITRE 5

## Se répéter définitivement avec les collections

Il existe de nombreux cas où l'on souhaite répéter une tâche plusieurs fois. Un exemple classique est celui des équations différentielles : supposons que nous sachions

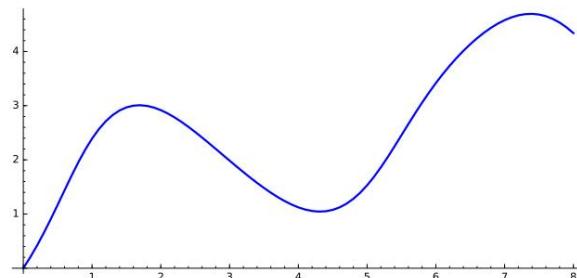
$$\frac{dy}{dx} = \sin y + 2 \cos x,$$

c'est-à-dire qu'à tout moment  $(x, y)$ , la valeur de  $y$  change de  $\sin y + 2 \cos x$ . Il est difficile, et souvent impossible, de trouver la formule exacte de  $y$  en termes de  $y$ , il est donc nécessaire d'approximer les valeurs « futures » de  $y$  à partir d'un point de départ  $(x, y)$ .<sup>1</sup> Dans ce cas, vous décidez généralement de faire un certain nombre de très petits « pas en avant » avec  $x$ , et de réévaluer  $y$  à chaque point, en traçant le comportement résultant. Dans ce cas, supposons que nous commençons à  $(0, 0)$  ; alors  $y(0, 0) = 0$ , ce qui suggère que la fonction souhaite se déplacer le long d'une droite de pente 2. Après tout, vous calculez la dérivée, qui est la pente de la tangente, qui va dans la même direction que la courbe en ce point. Cette technique de déplacement le long de la tangente est connue sous le nom de méthode d'Euler.

Cependant, la courbe ne veut suivre la direction de sa tangente que pendant un instant ; dès que l'on s'éloigne du point  $(0, 0)$ , la dérivée change très légèrement et la tangente n'est plus valide. Pour éviter toute erreur, on avance d'un petit pas, disons  $\Delta x = 1/10$ , le long de cette droite, ce qui nous amène au point  $(.1, .2)$ . Ici,  $y(.1, .2) \approx 2.1$ , donc on avance le long d'une droite de pente 2.1, ce qui nous amène au point  $(.2, .41)$ . Ici,  $y(.2, .41) \approx 2.0$ , ce qui nous amène à  $(.3, .61)$ .

Ici,  $y(.3, .61) \approx 1.9$ , ce qui nous amène à  $(.4, .8)$ . Ici,  $y(.4, .8) \approx 1.8$ , ce qui nous amène à  $(.5, .98)$ .

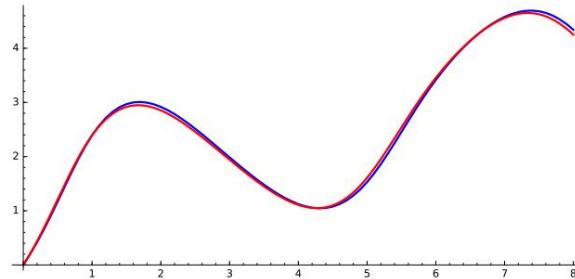
Ici,  $y(0.5, 0.98) \approx 1.6$ , ce qui nous amène à  $(0.6, 1.14)$ . En allant suffisamment loin, vous obtiendrez une image ressemblant à ceci :



Pour générer cette courbe, nous avons utilisé un pas de  $\Delta x = 1/10$ . Répétez le processus depuis  $(0, 0)$  avec un pas plus petit, par exemple  $\Delta x = 1/100$ , et vous obtiendrez un résultat légèrement différent :

---

<sup>1</sup>Par exemple, voici comment fonctionnent les prévisions météorologiques.



Cela reflète le fait que notre approximation comporte une certaine erreur. Pourtant, ce n'est pas si grave ; les approximations sont assez proches. Par ailleurs, nous avons obtenu la courbe bleue avec 80 points et la courbe rouge avec 800 points. Vous ne voulez pas faire cela à la main, n'est-ce pas ?

Lorsqu'une tâche (ou un ensemble de tâches) doit être répétée plusieurs fois sur le résultat de l'application précédente, on parle d'itération de répétition. L'itération étant récurrente en mathématiques computationnelles, les langages de programmation proposent généralement une structure de contrôle appelée boucle. Contrairement à la méthode d'Euler, on ne sait pas toujours à l'avance combien de fois la boucle doit se répéter ; de nombreuses boucles sont donc indéfinies. Néanmoins, il est très fréquent de pouvoir déterminer dès le départ le nombre exact de répétitions d'une tâche ; on parle alors de boucle définie. Ce chapitre présente les boucles définies ; nous reportons les boucles indéfinies au chapitre 8.

### Comment faire en sorte qu'un ordinateur répète un nombre fixe de fois ?

Pseudo-code. Nous pouvons décrire la méthode d'Euler en pseudo-code comme suit.

```

algorithme Eulers_method
entrées
    • df , la dérivée d'une fonction • (x0 , y0 ),
    valeurs initiales de x et y • Δx, taille du
    pas • n, nombre
    de pas à effectuer sorties •

approximation de (x0 + nΔx, f (x0 + nΔx))

laissez a = x0 , b = y0
répéter n fois
    ajouter Δx · df (a, b) à b
    ajouter Δx à a
retourner (a, b)

```

Nous utilisons la répétition dans le pseudo-code pour indiquer qu'un ensemble de tâches doit être répété un certain nombre de fois et indentons les tâches à répéter.

[2](#) Pour Code Sage. Comme la plupart des langages informatiques, Sage ne possède pas de mot-clé « repeat ». Pour les boucles définies, Sage utilise un mot-clé plus généraliste : `for`. Lorsque vous savez que vous devez répéter une tâche  $n$  fois, la construction est relativement simple : `for variable in range(n) :`

```
range(n) :
```

Pour la variable, vous choisissez un identifiant qui contiendra le numéro de la boucle à chaque fois que vous la traverserez.

Nous pouvons maintenant implémenter la méthode d'Euler dans Sage :

```
sage : def eulers_method(df, x0, y0, Delta_x, n) :
    # point de départ a, b =
    x0, y0 # calculer les
    lignes tangentes et avancer pour i dans la plage (n) : b =
    Delta_x * df(a, b) + ba =
        Delta_x + a

    retourner a, b
```

Cela semble assez simple. Essayons :

```
sage : df(x,y) = sin(y) + 2*cos(x) sage :
méthode_eulers(df, 0, 0, 1/10, 80)
```

Si vous essayez, vous remarquerez que le problème prend énormément de temps à se résoudre, certainement plus de quelques secondes. Pour comprendre pourquoi, interrompez le calcul (appuyez sur le bouton « Arrêter » dans le cloud, cliquez sur le menu « Action » et cliquez sur « Interrompre » sur un serveur indépendant, ou maintenez la touche Ctrl enfoncée et appuyez sur C en ligne de commande) et réexécutez le code avec une valeur  $n$  inférieure :

```
sage : méthode_eulers(df, 0, 0, 1/10, 10) (1, 1/5*cos(9/10) +
1/5*cos(4/5) + 1/5*cos(7/10) + 1/5*cos(3/5) + 1/5*cos(1/2) ...
```

(Les points de suspension à la fin indiquent qu'il y a beaucoup plus après cela. Beaucoup plus !)

Vous voyez ce qui se passe ? Sage calcule des valeurs exactes ; et la valeur exacte de ce nombre devient de plus en plus complexe à chaque itération. Vous pouvez modifier le code pour simplifier `b` après chaque calcul, mais cela ne sert à rien. Cela illustre simplement un inconvénient du calcul symbolique : pour obtenir des valeurs « exactes », on perd du temps. Mais inutile de sacrifier ce temps ici ! Après tout, nous approximons la valeur de toute façon. Dans ce cas, passons aux valeurs à virgule flottante et voyons si cela accélère les choses. Remplaçons  $1/10$  par  $0.1$  et voyons comment cela évolue.

dehors.

---

Notre utilisation de la répétition vise à illustrer comment le pseudo-code vise la clarté de la communication, plutôt que l'imitation d'un langage particulier. Il n'est pas rare de voir cette fonction utilisée dans le pseudo-code. Cela dit, certains langages de programmation proposent une structure de répétition pour les boucles.

```
sage : eulers_method(df, 0, 0, .1, 80)
(7,99999999999999, 4,340418570291038)
```

Cela revient à  $(8, 4.34)$ . Non seulement nous avons obtenu une réponse rapidement, mais elle a été presque instantanée ! Il est clairement préférable de se fier aux nombres à virgule flottante lorsqu'on sait qu'on effectue une approximation.

Que vient-il de se passer ? Que se passe-t-il lorsqu'on exécute une boucle ? Examinons ce qui se passe. stylos, en regardant attentivement les valeurs de  $a$  et  $b$  à chaque fois que nous passons par la boucle.

Lorsque nous avons exécuté le programme,  $a$  et  $b$  prennent tous deux la valeur 0, car c'est ce que nous avons transmis comme arguments pour la valeur initiale. Nous arrivons ensuite à la ligne

for  $i$  in range( $n$ ) :

Rappelons que cela demande à Sage de répéter les tâches suivantes  $n$  fois. La valeur de la boucle est stockée dans la variable  $i$ , au cas où vous en auriez besoin ; nous n'en avons pas besoin pour ce problème, mais un exemple ultérieur illustrera son utilité.

Maintenant,  $n$  est un argument, et dans ce cas, nous lui avons attribué la valeur 80. Les tâches indentées se répéteront donc 80 fois. Illustrons maintenant ce qui se passe les premières fois : lorsque  $i = 0$  : la

première ligne indique à Sage de calculer  $\Delta xdf(a, b)$  et de l'ajouter à  $b$ , puis d'attribuer le résultat.

$\rightarrow b$ . Après cela,  $b = .1 f(0, 0) + 0 = .2$ .

La deuxième ligne indique à Sage d'ajouter  $\Delta x$  à  $a$ , puis d'attribuer le résultat à  $a$ . Après cela,  $a = 0,1 + 0 = 0,1$ .

Lorsque  $i = 1$  : la première ligne indique à Sage de calculer  $\Delta xdf(a, b)$  et de l'ajouter à  $b$ , puis d'affecter le résultat à  $b$ . Après cela,  $b = .1 f(.1, .2) + .2 \approx .42$ .

La deuxième ligne indique à Sage d'ajouter  $\Delta x$  à  $a$ , puis d'attribuer le résultat à  $a$ . Après cela,  $a = 0,1 + 0,1 = 0,2$ .

Lorsque  $i = 2$  : la première ligne indique à Sage de calculer  $\Delta xdf(a, b)$  et de l'ajouter à  $b$ , puis d'attribuer le résultat à  $b$ . Après cela,  $b = .1 f(.2, .42) + .42 \approx .66$ .

La deuxième ligne indique à Sage d'ajouter  $\Delta x$  à  $a$ , puis d'attribuer le résultat à  $a$ . Après cela,  $a = 0,1 + 0,2 = 0,3$ .

... et ainsi de suite. Répétez cette opération 80 fois et vous obtiendrez la valeur indiquée par Sage.

#### Comment ça marche ? ou une introduction aux collections

Nous allons discuter plus en détail du fonctionnement de ce processus. Les boucles définies fonctionnent en passant par un collection ; en général, vous pouvez utiliser le mot-clé `for` sous la forme `for variable`

`in collection:` et Sage effectuera

les tâches indentées suivantes autant de fois qu'il y a d'objets dans la collection.

Lors du premier passage dans la boucle, la variable prend la valeur du « premier » élément de la collection ; à chaque passage suivant, la variable prend la valeur de l'élément de la collection qui « suit » la valeur actuelle de la variable.

Nous avons mis « premier » et « suivant » entre guillemets car, dans certaines collections, l'élément que Sage considère comme « premier » peut ne pas correspondre à vos attentes, et la valeur qu'il considère comme « suivant » la valeur actuelle peut ne pas correspondre à vos attentes. Ce problème n'est pas aussi grave qu'on pourrait le croire, car il se produit uniquement dans les collections où il ne faut pas s'attendre à un « premier », un « second », etc. Nous en parlerons dans une seconde.

Mais qu'est-ce qu'une collection ? Comme son nom l'indique, une collection est un objet qui « contient d'autres objets. On peut classer les collections de deux manières.

- La première classification consiste à savoir si une collection est indexée.
  - Les collections indexées ordonnent leurs éléments qui vous permettent d'accéder à n'importe quel élément en fonction de son emplacement.
  - Les collections non indexées n'ordonnent pas leurs éléments, vous pouvez donc accéder à n'importe quel élément, mais pas par son emplacement.
- La deuxième classification est de savoir si une collection est mutable.
  - Les collections mutables vous permettent de modifier leurs valeurs.
  - Les collections immuables ne vous permettent pas de modifier leurs valeurs.

Nous utilisons cinq types de collections.

Un tuple est une collection indexée et immuable ; nous appelons un tuple de n éléments un n-tuple.

Vous créez un tuple en utilisant des parenthèses ou la commande `tuple()`, en insérant une autre collection entre ses parenthèses ; par exemple, les deux commandes suivantes font la même chose :

```
sage : a_tuple = (3, pi, -I) sage :
b_tuple = tuple([3, pi, -I]) sage : a_tuple ==
b_tuple
Vrai
```

Vous pouvez également créer un « tuple vide » en ne plaçant rien entre les parenthèses, ni `()` ni `tuple()`. En tant que collections immuables, les tuples sont utiles pour communiquer des données qui ne doivent pas être modifiées, par exemple des constantes. Vous avez déjà vu et utilisé les tuples ; nous les avons largement utilisés pour fournir des points aux commandes de traçage.

Une liste est une collection indexée et modifiable. On crée une liste en utilisant des crochets ou la fonction `list()`. commande, en insérant une autre collection entre les crochets ou les parenthèses ; par exemple,

```
sage : a_list = [3, pi, -I]
```

Vous pouvez également créer une liste vide en ne plaçant aucun caractère entre parenthèses, ni `[]` ni `list()`. Comme les collections sont modifiables, il est facile de modifier une liste et ses éléments. Si vous devez stocker des valeurs susceptibles de changer, vous avez besoin d'une liste, et non d'un tuple, car un tuple est indémodable.

Un ensemble est une collection non indexée et modifiable. On le crée à l'aide d'accolades ou de la commande `set()` ; par exemple :

```
sage: a_set = {3, pi, -I} sage: a_set
{-I, 3, pi}
```

Remarquez comment l'ordre des éléments a changé. Vous pouvez également créer un ensemble vide en ne plaçant rien entre les parenthèses, comme ceci : `set()`. Comme ils sont mutables, vous pouvez modifier un ensemble. Une propriété importante d'un ensemble est qu'il ne stocke qu'une seule copie de chaque élément ; ajouter des copies supplémentaires ne nous en laisse qu'une seule. Par exemple :

```
sage: another_set = {2, 2, 2, 2, 2, 2} sage:
another_set {2}
```

Nous n'utilisons pas souvent les ensembles dans ce texte ; bien qu'ils aient des utilités importantes, leur utilisation peut s'avérer complexe, car ils n'acceptent que des objets immuables, et de nombreux objets Sage sont mutables. Par exemple, vous pouvez stocker des tuples dans un ensemble, mais pas des listes. D'ailleurs, les ensembles eux-mêmes sont mutables ; il est donc impossible de stocker un ensemble dans un autre.

```
sage : { un_tuple } {(3,
pi, -l)} sage :
{ un_ensemble }
TypeError : type impossible à hacher : 'set'
```

Un ensemble gelé est une collection non indexée et immuable. On crée un ensemble gelé avec la méthode `frozenset()`. commande; par exemple,

```
sage : f_set = frozenset(a_set) sage :
f_set frozenset({-
l, 3, pi})
```

Vous pouvez également créer un « ensemble gelé vide » en ne plaçant rien entre les parenthèses, `frozenset()`.

Les ensembles gelés sont particulièrement utiles lorsque vous avez besoin d'ensembles d'ensembles : vous ne pouvez pas stocker un ensemble mutable à l'intérieur d'un ensemble, vous stockez donc un ensemble gelé à l'intérieur d'un ensemble.

Un dictionnaire est comparable à une liste : il s'agit d'une collection indexée et modifiable. À la différence d'une liste, l'indexation se fait par entrée plutôt que par position. Un dictionnaire est créé à l'aide d'accolades et de deux-points ; les accolades le délimitent, tandis que les deux-points indiquent une correspondance entre les clés (entrées du dictionnaire) et les valeurs (définitions des entrées). Par exemple :

```
sage : a_dict = {x:2, x^2 + 1:'bonjour'}
```

Dans ce dictionnaire (assez absurde), l'entrée `x` correspond à la valeur 2, tandis que l'entrée `x^2 + 1` correspond à la valeur « hello ». Une autre façon de créer un dictionnaire est d'utiliser la commande `dict()` avec une collection de tuples ; la première entrée du tuple devient la clé, la seconde la valeur.

```
sage : tup_dict = dict(( (x,1), (15,-71), (cos(x),3) )) sage : tup_dict[cos(x)] 3
```

Vous pouvez également créer un « dictionnaire vide » en ne plaçant rien entre les parenthèses d'une commande `dict()` (`dict()`). Nous avons déjà utilisé des dictionnaires pour effectuer une substitution de dictionnaire.

Comment fonctionne l'indexation ? La réponse à cette question dépend du type de collection, mais elle implique toujours l'utilisation de crochets `[]`.

Pour un dictionnaire, vous tapez la clé entre les crochets et Sage renvoie la valeur attribuée à cette clé :

```
sage : a_dict[x^2 + 1] 'bonjour'
```

Nous pouvons maintenant expliquer comment fonctionne la substitution de dictionnaire dans une expression : lorsque vous tapez

```
sage : f = x^2 + 2 sage :
f{x:1}
```

alors Sage utilise le dictionnaire `{x:1}` pour interpréter chaque `x` dans `f` comme un 1 à la place.

Pour les tuples et les listes, l'indexation a une signification analogue à celle d'un indice en mathématiques.

Tout comme `a1`, `a2`, ..., `ai`, ... indiquent les premier, deuxième, ..., ième, ... éléments d'une séquence, `LorT[i]` indique l'élément en position `i`. Cette légère différence de formulation est importante ; pour Sage, vous devez fournir une « position légale », qui n'est pas tout à fait celle attendue :

```
sage : un_tuple (3,
pi, -l) sage :
un_tuple[1] pi sage :

un_tuple[2]
-JE
```

Si vous regardez attentivement la numérotation, vous remarquerez que Sage commence à numérotter ses éléments à la position 0, et non à la position 1. Pour lire le premier élément de `a_tuple`, vous devez en fait taper `a_tuple[0]`.

Supposons que la collection `C` soit un tuple ou une liste, et qu'elle comporte `n` éléments. La signification de « légal » position correspond au tableau suivant :

<code>C[0]</code>	<code>C[1]</code>	<code>C[2]</code>	<code>C[n-2]</code>	<code>C[n-1]</code>	<code>C[n]</code>
quel élément ? <b>PANIQUE !</b>	premier	deuxième	troisième	.. avant-dernier	dernier <b>PANIQUE !</b>
<code>C[-n-1]</code>	<code>C[-n]</code>	<code>C[-n+1]</code>	<code>C[-n+2]</code>	<code>C[-2]</code>	<code>C[-1]</code>

Comme d'habitude, **PANIC!** signifie « une erreur s'est produite ! ». Nous en parlerons dans la section suivante, mais remarquez une chose surprenante : les indices négatifs ont une signification ! Ils vous font remonter les éléments du langage `C`. Nous n'utiliserons pas cette fonctionnalité dans ce texte, mais elle peut parfois être utile.

Ce que vous pouvez faire avec les collections. Nous n'aborderons pas ici les applications des collections, mais plutôt les commandes que vous pouvez utiliser et les méthodes que vous pouvez leur envoyer. Nous avons déjà vu comment accéder aux éléments des collections indexées via l'opérateur crochet.

Les cinq collections. Les procédures et opérations suivantes sont communes aux cinq collections :

élément	le nombre d'éléments dans la collection
len(collection) dans la collection	Vrai si et seulement si l'élément est dans la collection (si la collection est un dictionnaire, cela signifie que l'élément apparaît comme une clé)
max(collection)	le plus grand élément de la collection le
min(collection)	plus petit élément de la collection renvoie
trié(collection, reverse=Vrai ou Faux)	une copie triée de la collection (ordre inverse si reverse=True) renvoie une copie
trié(collection,clé, inverse=Vrai ou Faux)	de la collection, triée selon la clé (ordre inverse si reverse=True)

La plupart du temps, un programme ne sait pas à l'avance avec combien d'éléments il doit travailler, donc avoir un moyen de déterminer ce nombre est extrêmement utile.

```
sage : len(a_set) 3

sage : len(un autre ensemble)
1

sage : sqrt(2) dans a_list
FAUX

sage : -sqrt(-1) dans un tuple
Vrai

sage : trié(un_tuple) [3, pi, -]
```

Notez comment la commande `sorted()` renvoie toujours une liste, même si nous avons fourni un tuple pour son entrée.

Il peut arriver que vous souhaitiez trier une collection différemment de la méthode par défaut de Sage. Vous pouvez modifier les critères de tri à l'aide de l' option clé , à laquelle vous attribuez une procédure renvoyant un objet servant de clé de tri, un peu comme un dictionnaire. Nous n'utiliserons pas cette option souvent, mais l'exemple ci-dessus illustre bien ce point : du point de vue de l'analyse complexe, il semble étrange de trier  $-i$  après 3 et  $\pi$ , lorsque sa « norme » est plus petite. Pour trier selon la norme, nous pouvons écrire une procédure qui calcule la norme de tout nombre complexe et l'utiliser comme clé :

```
sage : def by_norm(z) :
    renvoie real_part(z)**2 + imag_part(z)**2
sage : trié(un_tuple, clé=par_norme)
[-i, 3, pi]
```

Listes et tuples. Les méthodes et opérations suivantes s'appliquent uniquement aux listes et tuples :

LorT.count(élément)	le nombre d'éléments temporels apparaît dans LorT
LorT.index(élément)	l'emplacement de l'élément dans LorT
LorT1+LorT2	crée une nouvelle liste/tuple dont les éléments sont ceux de LorT1, suivis de ceux de LorT2

Il devrait être logique que vous ne puissiez pas appliquer ces techniques aux ensembles, aux ensembles figés ou aux dictionnaires.

Aucun élément ni aucune clé ne peut apparaître plus d'une fois dans un ensemble, un ensemble figé ou un dictionnaire ; count() renvoie donc au maximum 1 dans chaque cas. Les ensembles et les ensembles figés ne sont pas indexables, donc index() n'a pas de sens. Pour les dictionnaires, index() pourrait être pertinent, mais il n'est pas implémenté directement.

Nous ne pouvons « ajouter » que deux listes ou deux tuples ; vous ne pouvez pas ajouter une liste à un tuple, ou vice-versa.

Nous avons épuisé les commandes disponibles pour les tuples, mais une liste offre encore plus de possibilités. Les listes étant indexables et modifiables, nous pouvons modifier un élément particulier d'une liste grâce à l'affectation d'éléments :

```
sage: a_list[0] = 1 sage:
a_list [1, pi, -1]
sage: a_list[0]
= 3 sage: a_list [3, pi, -1]
```

Comme d'habitude, nous utilisons 0 car Sage considère que le premier élément a l'emplacement 0.

Outre l'attribution d'éléments, les listes comportent certaines méthodes qui ne sont pas disponibles pour les tuples ; voir le tableau 1.<sup>4</sup> UN quelques distinctions méritent d'être faites à propos de ces commandes :

- .pop() et .remove() diffèrent en ce que l'un fait référence à un emplacement, tandis que l'autre fait référence à un élément particulier.

```
sage: a_list.pop(1) pi sage:
a_list.remove(3) sage: a_list [-1]
```

Il n'y a jamais eu d'élément à l'emplacement 3, ce qui souligne que remove() recherchait un élément dont la valeur était 3.

Cette opération peut être assez sophistiquée, car Sage effectue des réductions évidentes pour vérifier si un élément a une valeur donnée :

---

<sup>3</sup>C'est en fait faisable, mais un peu compliqué.

Comme d'habitude, cette liste n'est peut-être pas exhaustive, et ne l'est probablement pas. Nous nous concentrerons uniquement sur les commandes dont nous pensons que vous aurez le plus souvent besoin et celles disponibles au moment de la rédaction de cet article. Pour vérifier si la liste est exhaustive, n'oubliez pas que vous pouvez l'obtenir en saisissant « a\_list.» puis en appuyant sur la touche Tab .

L.append(élément)	ajouter un élément à la fin de L
L.extend(collection)	ajouter les éléments de la collection à L
L.insert(emplacement, élément)	ajouter un élément à l'emplacement donné (à partir de 0)
L.pop()	supprime (et renvoie) le dernier élément de la liste supprime et
L.pop(emplacement)	renvoie l'élément à l'emplacement indiqué (à partir de 0) supprime l'élément nommé inverse la liste
L.remove(élément)	
L.reverse()	
L.sort()	trie la liste selon le mécanisme par défaut de Sage trié la
L.sort(clé)	liste selon la clé donnée trié la liste dans
L.sort(key, reverse=Vrai ou Faux)	l'ordre inverse de celui donné par la clé TABLEAU 1.

## Opérations propres aux listes

```
sage : une_liste = [3, pi, -I, (x+1)*(x-1)] sage :
une_liste.remove(x**2 - 1) sage :
une_liste [3, pi,
-I]
```

- .sort() diffère de sorted() en ce qu'il ne copie pas d'abord la liste et renvoie None.

La liste est triée « sur place ». Vous pouvez considérer que sorted() laisse la liste d'origine intacte, et .sort() comme une modification de la liste.

Ensembles et ensembles figés. Les méthodes du tableau 2 s'appliquent uniquement aux ensembles et aux ensembles figés. Opérations qui modifient l'ensemble ne s'appliquent pas aux ensembles gelés.

Plusieurs méthodes correspondent à des opérations ou relations mathématiques. Il est important de distinguer les méthodes se terminant par \_update qui modifient l'ensemble lui-même et renvoient None, tandis que les méthodes correspondantes, sans \_update, renvoient un nouvel ensemble et laissent l'ensemble d'origine intact.

Dictionnaires. Nous n'utiliserons pas la plupart des fonctionnalités d'un dictionnaire. Les seules mentionnées sont présentées dans le tableau 3. Nous avons déjà montré comment accéder aux éléments d'un dictionnaire à l'aide de l'opérateur crochet [] ; nous précisons donc simplement que vous pouvez ajouter ou modifier des entrées dans un dictionnaire de la même manière que dans une liste.

S.add(élément)	ajouter un élément à l'ensemble S
S.difference(C)	renvoie une copie de l'ensemble ou de l'ensemble gelé S, moins tous les éléments de la collection C
S.difference_update(C)	supprime les éléments de la collection C de l'ensemble S lui-même
Intersection sud (C)	renvoie l'ensemble des éléments de l'ensemble ou de l'ensemble gelé S et de la collection C supprime
S.intersection_update(C)	de l'ensemble S tous les éléments de C
S.isdisjoint(C)	Vrai si et seulement si l'ensemble ou l'ensemble gelé S n'a aucun élément en commun avec la collection C
S.issubset(C)	Vrai si et seulement si tous les éléments de l'ensemble ou de l'ensemble gelé S sont également dans la collection C
S.superset(C)	Vrai si et seulement si tous les éléments de la collection C sont également dans l'ensemble ou l'ensemble gelé S
S.pop()	supprime et renvoie un élément de l'ensemble S
S.remove(élément)	supprime l'élément s'il apparaît dans S ; déclenche un KeyError si ce n'est pas le cas
S.difference_symétrique(C)	renvoie la différence symétrique entre S et T ; c'est-à-dire les éléments non communs à S et T
S.symmetric_difference_update(C)	supprimer de S tout élément qui apparaît dans C, et ajoute à S les éléments de T qui ne sont pas dans S
S.union(C)	renvoie l'union de l'ensemble ou de l'ensemble gelé S avec la collection C
S.mise à jour (C)	ajoute tous les éléments de la collection C à l'ensemble S TABLEAU

## 2. Opérations propres aux ensembles

```
sage: a_dict
{x^2 + 1: 'bonjour', x: 2} sage:
a_dict[0] = 'au revoir' sage: a_dict[0]
= -3 sage: a_dict {0: -3, x^2
+ 1: 'bonjour', x:
2}
```

D.clear()	supprimer toutes les entrées dans D
D.has_key(clé)	renvoie True si et seulement si la clé a une valeur dans D
D.pop(clé)	supprime l'entrée pour la clé de D et renvoie sa valeur
D.popitem()	supprimer une entrée de D et renvoyer le tuple (clé, valeur)
D.mise à jour(E)	ajoute les définitions de E à D

TABLEAU 3. Opérations propres aux dictionnaires

Un assistant pour créer des collections. Nous avons déjà utilisé la procédure range() , mais nous n'avons pas encore abordé sa signification. Son but est de créer un objet range , la méthode utilisée par Sage pour suivre une séquence d'entiers. Voici trois façons de créer une plage : range(b) la

séquence 0, 1,..., b - 1 ; range(a,b) la séquence a, a + 1,..., b -	
1 ; a, a + d,..., a + kd où a + kd < b et a + (k + 1) d ≥ b	
plage(a,b,d)	

Vous pouvez considérer a comme une valeur de « départ », b comme une valeur d'« arrêt » et d comme une valeur d'« étape », de sorte que range(-3,8,3) représente la séquence finie où a = -3, b = 8 et d = 3, ou

$$-3, -3 + 3, -3 + 2 \times 3, -3 + 3 \times 3 \rightarrow -3, 0, 3, 6$$

Notez que 8 n'est pas un élément de la séquence ; en effet, b n'est jamais un élément de la séquence.

La procédure range() ne crée pas une séquence concrète d'entiers !<sup>5</sup> Elle crée plutôt un objet range : le début et la fin d'une séquence, le long de l'étape d'un élément à l'autre.

```
sage: L = range(1, 10, 2) sage: L range(1, 10, 2) sage:
L.start 1

sage : L.stop 10

sage : L.étape 2

sauge : 5 po L
Vrai
sauge : 6 po L
FAUX
```

<sup>5</sup>C'était le cas dans les versions antérieures de Python, et donc de Sage, mais cela a changé avec Python 3.

Erreurs lors de la création ou de l'accès aux collections. Il est judicieux d'analyser les types d'erreurs qui surviennent lors de la création d'une collection ou de l'accès à un élément.

- Lorsque vous créez

une collection à l'aide d'une commande plutôt que de symboles, n'oubliez pas de fournir une autre collection. Sage n'acceptera pas une simple séquence d'éléments séparés par des virgules.

ments.

```
sage : nouveau_tuple = tuple(2, 3, 4)
TypeError : tuple() prend au plus 1 argument (3 donnés)
```

- Les tuples sont immuables. N'essayez pas de modifier leurs valeurs. Si vous pensez devoir modifier la valeur d'un élément, utilisez plutôt une liste.

```
sage : a_tuple[0] = 1
TypeError : l'objet « tuple » ne prend pas en charge l'affectation d'éléments
```

- Bien que les listes et les tuples acceptent des indices négatifs, ils ne s'enroulent pas plus que cela. Si la collection a une longueur n, n'essayez pas d'accéder aux éléments n ou -n – 1.

```
sage : len(a_list) 3

sage : a_list[3]
IndexError : index de liste hors limites
```

- Comme mentionné précédemment, n'essayez pas d'ajouter une liste à un tuple, ou vice-versa.

```
sage : une_liste + un_tuple
TypeError : ne peut concaténer qu'une liste (pas un « tuple ») à une liste
```

- Les ensembles ne sont pas indexés, vous ne pouvez donc pas accéder à un élément particulier d'un ensemble.

```
sage : len(a_set) 3

sage : a_set[1]
TypeError : l'objet 'set' ne prend pas en charge l'indexation
```

- Sage génère une erreur chaque fois que vous essayez d'accéder à une entrée inexistante d'un dictionnaire.

```
sage : a_dict[12]
KeyError : 12
```

### Répéter sur une collection

Revenons maintenant à l'objet principal de ce chapitre : le mot-clé « for » de Sage . Rappelons que sa forme générale est

pour la variable dans la collection :

suivi d'une liste de tâches indentées ; ceci correspond au pseudo-code de variable  
collection suivi d'une liste

de tâches indentées. Lors du premier passage dans la boucle, variable prend la valeur du « premier » élément de collection ; à chaque passage suivant, variable prend la valeur de l'élément de collection qui « suit » sa valeur courante. Nous appelons variable la variable de boucle et collection le domaine de boucle, ou simplement « domaine ».

Ainsi, si l'élément est dans le domaine C, la boucle for affectera à un moment donné sa valeur à la variable, puis exécutera toutes les tâches indentées. Cette propriété d'un traitement complet du C ne change pas, même si vous modifiez la variable pendant la boucle ; Sage mémorise l'élément sélectionné dans le domaine C et sélectionne l'élément qui le suit, plutôt que la valeur de la variable. Vous pouvez donc modifier la variable de la boucle si nécessaire, sans vous soucier du comportement de la boucle.

D'un autre côté, si vous modifiez le domaine, des choses très désagréables peuvent se produire. La boucle for peut parcourir chaque élément de son domaine, mais elle n'en fait pas de copie au départ. Ainsi, si le corps de la boucle modifie le domaine, des choses étranges peuvent se produire, y compris une boucle infinie.<sup>6</sup> Un des exercices de programmation illustrera cela : vous ne devriez pas l'essayer dans Sage à moins d'être prêt à appuyer sur le bouton Arrêter (cloud), à sélectionner le menu Action, puis Interrompre (autre serveur), ou à maintenir la touche Ctrl enfoncée et à appuyer sur C (ligne de commande).

Il existe de nombreux cas dans lesquels vous souhaitez utiliser la valeur de la variable de boucle.

Premier exemple : le calcul d'une intégrale de Riemann. Rappelons qu'il n'est pas toujours possible de simplifier une intégrale indéfinie à sa forme élémentaire. Ainsi, lorsqu'on a besoin d'une intégrale définie, on peut en approcher la valeur par plusieurs méthodes. L'intégrale est définie ainsi :

$$\int_a^b f(x) dx = \lim_{n \rightarrow \infty} \sum_{i=1}^n f(x_i) \Delta x, \text{ où } \Delta x = \frac{b-a}{n} \quad \text{et } x_i \text{ se trouve dans le } i\text{ème sous-intervalle.}$$

En clair, la valeur de l'intégrale est la limite des sommes de Riemann. Ces sommes approximent l'intégrale en utilisant n rectangles de largeur  $\Delta x$  et de hauteur  $f(x)$ . Il existe trois manières habituelles de sélectionner  $x$  : • par l'extrémité gauche, :

$$\begin{aligned} &\text{où } x = a + (i-1)\Delta x; \\ &\bullet \text{ par l'extrémité droite, où } x = a \\ &+ i\Delta x; \\ &\bullet \text{ par le milieu, où } x = a + (i-1/2)\Delta x. \end{aligned}$$

Lorsque n augmente, l'erreur diminue, il est donc possible d'approximer l'intégrale en évaluant

$$\sum_{i=1}^n f(x_i) \Delta x$$

pour une grande valeur de n. Le symbole de sommation  $\Sigma$  nous demande de laisser la variable de sommation i croître de 1 à n, et pour chaque valeur d'évaluer l'expression à sa droite, et d'ajouter cela à la somme croissante.

<sup>6</sup>Cela diffère grandement de nombreux langages informatiques, tels que C, C++ et Java. Dans ces langages, modifier la variable de boucle pendant une boucle for peut avoir des conséquences catastrophiques.

Si vous avez besoin d'une définition de « boucle infinie », consultez [l'index](#), où nous avons volé une blague du glossaire du manuel Amiga-DOS 3.1.

Notez le langage utilisé ici : « each » et « every ». Lorsque la solution d'un problème implique ce type de mots, c'est un signe révélateur qu'il faut une boucle définie sur l'ensemble dont les éléments sont en question.

Pseudo-code. Dans ce cas, nous allons créer une boucle d'approximation du point d'extrême gauche. Il est relativement facile de la transformer en pseudo-code :

```

algorithme Left_Riemann_approximation entrées •
a, b   •
      f , une
fonction intégrable sur [a, b] • n, le nombre de
rectangles à utiliser pour approximer les sorties
      b
       $\int_a^b f(x) dx$ 

• S, la somme de Riemann de l'extrême gauche      b
       $\int_a^b f(x) dx$ , en utilisant n rectangles
pour approximer l'aire

soit  $\Delta x = (b - a) / n$  soit S
= 0 pour i
(1,..., n) soit x =
a + (i - 1)  $\Delta x$   $\Delta x$  à S
ajouter un
retour fx S

```

Voyons ce que fait ce code. Il affecte initialement  $\Delta x$ , que nous devons absolument connaître, car il ne fait pas partie de l'entrée. Il initialise ensuite le résultat, S, à 0, ce qui est judicieux pour créer une somme. Une fois cette opération terminée, il passe dans la boucle et attribue à i chaque valeur de 1 à n.

Avec cette valeur, il effectue les deux tâches en retrait ci-dessous : choisir une valeur de x selon la formule pour les points d'extrême gauche et ajouter l'aire d'un rectangle à S. Une fois que la boucle a parcouru chaque valeur 1, ..., n, elle renvoie S.

Notez comment ce code utilise la valeur de la variable de boucle i pour construire  $x = a + (i - 1) \Delta x$ . L'implémentation Sage x. L'implémentation de ce pseudo-code dans Sage nécessite quelques modifications mineures. Le principal problème est que nous ne pouvons pas utiliser de symboles grecs, d'indices ou de décorateurs, donc les noms des variables doivent quelque peu changer.

Cependant, nous devons effectuer une autre modification qui peut facilement passer inaperçue. La formule que nous utilisons suppose que i prend les valeurs 1, ..., n. La méthode naturelle pour que Sage transmette ces nombres est la commande range(), mais par défaut, range(n) commence à 0 et se termine à n - 1 ! Il existe plusieurs façons de s'assurer d'avoir les bons nombres ; nous optons pour range(1,n+1). Autrement dit, nous voulons commencer à 1 et terminer à n + 1 ; cela inclut 1, mais pas n + 1.

```
sage : def approximation_de_Riemann_gauche(f, a, b, n, x=x) : f(x) = f

    Delta_x = (b - a) / n
    S = 0
    pour i dans la plage (1, n + 1) :
        # extrémité gauche de l'intervalle xi =
        a + (i - 1) * Delta_x # ajoute l'aire du
        rectangle sur l'intervalle
        S = S + f(xi) * Delta_x
    retour S
```

Cela fonctionne plutôt bien :

```
sage : approximation_de_Riemann_gauche(t^2 + 1, 0, 1, 100, t)
```

Deuxième exemple : vérifier si est un corps. Pour un autre exemple, rappelez-vous l'exercice p. 51, où nous devons vérifier si est un corps. Nous savons déjà qu'il s'agit d'un anneau ; il nous suffisait donc de vérifier que tout élément non nul possède un inverse multiplicatif. Vérifier tous les éléments manuellement est assez fastidieux, surtout lorsque n augmente. Une boucle serait donc un outil utile ici. Il suffit de vérifier si chaque élément possède un inverse, et nous pouvons le faire en vérifiant le produit de chaque élément par tous les autres.

Notez encore que nous utilisons les mots « chaque » et « tous », ce qui nous indique que nous devons utiliser une boucle définie.

Pseudocode. Nous décrivons une implémentation assez simple de notre solution dans pseu-docode:8

```
algorithme produire_tous_les_produits
entrées •
n      avec n ≥ 2
sorties •
une liste L telle que Li est l'ensemble des produits de i      éléments de      n      avec tous les autres
n
faire
Soit L = []
pour i      n
    Soit M =
        pour j      n
            ajouter ij à
        M ajouter M à L
    renvoyer L
```

---

Ce n'est pas une solution idéale. Nous l'améliorerons lorsque nous discuterons de la prise de décision.

Voyons ce que fait ce code. Il crée une liste L, initialement vide. (Notre pseudo-code utilise des crochets pour désigner une liste, et des crochets vides pour désigner une liste vide. Certains auteurs utilisent des parenthèses, mais ce n'est pas une règle absolue, et nous ne voulons pas risquer de confusion avec les tuples.) Le code boucle ensuite sur tous les éléments de n , appelant l'élément de chaque passage i. Pour chacun de ces éléments, il crée un nouvel ensemble M, initialement vide. Il boucle ensuite à nouveau, appelant l'élément de chaque passage j. Il est important de noter que i reste fixe tandis que cette boucle interne modifie j à chaque passage ; de ce fait, nous pouvons affirmer avec certitude que M contient le produit de ce i fixe par tous les autres éléments de une fois la boucle interne terminée.

Le code ajoute ensuite cet ensemble à L, concluant ainsi les tâches de la boucle externe. Une fois que la boucle externe a parcouru chaque élément de... Notez que  $n$ , le code peut renvoyer L.

le code utilise les valeurs des variables de boucle i et j, qui sont elles-mêmes des entrées.  
de  $n$ .

En conservant L sous forme de liste, nous garantissons que ses éléments correspondent à l'ordre dans lequel i passe par les éléments de . Ce n'est pas si important pour M ; nous voulons simplement connaître les produits de i et de chaque autre élément, pas nécessairement leur ordre, bien que cela puisse être utile dans certains contextes.

Ce pseudo-code présente une propriété importante des boucles : l'imbrication. Elle se produit lorsqu'une structure de contrôle est incluse dans une autre. C'est souvent utile, mais cela peut aussi être source de confusion ; il est donc conseillé d'éviter de trop en faire. Si votre imbrication dépasse 3 ou 4, il est judicieux de séparer les boucles internes dans une autre procédure. Cela facilite la compréhension du code et, comme les tâches sont souvent utilisables à plusieurs endroits, cela peut également vous faire gagner du temps.

Implémentation Sage. Il est relativement facile de traduire cela en code Sage. La principale différence avec le pseudo-code réside dans la nécessité de créer un anneau et de s'assurer que Sage considère i et j comme des éléments de cet anneau. Nous allons d'abord présenter la méthode « évidente », puis une méthode plus intelligente.

```
sage : def produce_all_products(n) : R =
    ZZ quo(n) #
    résultat final : L[i] est le produit de i L = list() pour i dans la
    plage(n) : M
    = set() # ensemble de
    multiples de i pour j dans la plage(n) :
    M.add(R(i)*R(j))

    L.append(M)
retour L
```

Pour voir comment cela fonctionne, essayez-le avec quelques valeurs de n :

```
sage: produire_tous_les_produits(4) [{0},
{0, 1, 2, 3}, {0, 2}, {0, 1, 2, 3}] sage:
produire_tous_les_produits(5) [{0}, {0, 1,
2, 3, 4}, {0, 1, 2, 3, 4}, {0, 1, 2, 3, 4}, {0, 1, 2, 3, 4}]
```

Nous voyons que :

4, 1 n'apparaît pas dans les produits de 0 et 2. 0 ne nous dérange pas, puisque nous ne nous soucions que des éléments non nuls, mais 2 est fatal : ce n'est pas un corps.

- Pour 5, 1 apparaît dans tous les produits sauf 0, il s'agit donc en fait d'un corps.

Malheureusement, ce code n'est pas une solution idéale, car plus  $n$  augmente, plus les listes de produits s'allongent et deviennent rapides, ce qui rend difficile de vérifier si 1 est un élément d'un ensemble. Cela peut nous amener à nous demander : pourquoi vérifions-nous cela ? Il est facile de demander à Sage de vérifier si un élément apparaît dans une collection. Modifions notre pseudo-code à partir de ceci :

ajouter M à L

à ceci:9

ajouter True à L si 1 ∈ M ; False sinon

Pour le code Sage, nous utilisons le fait que l'expression 1 dans M se simplifie automatiquement (le seul changement apparaît en rouge) :

```
sage : def produce_all_products(n) : R =
    ZZ.quo(n) #
    résultat final : L[i] est le produit de i L = list() pour i dans la
    plage(n) : M
    = set() # ensemble de
        multiples de i pour j dans la plage(n) :
        M.add(R(i)*R(j))

    L.append(1 dans M)
retour L
```

Regardez comment cela se comporte beaucoup plus facilement qu'avant :

```
sage : produire_tous_les_produits(4)
[Faux, Vrai, Faux, Vrai] sage :
produce_all_products(5)
[Faux, Vrai, Vrai, Vrai, Vrai] sage :
produce_all_products(10)
[Faux, Vrai, Faux, Vrai, Faux, Faux, Faux, Vrai, Faux, Vrai]
```

Dans ce cas, il est extrêmement facile de déterminer si autre la  $n$  est un champ : voir si False apparaît quelque part première position (qui correspond à 0, pour laquelle aucun inverse n'est nécessaire).

On pourrait se demander s'il n'est pas possible de simplifier encore davantage. En effet, c'est possible. Une méthode nécessiterait un peu d'algèbre booléenne, nous la remettons donc à plus tard. Mais une autre méthode consiste à observer que Sage considère  $ZZ.quo(n)$  comme une collection et que nous pouvons également parcourir ses éléments : (les modifications apparaissent en rouge).

---

Ce n'est pas encore une solution idéale. Nous l'améliorerons lorsque nous discuterons de la prise de décision.

```
sage : def produce_all_products(n) : R =
    ZZ quo(n) #
    résultat final : L[i] est le produit de i L = list()

pour i dans R :
    M = set() # ensemble de multiples de i pour j
    dans R :
        M.add(i*j)
    L.append(1 dans M)
renvoie L
```

Si vous testez ceci, vous constaterez que cela fonctionne comme avant, même si le code est plus simple. Gardez à l'esprit que Sage facilite souvent le travail avec des objets mathématiques de manière naturelle.

### Compréhensions : répétition dans une collection

Sage propose une méthode spéciale pour créer des collections qui abrège la structure de la boucle `for` et la rend plus facile à utiliser et à lire. Ces méthodes, appelées compréhensions, imitent la notation mathématique des ensembles. Dans cette notation, on définit un ensemble en spécifiant d'abord un domaine, puis un critère de sélection des éléments de ce domaine. Par exemple, l'expression

$$S = \{n \mid 2 \leq n \leq 210\} \quad 10$$

utilise la notation de construction d'ensemble pour définir `S` comme l'ensemble de tous les nombres naturels compris entre 2 et 210, inclusif. Les compréhensions donnent à Sage un moyen naturel d'imiter cela dans les endroits où cela serait utile et faisable.

Pour définir une compréhension, utilisez le modèle suivant :

`collection(expression pour la variable dans collection_or_range)`

Cela équivaut effectivement à l'une des séquences de commandes suivantes (`D` est une collection ; `a`, `b` et `n` sont des entiers) :

```
sage : C = collection() sage :
pour d dans D :
    C = C.append/insert/update(expression)
```

(Ici, vous pouvez utiliser « `collection` » pour n'importe quelle liste ou ensemble, en choisissant d'ajouter, d'insérer ou de mettre à jour de manière appropriée.) Ou :

```
sage : C = collection() sage :
pour d dans la plage(a, b C = , n):
    C.append/insert/update(expression)
```

Dans chaque cas, le résultat est que `C` contient les valeurs prises par l'expression pour chaque valeur de `D` (premier cas) ou de la plage d'entiers spécifiée (deuxième cas).

Pour voir cela en pratique, revenons à notre exemple des sommes de Riemann. Une façon de procéder pourrait Utiliser une méthode de compréhension pour générer les valeurs x. Les extrémités gauches ont la forme

$$a + (i - 1)\Delta x \text{ où } i = 1, \dots, n$$

et nous pouvons attribuer ceci à une liste X de valeurs x en utilisant une compréhension de liste comme

$$X = [a + (i - 1)*\Delta x \text{ pour } i \text{ dans la plage}(1, n+1)] .$$

(N'oubliez pas que nous avons besoin de n+1 car la commande range() continue jusqu'au deuxième nombre mentionné, mais sans l'inclure.) Nous pourrions alors boucler sur X, comme ceci :

```
sage : def approximation_de_Riemann_gauche(f, a, b, n, x=x) : f(x) = f
```

```
Delta_x = (b - a) / n # crée tous les
points d'extrémité gauche en un seul passage
X = [a + (i - 1)*Delta_x pour i dans la plage(1, n+1)]
S = 0
pour xi dans X :
    # ajouter une zone avec une hauteur à f(xi)
    S = S + f(xi) * Delta_x renvoie S
```

C'est un peu plus simple qu'avant et cela a l'avantage de définir les valeurs x d'une manière qui semble mathématique.

Nous pouvons également décrire une méthode plus simple pour contourner la pénalité liée à la création de la liste X. Sage dispose d'une commande sum() facile à comprendre. Plutôt que d'initialiser une variable somme S à zéro et d'y ajouter des sommes partielles à chaque passage dans la boucle, nous pourrions utiliser la commande sum() avec une compréhension de liste pour simplifier le programme et le rendre plus proche du concept mathématique. Puisque nous utilisons des sommes à gauche, nous pouvons réécrire.

$$\sum_{i=1}^n f(x_{i-1}) \Delta x$$

$$\sum_{i=1}^n f(a + i\Delta x) \Delta x$$

qui devient

$$\text{somme}(f(a + (i*\Delta x)) * \Delta x \text{ pour } i \text{ dans la plage}(1, n+1)) .$$

Tout ce que nous avons fait ici est de « traduire » l'idée mathématique en une commande Sage correspondante. Cela peut paraître difficile à lire au début, mais une fois habitué, c'est assez naturel. Le code Sage obtenu est

```
sage : def approximation_de_Riemann_gauche(f, a, b, n, x=x) : f(x) = f
```

```
Delta_x = (b - a) / n renvoie la
somme (f(a + (i*Delta_x)) * Delta_x \
pour i dans la plage(1, n+1))
```

C'est beaucoup plus court que ce que nous avions avant.

#### Animation à nouveau

Rappelons que, p. 70 , nous avons créé une courte animation d'une fleur de lys modifiée en créant plusieurs images, puis en les joignant avec la commande `animate()` . La plupart des animations intéressantes ou instructives nécessitent un nombre important d'images ; les créer à la main peut s'avérer fastidieux, voire impossible. D'autre part, la plupart des animations dépendent aussi de motifs ; par exemple, le motif de notre animation p. 70 peut se résumer à  $\cos(nx)\sin((n + 1)x)$ , où n est compris entre 2 et 7 inclus. Bien sûr, nous pourrions vouloir plus d'images. Les

#### compréhensions de listes

permettent de créer une liste d'images, comme celle-ci :

```
sage : cadres = [polar_plot(cos(n*x)*sin((n+1)*x), (x, 0, pi), fill=True, \
épaisseur=2, couleur de remplissage='jaune', \
couleur='verge d'or', axes=False) \ pour n
dans la plage(2,20)] sage:
fdl_anim = animate(images, xmin=-1, xmax=1, \ ymin=-0.5, ymax=1.5,
rapport_aspect=1) sage: show(fdl_anim, gif=True,
delay=8)
```

Le résultat devrait être une animation d'une vitesse et d'une fluidité raisonnables. Si vous visualisez ce texte dans Acrobat Reader, vous devriez voir la même animation ci-dessous, bien que la vitesse puisse légèrement différer. Si vous consultez une version papier, vous devriez voir les images individuelles de l'animation :

#### Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Une instruction `for` implémente une boucle définie.

2. Le concept que nous exprimons en pseudo-code par « répéter n fois » n'a pas de mot-clé correspondant dans la plupart des langages informatiques.
3. Vous ne devez pas modifier la valeur d'une variable de boucle dans une boucle `for`, car cela peut affecter le passage suivant à travers la boucle.
4. Vous ne devez pas modifier les entrées du domaine d'une boucle `for`, car cela peut affecter le passage suivant la boucle.
5. Les boucles définies ne sont pas un modèle utile pour l'itération.
6. La seule structure immuable que Sage vous propose est un tuple, donc si vous voulez l'immuabilité, vous êtes coincé avec l'indexation.
  
7. Vous ne pouvez construire un ensemble qu'en utilisant des accolades ; par exemple, `{3, x^2,-pi}`.
8. Vous ne pouvez créer un ensemble vide que sur un ordinateur dont le clavier comporte un symbole.
9. La clé de tri renvoie `True` si et seulement si le premier argument est plus petit que le second.
10. Les compréhensions de liste vous permettent de créer une liste sans utiliser la structure de boucle `for` conventionnelle. Ici.

Choix multiple.

1. L'indexation dans les tuples et les listes correspond à quelle notation mathématique ?
  - A. fonctions
  - B. expressions
  - C. indices
  - D. apostrophes
2. Nous ne pouvons pas accéder à un élément d'un ensemble à l'emplacement numérique `i` car :
  - A. Les programmeurs de Sage étaient paresseux.
  - B. Les ensembles ne sont pas ordonnés et ne sont donc pas indexables.
  - C. Cela pourrait casser la boucle `for`.
  - D. La prémissie est incorrecte ; les parenthèses nous permettent d'accéder à un élément d'un ensemble à un niveau numérique emplacement `i`.
3. Nous ne pouvons pas accéder à un élément d'un dictionnaire à l'emplacement numérique `i` parce que :
  - A. Les programmeurs de Sage étaient paresseux.
  - B. Les dictionnaires ne sont pas ordonnés et ne sont donc pas indexables.
  - C. Les dictionnaires sont classés par entrée, ou « clé », plutôt que par emplacement numérique.
  - D. La prémissie est incorrecte ; les parenthèses nous permettent d'accéder à un élément d'un dictionnaire à un numéro emplacement géographique `i`.
4. Nous ne pouvons pas accéder à un élément d'un tuple à l'emplacement numérique `i` parce que :
  - A. Les programmeurs de Sage étaient paresseux.
  - B. Les tuples ne sont pas ordonnés et ne sont donc pas indexables.
  - C. Les tuples sont supposés immuables, et accéder à un élément particulier violerait cette règle.
  - D. La prémissie est incorrecte ; les parenthèses nous permettent d'accéder à un élément d'un tuple à une échelle numérique. emplacement `i`.
5. Laquelle des techniques mathématiques suivantes pourrait motiver l'utilisation d'une boucle `for` ?
  - A. itération
  - B. vérification de tous les éléments d'un ensemble
  - C. répétition d'un ensemble de tâches `n` fois
  - D. tout ce qui précède
6. Laquelle des collections suivantes est immuable ?
  - A. un tuple

- B. une liste
- C. un ensemble
- D. une plage()

7. Laquelle des collections suivantes n'est pas indexable ?

- A. un tuple B.
- une liste C.
- un ensemble gelé D.
- une plage()

8. Lequel des symboles ou identifiants suivants utilisons-nous dans notre norme de pseudo-code pour tester l'appartenance à une séquence ou à un ensemble ?

- A. B.
- C. dans
- D. dans

9. Modèle de compréhension quelle notation mathématique ?

- A. fonction B.
- ensemble
- C. constructeur
- d'ensembles D.

indices 10. L'imbrication se produit lorsque :

- A. une ou plusieurs boucles for sont placées à l'intérieur d'une autre B.
- nous créons une collection en utilisant la notation de construction
- d'ensemble C. nous créons un ensemble pour contenir un certain nombre d'éléments du même type D. deux oiseaux s'engagent dans le Sacre du Printemps<sup>10</sup>

Réponse courte.

1. Résumez le comportement de la commande `range()` lorsqu'elle reçoit 1, 2 ou 3 entrées.

2. L'associativité de la multiplication est vraie pour un ensemble S dès lors qu'un triplet d'éléments s, t, u    S satisfait  $s(t)u = (st)u$ . Supposons qu'un ensemble S comporte 9 éléments. Combien de produits faudrait-il pour examiner tous les produits possibles ? Indice : la réponse est suffisamment grande pour que vous ne souhaitiez pas la faire à la main.

3. Considérez le code Sage suivant.

```
sage : pour i dans la plage(10) :
    pour j dans la plage(i,10) :
        pour k dans la plage(j,10) :
            print(i, j, k)
```

(a) Qu'est-ce qu'il imprime ? Ne donnez pas tout, juste assez pour illustrer le motif. Utilisez mots pour expliquer l'ordre dans lequel les lignes sont imprimées.

(b) Trouvez une inégalité quadruple<sup>11</sup> impliquant i, j et k qui est vraie pour chaque ligne.

4. Ce problème considère le code Sage suivant.

<sup>10</sup>Nous apprécierions que Stravinski les héritiers ne poursuivraient pas.

<sup>11</sup>Une quadruple inégalité a la forme  $a \leq b \leq c \leq d \leq e$ . Outre i, j et k, vous devrez utiliser deux constantes.

```
sage : def ec(k) : var('y')
p =
Graphics() pour i dans
la plage(2, k+1) : p = p +
implicit_plot(x**2 + y**2/(1-sqrt(i))==1, (x,-2,2), (y,-2,2), couleur=(0,i/k,.8-i/k))

retour p
```

(a) Décrivez ce que renvoie l'appel `ec(10)`. (b)

Expliquez ce qui se passe si nous utilisons des valeurs de plus en plus grandes de  $k$  dans l'appel `ec(k)`.

5. Cet exercice examine la question de l'addition d'entiers consécutifs.

(a) Quelle est la formule pour additionner les  $n$  premiers entiers positifs ? (Vous auriez dû voir cela avant, peut-être dans Calcul II dans la section sur les sommes de Riemann.) (b) Suzy écrit la procédure suivante pour additionner les  $n$  premiers entiers positifs.

```
sage : définition de somme(N) : total = 0

pour i dans la plage (N) :
    total = total + i
retour total
```

Quel est le résultat de son invocation, `sum_through(5)` ? Indiquez la valeur de `total` après chaque passage dans la boucle. (c) Utilisez la notation de sommation pour décrire la valeur réellement calculée par le programme de Suzy. Quelle serait la formule correspondante ? (d) Comment Suzy devrait-elle corriger son programme ?

6. Supposons que vous ayez un nourrisson qui demande à être nourri toutes les 3 heures.

(a) Si vous commencez à la nourrir à 7 heures du matin, à quelles heures l'alimentation aura-t-elle lieu ? (b) Rédigez une compréhension qui génère une liste avec chacune de ces heures pour les 24 prochaines heures. (c) Répétez la partie (b), mais en supposant que le nourrisson est un peu plus âgé et ne demande à être nourri que toutes les 5 heures.

Programmation.

- Écrivez une procédure pour calculer la valeur moyenne des éléments d'une collection. (N'oubliez pas que « valeur moyenne » est un terme sophistiqué pour « moyenne ».) Ce sera plus simple avec une compréhension à l'intérieur d'une fonction `sum()`, mais vous pouvez également utiliser une boucle `for`.
- Écrivez une procédure pour calculer la valeur médiane des éléments d'une collection. (N'oubliez pas que « valeur médiane » est un terme sophistiqué pour « valeur moyenne », c'est-à-dire que la moitié des valeurs sont plus grandes et l'autre moitié plus petites.) La meilleure façon de procéder est probablement de convertir la collection en liste, de la trier, puis de renvoyer le nombre central.
- Écrivez une procédure qui prend une liste de points  $(x_i, y_i)$  et renvoie deux listes : la liste des coordonnées  $x$  et la liste des coordonnées  $y$ .
- Créez une animation illustrant l'approche d'une sécante par une tangente. Utilisez la fonction avec la tangente passant  $f(x) = x^2$ , par le point  $x = 2$ , et les sécantes joignant  $x = 2$ .

<sup>12</sup>Eh bien, oui, certains d'entre nous en ont une expérience intime. Qu'est-ce qui vous fait poser cette question ?

Avec 30 points entre  $x = -1$  et  $x = 2$ . Tracez le tracé de  $f$  en noir, avec une épaisseur de 2 ; coloriez la tangente en bleu et les sécantes en rouge. Ajoutez un point bleu en  $(2, 4)$  pour mettre en évidence l'intersection de la courbe, de ses tangentes et sécantes. Le résultat devrait être comparable, voire supérieur, à l'animation ci-dessous si vous lisez ce texte dans Acrobat Reader :

5. À la page 104, nous avons parlé d'une série de Taylor tronquée à 4 termes. Nous ne disposions pas de boucles for, cette approche était donc à la fois fastidieuse et peu flexible. (a) Adaptez le pseudo-code pour que l'utilisateur puisse saisir un nombre arbitraire de termes. (b) Implémentez le pseudo-code sous forme de programme Sage.
6. (Ce problème s'adresse aux étudiants qui ont suivi des cours de calcul multivariable.) L'intégrale double

$$\int_D f(x, y) dx$$

Ce problème apparaît fréquemment en calcul tridimensionnel. Ici,  $D$  est un sous-ensemble du plan  $\mathbb{R}^2$ , appelé domaine de l'intégrale. Lorsque  $D$  est une région rectangulaire définie par les intervalles  $x \in (a, b)$  et  $y \in (c, d)$ , on peut approximer cette intégrale en divisant le domaine en  $m \times n$  sous-rectangles et en évaluant la valeur  $z$  de  $f$  sur chaque sous-rectangle :

$$\int_D f(x, y) dx \approx \sum_{i=1}^m \sum_{j=1}^n f(x_i, y_j) \Delta x \Delta y$$

où •  $m$

- $m$  est le nombre de sous-intervalles que nous voulons le long de  $(a, b)$  ;
- $n$  est le nombre de sous-intervalles que nous voulons le long de  $(c, d)$  ;
- $\Delta x = \frac{(b-a)}{m}$  est la largeur de chaque sous-intervalle de  $(a, b)$  ;
- $\Delta y = \frac{(d-c)}{n}$  est la largeur de chaque sous-intervalle de  $(c, d)$  ; et
- $x_{i,j}, y_{i,j}$  est un point dans le sous-rectangle défini par le  $i$ ème sous-intervalle de  $(a, b)$  et le  $j$ ème sous-intervalle de  $(c, d)$ .

Nous pouvons sélectionner  $x_{i,j}, y_{i,j}$  en utilisant une règle du point médian comme

$$x_{i,j} = a + i - 2 - \Delta x \text{ et } y_{i,j} = c + j - 2 - \Delta y.$$

Écrivez une procédure `double_integral_midpoint()` qui prend comme arguments  $f$ ,  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $m$  et  $n$ , et renvoie une approximation de l'intégrale double de  $f$  sur  $(a, b) \times (c, d)$  en utilisant  $m$  sous-intervalles le long de  $(a, b)$  et  $n$  intervalles le long de  $(c, d)$ .

Astuce : ce code sera très similaire à celui que nous avons utilisé pour approximer une intégrale simple, mais vous devrez imbriquer une boucle for pour  $y$  dans une boucle for pour  $x$ . Vous pouvez également le faire avec une compréhension, mais c'est un peu plus compliqué.

7. Soit  $n$  un entier positif. La factorielle de  $n$ , notée  $n!$ , est le produit  $n! = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1$ .

Sage propose déjà une procédure `.factorial()`, mais dans cet exercice, vous utiliserez des boucles pour la réaliser, de deux

manières différentes. (a) La commande `product()` de Sage fonctionne de manière similaire à la commande `sum()` : elle calcule le produit de ce qui se trouve entre les parenthèses, et vous pouvez utiliser une compréhension pour spécifier une plage, plutôt que d'en créer une explicitement. Cela correspond à l'expression mathématique.

 $n$ 

$$\prod_{i=1}^n f(i)$$

qui calcule le produit de tous les  $f(i)$  où  $i = 1, 2, \dots, n$ . Dans cette notation,

$$n! = \prod_{i=1}^n i = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1.$$

Écrivez une procédure appelée `factorial_comprehension(s)` qui accepte un argument,  $n$ , puis construit le produit à l'aide d'une compréhension.

- (b) Une méthode plus traditionnelle pour calculer la factorielle consiste à initialiser une variable `produit P` à 1 (le « produit vide »), puis à effectuer une boucle définie de 1 à  $n$ , en multipliant `P` par chaque nombre au passage. Écrivez une procédure appelée `factorial_loop()` qui accepte un argument,  $n$ , puis construit le produit à l'aide d'une boucle définie.

8. Écrivez un programme pour calculer la factorielle croissante d'un nombre  $n$  en  $k$  étapes. La factorielle croissante  $rf(n, k)$  est

$$rf(n, k) = n \times (n + 1) \times \cdots \times (n + k - 1).$$

Bien que  $k$  soit toujours un entier positif, votre programme devrait fonctionner même si  $n$  ne l'est pas. Par exemple,  $rf(1/2, 5) = (1/2)(3/2)(5/2)(7/2)(9/2) = 945/32$ .

9. Écrivez un programme pour calculer la factorielle décroissante d'un nombre  $n$  en  $k$  étapes. La factorielle décroissante  $ff(n, k)$  est

$$ff(n, k) = n \times (n - 1) \times \cdots \times (n - (k - 1)).$$

Bien que  $k$  soit toujours un entier positif, votre programme devrait fonctionner même si  $n$  ne l'est pas. Par exemple,  $ff(9/2, 5) = (9/2)(7/2)(5/2)(3/2)(1/2) = 945/32$ .

10. Une permutation est une façon d'ordonner des objets distincts. Par exemple, dans la liste  $(1, 2, 3)$ , il y a six permutations :  $(1, 2, 3)$  elle-même, puis  $(2, 1, 3)$ ,  $(3, 2, 1)$ ,  $(3, 1, 2)$ ,  $(2, 3, 1)$  et  $(3, 2, 1)$ . Il arrive que l'on souhaite permuter seulement  $c$  objets d'un ensemble de  $n$ . La formule est la suivante :

$$n \text{ } P_c = \frac{n!}{c!}.$$

(Voir l'exercice précédent pour les factorielles.) Il y a deux façons de faire cela. (a)

La façon évidente, « par force brute », de le faire est de calculer  $n!$  et  $c!$ , puis de diviser. Écrivez une procédure Sage nommée `n_P_c_brute()`, qui accepte les deux arguments  $n$  et  $c$ , puis appelle soit `factorial_loop()` soit `factorial_comprehension()` pour déterminer  $n!$  et  $c!$ . (b) Une façon plus intelligente de faire cela est d'utiliser votre tête. Développez à la main les factorielles dans  $n \text{ } P_c$  pour voir le modèle.

Transformez la formule mathématique résultante en une procédure Sage nommée `n_P_c_smarter()`, qui accepte les deux arguments  $n$  et  $c$ , puis détermine  $n \text{ } P_c$  sans calculer  $n!$  ou  $c!$ .

11. Supposons que  $b > 1$  soit un entier. Pour écrire un nombre positif  $n$  en base  $b$ , on peut répéter la formule suivante : étapes suivantes :

- Soit  $d = \log_b n + 1$  ; cela nous indique combien de chiffres le résultat aura.
- Soit  $L$  une liste vide.
- Répétez d fois :
  - Soit  $r$  le reste de  $n$  divisé par  $b$ .
  - Soustraire  $r$  de  $n$ .
  - Remplacez  $n$  par la valeur de  $n$  lorsqu'il est divisé par  $b$ .
  - Ajoutez  $r$  à  $L$ .

Inversez  $L$  et renvoyez le résultat.

Convertissez cette liste informelle d'instructions en pseudo-code formel et traduisez le pseudo-code en code Sage.

12. Écrivez une procédure Sage nommée `by_degree_then_lc()` qui, lorsqu'on lui donne un polynôme  $f$  en entrée, renvoie un doublet composé du degré du polynôme, suivi de son coefficient dominant. Nous ne vous avons pas encore présenté les commandes Sage pour cela, mais elles fonctionnent comme des méthodes pour un polynôme ; vous pouvez le trouver de la manière suivante :

```
sage : f = 3*x**2 + x + 1 sage :
f.<Tab>
```

...où `<Tab>` indique d'appuyer sur la touche Tab du clavier. Les deux méthodes prennent comme argument l'indéterminé, qui est une autre méthode d'un polynôme que l'on peut trouver de la même manière. Assurez-vous de pouvoir utiliser ces méthodes avec succès avant d'écrire le programme.

Testez minutieusement cette procédure. Elle devrait produire les résultats suivants :

```
sage : f = 3*x**2 + x + 1 sage :
par_degré_alors_lc(f) (2, 3) sage : g
= 2*t**4
- t**2 sage : par_degré_alors_lc(g)
(4, 2)
```

Une fois que vous l'avez fait fonctionner, assurez-vous qu'il trie la liste suivante de polynômes en  $t$  dans le bon ordre. Notez que, puisque nous utilisons  $t$  dans cette liste, votre procédure ne peut pas présumer de l'indéterminée ; elle doit donc également être un argument de la procédure, bien que la valeur par défaut soit  $x$ .

`[ t^2 + t + 1, 2*t - 1, 3*t^2 + 4, -3*t^10 + 1 ]`

## CHAPITRE 6

## Résolution d'équations

Jusqu'ici, nous vous avons demandé de résoudre les équations manuellement, mais il est logique que nous souhaitions que Sage les résolve pour nous. Ce chapitre examine la commande `solve()` et certains de ses dérivés, tant sur des équations simples que sur des systèmes d'équations. Ceci nous amène aux matrices, dont nous examinerons également certaines.

Il n'est généralement pas possible de décrire la solution exacte de chaque équation en termes « purement algébriques », c'est-à-dire en utilisant l'arithmétique et les radicaux avec des nombres rationnels. Cela est généralement vrai même lorsque l'équation est constituée uniquement de polynômes ! Ainsi, même si nous nous concentrons principalement sur les méthodes permettant de trouver des solutions exactes, Sage propose également des méthodes permettant de calculer des solutions approximatives aux équations, et nous les examinons brièvement.

## Les bases

Pour trouver la solution exacte d'une équation ou d'un système d'équations, l'outil principal dont vous avez besoin est la commande `solve()`.

<code>résoudre(eq_or_ineq,indet)</code>	résout l'équation unique ou l'inégalité <code>eq_or_ineq</code> pour <code>indet</code> résout la
<code>résoudre(eq_or_ineq_list,indet_list)</code>	collection d'équations <code>eq_or_ineq_list</code> pour les indéterminées répertoriées dans la collection <code>in-det_list</code>

Alors que `eq_or_ineq` devrait impliquer une équation ou une inégalité, du point de vue de Sage, nous pouvons utiliser une expression, une fonction, une équation ou une inégalité.

Une équation ou une inégalité, pour une indéterminée

La commande `solve()` résoudra bien sûr les problèmes d'algèbre de base du lycée.

Équations.

```
sage : résoudre(x**2 - 1 == 0, x) [x ==
-1, x == 1]
```

Il est très courant de rencontrer des équations avec 0 d'un côté ; la solution de ce type d'équation est appelée racine. Il n'est pas nécessaire de spécifier qu'une expression est égale à 0 pour trouver une racine ; il suffit de fournir l'expression, et la commande `solve()` en déduit que nous voulons ses racines.

```
sage : résoudre(x**2 - 1, x) [x ==
-1, x == 1]
```

Sage peut également résoudre des équations où tous les coefficients sont symboliques.

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

Le polynôme  $ax^2 + bx + c$  est dit quadratique car son degré est 2,1. Vous pouvez distinguer la formule quadratique dans les deux réponses :

$$x = -\frac{1}{2} \times \frac{b + \sqrt{b^2 - 4ac}}{2a} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

et

$$x = -\frac{1}{2} \times \frac{b - \sqrt{b^2 - 4ac}}{2a} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Cette réponse correspond à celle que vous avez apprise à l'école,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Dans les deux cas, le résultat était une liste d'équations. Chaque équation indique une valeur de l'indéterminée qui permettra de la résoudre. Le résultat étant sous forme de liste, les solutions sont accessibles entre crochets `[]`. Pour manipuler les solutions, il est préférable d'affecter la liste à une variable, généralement appelée « `sols` », puis d'utiliser « `sols[i]` » pour accéder à la ième solution.

```
sage : sols = résoudre(x**5 + x**3 + x, x) sage :
len(sols)
5
sage : sols[0] x
== -sqrt(1/2*I*sqrt(3)) - 1/2
```

Donc  $x = -\sqrt[5]{3}/2 - 1/2$  est l'une des solutions.

Parfois, vous souhaiterez simplement connaître la valeur de la solution. Les équations Sage offrent deux méthodes utiles : `rhs()` et `lhs()` qui extraient les côtés gauche et droit :

<code>eq_or_ineq.rhs()</code>	côté droit de <code>eq_or_ineq</code>
<code>eq_or_ineq.lhs()</code>	côté gauche de <code>eq_or_ineq</code>

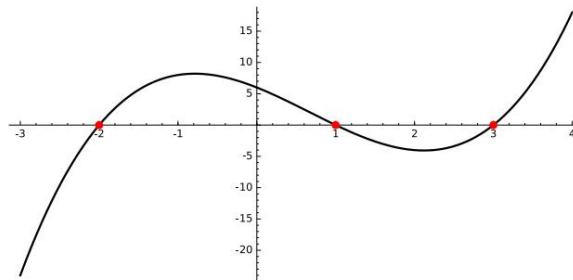
Poursuivant notre exemple précédent,

<sup>1</sup>Le degré d'un polynôme dans une seule indéterminée est le plus grand exposant qui apparaît sur cette indéterminée. La méthode `.degree()` vous le signalera.

```
sage : sols[0].rhs()
-sqrt(1/2*I*sqrt(3) - 1/2)
```

Comme le résultat de `solve()` est sous forme de liste, vous pouvez également parcourir la liste des solutions à l'aide d'une boucle `for`. Voici une façon de procéder : traçons un polynôme et ses racines.

```
sage : f(x) = x^3 - 2*x^2 - 5*x + 6 sage : sols =
solve(f, x) sage : X = [sol.rhs()
pour sol dans sols] sage : p = plot(f, min(X) - 1,
max(X) + 1, color='black', \
épaisseur=2)
sage : pour xi dans X :
p = p + point((xi,0), taille des points=60, couleur='rouge', \
zorder=5)
sage : p
```



Vous auriez dû être capable de le faire seul, mais cela implique de rassembler de nombreuses idées différentes. Décomposons donc chaque étape. • Nous avons

d'abord défini la procédure `f`, un polynôme de degré 3. • Nous avons utilisé `solve()` pour trouver ses racines et les avons stockées dans la variable `sols`. Si vous jetiez un œil à `sols`, vous verriez ce qui suit :

```
sauge : sols
[x == 3, x == -2, x == 1]
```

• Sage renvoie les solutions sous forme d'une liste d'équations, et nous souhaitons uniquement la valeur de la racine. Pour ce faire, nous avons affecté à une nouvelle variable, `X`, une liste formée par compréhension. L'expression `pour sol dans sols` parcourt la liste stockée dans `sols` et stocke chaque valeur dans la variable `sol`. L'expression `sol.rhs()` nous donne le côté droit de cette valeur.

Le résultat est que `X` est une liste qui contient le côté droit de chaque solution dans `sols`. Si vous jetiez un œil à `X`, vous verriez ce qui suit :

```
sage : X
[3, -2, 1]
```

- Nous avons défini un objet graphique  $p$  comme le tracé de  $f$  sur l'intervalle  $\min(X) - 1$  et  $\max(X) + 1$ . Cela donne au graphique un peu d'espace pour respirer au-delà des racines.
- Nous avons effectué une boucle définie sur les solutions stockées dans  $X$ . La boucle stocke l'entrée de  $X$  dans la variable de boucle  $xi$ , que Sage utilise pour ajouter un point à  $p$ . Par exemple, lors du premier passage dans la boucle, il attribue  $xi=3$ , donc il ajoute le point à  $(3, 0)$ .

Multiplicités. Les expressions sages offrent également une méthode qui nous fournit parfois les deux racines et multiplicités.

<code>f.roots()</code>	renvoie les racines de $f$ , ainsi que leurs multiplicités.
------------------------	---

Les multiplicités, rappelons-le, nous indiquent combien de fois une racine apparaît dans la factorisation d'un polynôme. Par exemple, le polynôme  $(x - 1)(x + 2)^2(x - 4)^3$  a trois racines,  $1^1, (-2)^2$  et  $4^3$ , de multiplicités respectives 1, 2 et 3.

```
sage : f = (x - 1)*(x + 2)**2*(x - 4)**3 sage : f.roots() [(-2, 2), (1, 1), (4, 3)]
```

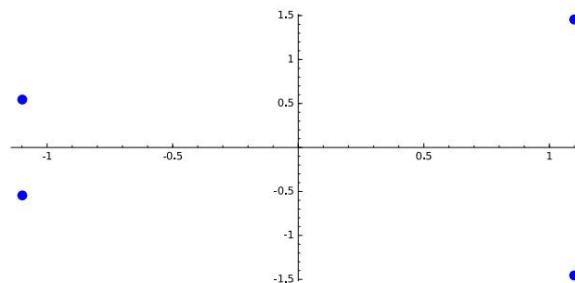
Vous voyez que le résultat est une liste de tuples ; chaque tuple répertorie d'abord la racine, puis sa multiplicité. Les multiplicités sont importantes pour de nombreuses raisons ; dans l'un des exercices de ce chapitre, vous pouvez examiner la zone autour des racines d'un polynôme et voir comment la multiplicité affecte la géométrie.

Le plan complexe. De nombreux polynômes ont des racines qui incluent des parties imaginaires. Il est évidemment impossible de représenter graphiquement ces racines avec leurs fonctions sur le plan réel, car il n'y a pas de place sur ce plan pour inclure, par exemple, le nombre  $i$ .

Néanmoins, il peut être très instructif de visualiser les racines sans leurs fonctions. Tout nombre complexe a la forme  $a + bi$ , où  $a$  est la partie réelle et  $b$  la partie imaginaire. On représente cela comme le point  $(a, b)$  sur le plan réel. Cette application attribue un point unique sur le plan réel à tout nombre complexe  $z$ , ce qui justifie le nom de ce modèle, le plan complexe.

La procédure suivante mappe un nombre complexe à un point sur le plan réel.

```
sage : def complex_point(z, *args, **kwds) : return
        point((real_part(z), imag_part(z)), *args, **kwds) sage : sum(complex_point(sol.rhs(),
points=90) \
pour sol dans solve(x**4 + 4*x + 5, x))
```



Que fait ce code ?

- Les deux premières lignes définissent une procédure, `complex_point()`, dont le seul argument requis est `z`, un nombre complexe.
    - Il crée un point dont la valeur `x` est la partie réelle de `z` et dont la valeur `y` est l'image partie indivisible de `z`.
    - Pour éviter de reformuler les arguments obligatoires et facultatifs d'un point, `complex_point()` utilise une astuce spéciale pour accepter les arguments obligatoires et facultatifs d'un point() régulier, puis les ignore sauf pour les transmettre à `point()`. •
- Les dernières lignes créent un graphique en utilisant la commande `sum()` avec une compréhension.
- Les boucles de compréhension sur les solutions sur  $x^4 + 4x^2 + 4 = 0$ <sup>2</sup>, qu'il trouve avec la commande `solve(x^4 + 4*x^2 + 4, x)`. Chaque solution est stockée dans la variable de boucle. soleil.
  - La boucle envoie chaque valeur de `sol` à la méthode `.rhs()`, obtenant en retour le membre de droite de l'équation utilisée par Sage pour décrire la solution. La boucle transmet ce nombre complexe à `complex_point()`, avec l' argument optionnel `pointsize=90`. Le résultat est un point dans le plan complexe, que la commande `sum()` combine finalement pour former l'image que vous voyez.

Résolution d'inéquations. Vous vous souviendrez peut-être que les inégalités entraînent des complications. Tout d'abord, il existe généralement une infinité de solutions, situées sur un ou plusieurs intervalles de la droite réelle. Ces intervalles sont parfois bornés, parfois non bornés. Par exemple,  $-1 \geq 0$  est  $(-\infty, -1]$

- la solution de  $x^2 - 1 \geq 0$  est  $[1, \infty)$ , tandis que  $-1 < 0$  est  $(-1, 1)$ .
- la solution de  $x^2 \geq 2$

Cette complexité se reflète dans la façon dont Sage décrit les solutions d'une inégalité. Les solutions d'une inégalité sont décrites dans une liste de listes. Chaque liste interne décrit un intervalle contenant des solutions. Cet intervalle contient soit une inégalité linéaire, qui représente un intervalle non borné dans une direction, soit deux inégalités linéaires, qui représentent un intervalle borné dans les deux directions.

Par exemple, il n'est pas difficile de vérifier à la main que l'inégalité  $(x - 3)(x - 1)(x + 1)(x + 3) \geq 0$  a la solution suivante :

$$(-\infty, -3] \cup [-1, 1] \cup [3, \infty)$$

Ainsi,  $x = -5$ ,  $x = 0$  et  $x = 12$  sont des solutions. Ceci est facile à représenter sur une droite numérique :



D'après la description donnée dans le paragraphe précédent, à quoi devrait ressembler la solution ? Il s'agit d'une liste de listes, il doit donc y avoir plusieurs paires de parenthèses dans une seule paire de parenthèses. Il y a trois intervalles, donc trois listes internes. Les intervalles les plus externes étant non bornés, Sage les décrira à l'aide d'une seule inégalité linéaire ; l'intervalle intermédiaire étant borné, Sage utilisera deux inégalités linéaires pour le décrire.

```
sage : résoudre((x-3)*(x-1)*(x+1)*(x+3) >= 0, x) [[x <= -3], [x >= -1, x <= 1], [x >= 3]]
```

---

<sup>2</sup>Il ne devrait pas être inutile de réfléchir à la façon de schématiser cela dans Sage. Si cela vous gêne, c'est probablement parce que Votre cerveau grandit, alors continuez comme ça.

Effectivement, Sage décrit l'intervalle le plus à gauche par  $x \leq -3$  ; l'intervalle le plus à droite par  $x \geq 3$ , et l'intervalle le plus à l'intérieur par deux intervalles,  $x \geq -1$  et  $x \leq 1$ . C'est d'ailleurs ainsi qu'il faut le lire : «  $x \geq -1$  et  $x \leq 1$  ». Cela équivaut à l'inégalité  $-1 \leq x \leq 1$ .

Les résultats étant des listes, vous pouvez considérer chaque intervalle par accès entre parenthèses ou par itération. Les méthodes .rhs() et .lhs() séparent les membres gauche et droit d'une inéquation, comme pour une équation.

Erreurs ou surprises lors de la résolution de certaines équations :

certaines donnent des résultats étranges. Prenons l'exemple de cette équation générique du cinquième degré :

```
sage : résoudre(a*x**5 + b*x**4 + c*x**3 + d*x**2 + e*x + f, x) [0 == a*x^5 + b*x^4 +
c*x^3 + d*x^2 + e*x + f]
```

Sage semble renvoyer la même équation que celle que vous lui avez demandé de résoudre. Si cela vous rappelle l'exemple de la page 41 où Sage a « refusé » de calculer une intégrale indéfinie, félicitations ! Il s'agit plus ou moins du même phénomène : Sage ne trouve pas le moyen d'exprimer la solution par une expression algébrique sur les coefficients, sauf en vous renvoyant l'équation elle-même. Il est bien connu que nous ne pouvons pas résoudre les équations polynomiales génériques de degré 5 ou supérieur d'une manière aussi « simple » que la formule quadratique. Cela touche à un sujet appelé la solvabilité par radicaux.

N'oubliez pas que, dans Sage, une équation utilise deux signes égal. Si vous oubliez d'utiliser deux signes égal lors de l'utilisation de la commande solve() , des problèmes peuvent survenir. Vous êtes prévenu.

```
sage : résoudre(2*x + 1 = 3*x - 2, x)
SyntaxError : le mot-clé ne peut pas être une expression
```

### Solutions approximatives d'une équation

Nous avons vu il y a un instant que Sage ne peut pas décrire les racines du polynôme générique du cinquième degré en termes exacts. Cela reste également vrai pour de nombreux polynômes spécifiques du cinquième degré.

```
sage : résoudre(x**5 - 6*x + 4, x) [0 == x^5
- 6*x + 4]
```

Il nous faut donc parfois nous contenter de solutions approximatives. Il est également possible d'opter pour une solution approximative lorsque la solution exacte est tout simplement trop complexe à mettre en œuvre.

```
sage : résoudre(x**3 + x + 1, x)[0].rhs()
-1/2*(1/18*sqrt(31)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1) + 1/6*(-I*sqrt(3) + 1)/
(1/18*sqrt(31)*sqrt(3) - 1/2)^(1/3)
```

Aie.

Pour trouver une solution approximative à une équation, nous utilisons quelque chose de différent de la commande solve() .

trouver_racine(eq,a,b)	trouver une solution de l'équation sur l'intervalle (a, b)
------------------------	--

Pour utiliser cette fonction, nous avons besoin d'une idée de l'emplacement de la racine, afin de pouvoir spécifier  $a$  et  $b$ . Bien que le terme « racine » implique que l'équation doit avoir 0 d'un côté, ce n'est pas strictement nécessaire ; nous pouvons fournir une équation avec des expressions non nulles des deux côtés, et Sage la résoudra tout de même.

```
sage : trouver_racine(x**5 - 6*x == -4, -5, 5)
-1.7000399860584985
```

Notez qu'une seule racine est renvoyée, même si plusieurs racines existent sur le même intervalle. Lorsque nous savons qu'il existe plusieurs racines, nous pouvons modifier l'intervalle en conséquence.

```
sage : trouver_racine(x**5 - 6*x == -4, 0, 5) 1,3102349335013999
sage : trouver_racine(x**5 - 6*x == -4, 0, 1,3) 0,693378264514721
```

Si nous devons spécifier un intervalle où l'équation n'a pas de racines, Sage génère une erreur.

```
sage : trouver_racine(x**5 - 6*x == -4, 1,3, 5)
RuntimeError : f semble ne pas avoir de zéro sur l'intervalle
```

Une autre approche à essayer lorsque nous ne sommes pas sûrs de l'intervalle à utiliser est une deuxième forme de la méthode `.roots()`, qui nous permet de spécifier un anneau dans lequel rechercher des solutions. En particulier, nous pouvons spécifier l'anneau à

rechercher. <code>f.roots(ring=R)</code> trouve les racines de $f$ dans l'anneau $R$ , ainsi que leurs multiplicités. L'anneau
--

par défaut est l'anneau des rationnels, mais nous pouvons demander à Sage de résoudre des approximations de racines en dehors des rationnels en spécifiant les anneaux réels ou complexes ; en particulier, `RR` et `CC`.

```
sage : f = x**5 - 6*x + 4 sage : f.roots()
RuntimeError : aucune racine explicite trouvée sage :
f.roots(ring=RR) [(-1,70003998605850,
1), (0,693378264514721, 1), (1,31023493350140, 1)] sage : f.roots(ring=CC) [(-1,70003998605850, 1),
(0,693378264514721, 1),
(1,31023493350140, 1),
(-0,151786605978811 -
1,60213970994664*I, 1),
(-0,151786605978811 + 1,60213970994664*I, 1)]
```

L'approche `.roots()` n'est pas toujours efficace, même lorsque `find_root()` l'est. À moins que nous ne soyons particulièrement attachés à la recherche de multiplicités, `find_root()` est la méthode de choix.

```
sage : f = sin(x) + x - 1 sage : f.roots()

RuntimeError : aucune racine explicite trouvée sage :
f.roots(ring=RR)

TypeError : Impossible d'évaluer l'expression symbolique en valeur numérique. sage : find_root(f, 0, 1)
0,510973429388569
```

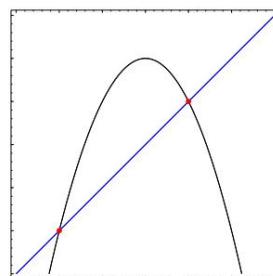
## Systèmes d'équations

De nombreux problèmes concrets impliquent plusieurs variables et plusieurs relations entre elles. Ces problèmes donnent lieu à des équations à plusieurs variables ; lorsqu'on tente d'en résoudre plusieurs à la fois, on parle de système d'équations. La fonction `solve()` permet de résoudre un système d'équations dans Sage ; il suffit de fournir le système sous forme de liste ou de tuple, suivi d'une liste ou d'un tuple des indéterminées des polynômes. La solution d'un système d'équations est similaire à celle d'une inéquation : Sage renvoie une liste de listes. Chaque liste interne correspond à une solution distincte de l'équation ; elle contient des équations indiquant la solution de chaque variable du système.

```
sage : résoudre((x**2 + y == 1, x - y == 1), (x,y)) [[y == -3, x == -2], [y == 0, x
== 1]]
```

Cette équation a deux solutions, correspondant aux points  $(-2, -3)$  et  $(1, 0)$ . On peut illustrer graphiquement la relation géométrique entre les courbes et ces points :

```
sage : p = implicit_plot(x**2 + y == 1, (x, -3, 3), (y, -4, 2), \
couleur='noir')
sage : p = p + implicit_plot(x - y == 1, (x, -3, 3), (y, -4, 2), \
couleur='bleu')
sage : p = p + point((1,0), couleur='rouge', pointsize=60, zorder=5) sage : p = p + point((-2,-3),
couleur='rouge', pointsize=60, zorder=5)
sage : p
```



## Matrices

Si le degré de chaque équation est au plus égal à 1, on parle d'équations linéaires. Un système d'équations linéaires est donc constitué d'un ensemble d'équations que l'on tente de résoudre simultanément. Il s'agit simplement d'un cas particulier de systèmes d'équations.

Les matrices sont intimement liées aux systèmes d'équations linéaires. Tout système d'équations linéaires

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m$$

correspond à une « équation matricielle »

$$Ax = b$$

où

$$A = \begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{matrix}, \quad x = \begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{matrix}, \quad \text{et } b = \begin{matrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{matrix}.$$

Les matrices  $x$  et  $b$  sont des matrices spéciales dans la mesure où elles n'ont qu'une seule colonne ; nous appelons de telles matrices vecteurs.

Nous supposons que vous connaissez les règles de l'arithmétique matricielle ; nous ne les aborderons donc pas ici. Cependant, si vous êtes encore en phase d'apprentissage, nous vous suggérons de réfléchir à la manière dont les règles de multiplication matricielle transforment  $Ax = b$  en le système d'équations ci-dessus. L'analyse matricielle est essentielle à la compréhension des systèmes d'équations linéaires, et comme de nombreuses approches des équations non linéaires impliquent d'abord leur transformation en un système d'équations linéaires, les matrices occupent une place fondamentale en mathématiques.

Créer des matrices, accéder aux éléments et modifier les propriétés fondamentales. Vous pouvez créer une matrice dans Sage à l'aide de la commande `matrix()`. Il existe plusieurs méthodes pour cela ; nous nous concentrerons sur trois d'entre elles.

<code>matrice(liste_lignes)</code>	crée une matrice à partir de <code>list_of_rows</code>
<code>matrice(anneau,liste_lignes)</code>	crée une matrice avec des entrées de l'anneau en utilisant <code>list_of_rows</code>

La raison d'être de ces deux formes est que Sage prend en compte l'anneau d'une matrice pour déterminer comment effectuer la plupart de ses calculs ; il doit donc le connaître. Utilisez la première forme si vous souhaitez laisser Sage deviner l'anneau. Vous pourriez constater que le choix de Sage ne vous convient pas ; dans ce cas, vous pouvez corriger la situation en douceur grâce à la méthode pratique `.change_ring()` :

<code>M.change_ring(anneau)</code>	convertit les entrées de <code>M</code> pour qu'elles résident dans l'anneau et renvoie le résultat
<code>M.base_ring()</code>	rapporte que l'anneau Sage pense que les éléments de <code>M</code> résident dans

Gardez à l'esprit que Sage ne modifiera pas la matrice elle-même ; il produit une nouvelle matrice et la renvoie. La matrice d'origine reste dans l'ancien anneau. Nous utilisons cette procédure un peu plus loin.

Nous allons d'abord illustrer la création d'une matrice. Pour spécifier la matrice,

$$M = \begin{matrix} & 1 & 1 & 0 \\ & 1 & & \\ & & 1 & \end{matrix}$$

nous allons lister chacune de ses lignes sous forme de liste à l'intérieur d'une autre liste :

```
sage : M = matrice([\ [1, 1],\ [0, 1]\ ])\ sage : M\n[1 1]\n[0 1]
```

Il n'est pas nécessaire de placer chaque ligne sur sa propre ligne, mais nous pensons que cela facilite la lecture. Vous pouvez placer la matrice entière sur une seule ligne si vous le souhaitez.

Il est bien sûr possible d'utiliser des compréhensions dans la définition d'une matrice :

```
sage : matrice([[i, i+1, i+2] pour i dans la plage(3)])\n[0 1 2]\n[1 2 3]\n[2 3 4]
```

Trois matrices spéciales que vous pouvez créer sont une matrice identité et une matrice diagonale.

matrice_identité(n)	renvoie la matrice identité de dimension n
matrice_diagonale(entrées)	renvoie une matrice avec 0 partout sauf sur la diagonale principale, et les valeurs des entrées sur la diagonale
matrice_zéro(m,n)	principale renvoie une matrice m × n de zéros

Vous pouvez également utiliser une compréhension de liste pour créer des listes d'entrées pour des matrices. Nous le démontrons pour une matrice diagonale :

```
sage : matrice_diagonale([i**2 pour i dans la plage(3)])\n[0 0 0]\n[0 1 0]\n[0 0 4]
```

Vous accédez aux éléments d'une matrice en utilisant l'opérateur crochet ; pour accéder à l'élément  $M_{i,j}$ , utilisez  $M[i,j]$ . Ici, « accès » ne signifie pas simplement « lecture » ; cela signifie également « écriture », ce qui permet de modifier facilement les entrées d'une matrice. N'oubliez pas que, comme pour les listes, la première ligne commence à la position 0 et non à la position 1 ; il faut donc en tenir compte.

```
sage : M[0,1] = 3
sage : M
[1 3]
[0 1]
```

Cette possibilité de modifier les éléments d'une matrice signifie que Sage considère une matrice comme modifiable, tout comme une liste. Il peut arriver que vous souhaitiez travailler avec une matrice immuable. Par exemple, vous ne pouvez stocker que des objets immuables dans un ensemble ; si vous souhaitez un ensemble de matrices, vous devez indiquer à Sage que vous n'avez pas l'intention de modifier les matrices concernées. Pour ce faire, utilisez la méthode `.set_immutable()`.

<code>M.set_immutable()</code>	rend un objet immuable Vrai si et
<code>M.isMutable()</code>	seulement si l'objet est mutable Vrai si et
<code>M.est_immutable()</code>	seulement si l'objet est immuable La commande ne

fonctionne que dans un sens ; pour rendre une matrice à nouveau mutable, faites-en une copie avec la commande `copy()` . `copy(obj)`

	renvoie une copie de l'objet obj
--	----------------------------------

```
sage : { M }
TypeError : les matrices mutables ne sont pas hachables
sage : M.set_immutable() sage :
M.isMutable()
FAUX
sage : { M } {[1
3] [0 1]}
sage :
M = copy(M) sage :
M.isMutable()
Vrai
```

Les matrices ne sont pas les seuls objets que Sage peut copier ou rendre immuables, ces commandes ont donc une applicabilité plus large.

Calcul et manipulation de matrices. Vous pouvez effectuer des calculs matriciels dans Sage en utilisant les symboles mathématiques habituels.

```
sage : N = matrice([ [1, x],
\ [0, 1]])
sage : M *
N [ 1 x + 3] [ 1]

0
```

Sage propose également un très grand nombre de méthodes permettant d'envoyer une matrice. Nous n'en listons que quelques-unes dans le tableau 1. Pour les consulter toutes, n'oubliez pas que vous pouvez toujours découvrir les méthodes qu'un objet comprend en saisissant son identifiant, suivi du point, puis en appuyant sur la touche Tab . Quant à

M.add_multiple_of_row(j,i,c)	ajoute c fois chaque entrée de la ligne i à l'entrée correspondante de la ligne j
M.caractéristique_polynomiale()	renvoie le polynôme caractéristique de M
M.déterminant()	renvoie le déterminant de M
M.dimensions()	renvoie un tuple (m, n) où m est le nombre de lignes et n le nombre de colonnes
M.echelon_form()	renvoie la forme échelonnée de M tout en laissant la matrice elle-même dans sa forme originale
M.echelonize()	transforme M en forme échelonnée ; renvoie Aucun
M.valeurs_propres()	renvoie les valeurs propres de M
M.eigenvectors_right()	renvoie les vecteurs propres de M
M.inverse()	renvoie l'inverse de M
M.kernel()	renvoie le noyau de M (pour accéder à la base les vecteurs d'un noyau K utilisent K.basis())
M.ncols()	renvoie le nombre de colonnes dans M
M.nrows()	renvoie le nombre de lignes dans M
M.nullité()	renvoie la dimension du noyau
M.rank()	renvoie le rang de M
M.set_row_to_multiple_of_row(j,i,c)	définit chaque élément de la ligne j au produit de c et l'entrée correspondante de la ligne i
M.swap_rows(i,j)	échange les lignes i et j de M
M.sous-matrice(i,j,k,)	renvoie la sous-matrice k× de M dont le sommet le coin gauche est dans la rangée i, colonne j de M
M.transpose()	renvoie la transposée de M

TABLEAU 1. méthodes comprises par une matrice Sage

les méthodes que nous listons ici, ne vous inquiétez pas si vous ne comprenez pas le but de chacune d'elles, mais chacune certains d'entre eux devraient à un moment donné s'avérer utiles dans les études de premier cycle.

Voyons comment certaines de ces commandes pourraient fonctionner et soulignons certaines des erreurs qui peuvent survenir. Par exemple, supposons que nous souhaitions transformer une matrice en forme triangulaire supérieure, et aussi en observant certains calculs qui se produisent en cours de route. nous effectuerons ce calcul étape par étape plutôt que de rechercher une commande Sage particulière qui fait tout en même temps. Notre matrice d'exemple sera

$$A = \begin{matrix} & 1 & 2 & 3 & 4 \\ & 0 & 2 & 2 & 3 \\ & 8 & 3 & 1 & 2 \\ & 0 & 1 & 2 & 3 \end{matrix}$$

Nous vous laissons le soin de créer une matrice pour A dans Sage. Pour la transformer en forme triangulaire supérieure, nous observons que A0,0 est déjà 1, tandis que A1,0 et A3,0 sont tous deux 0, nous n'avons donc rien à faire avec eux.

A2,0 = 8 = 0, nous voulons ajouter un multiple de la ligne 1 qui élimine le 8 ; c'est assez simple :

```
sage : A.add_multiple_of_row(2, 0, -8)
[2][[ 1      3 4]
  0      2 3]
  0 -13 -23 -30]
  0      1      2 3]
```

Passons à la deuxième colonne. Nous constatons que A1,1 = 2 = 1, il faut donc diviser la ligne par 2.

```
sage : A.set_row_to_multiple_of_row(1, 1, 1/2)
TypeError : Multiplication d'une ligne par un élément de champ rationnel
ne peut pas être fait sur Integer Ring, utilisez change_ring ou
with_row_set_to_multiple_of_row à la place.
```

## PANIQUE!

... eh bien, non, ne le faites pas. Bien sûr, nous avons reçu une erreur, mais celle-ci est très utile. Elle rend le tout parfaitement. Il est clair que le problème est que Sage voit A comme situé au-dessus de l'anneau. Les entiers ne peuvent pas être des fractions, et Si on multiplie la deuxième ligne par 1/2, on obtient 3/2 à la dernière position. Il faut que A soit au-dessus du rationnel. champ, et non sur l'anneau entier. Très bien ; nous avons déjà expliqué comment modifier l'anneau de une matrice, alors faisons-le. Il n'est pas nécessaire de conserver l'ancienne matrice A, nous allons donc lui attribuer le résultat de .change\_ring() en A. Rappelons que le symbole de Sage pour l'anneau des nombres rationnels est QQ.

```
sage : A = A.change_ring(QQ)
sage : A.set_row_to_multiple_of_row(1, 1, 1/2)
sage : A
[[ 2      3 4]
  0      1      1 3/2]
  0 -13 -23 -30]
  0      1      2 3]
```

Excellent ! Nous pouvons maintenant nettoyer A2,1 et A3,1.

```
sage : A.add_multiple_of_row(2, 1, 13)
sage : A.add_multiple_of_row(3, 1, -1)
[[ 1      2      4]
  0      1      1 3/2]
  0      0 -10 -21/2]
  0      0      1 3/2]
```

À partir de là, vous devriez être en mesure de terminer le processus par vous-même, en obtenant la matrice.

$$\begin{array}{r} 1 \ 2 \ 3 \ 4 \ 1 \\ 1 \ 1 \quad - \\ 1 \quad \frac{2 \ 2 \ 1}{20} \\ \hline 9 \\ \hline 20 \end{array}$$

(Les entrées vides représentent 0.)

Nous nous transformons. Une application des matrices implique l'utilisation de la transformation. Matrices en infographie. Nous abordons trois types de matrices de transformation :

- Une matrice d'échelle a la forme

$$\sigma = \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix}$$

Elle a pour effet de redimensionner un point  $(x, y)$  au point  $(sx, sy)$ . •  
Une matrice de rotation a la forme

$$p = \begin{pmatrix} \cos\alpha & -\sin\alpha & \sin\alpha \\ \sin\alpha & \cos\alpha & 0 \end{pmatrix}$$

Il a pour effet de faire tourner un point  $(x, y)$  autour de l'origine d'un angle  $\alpha$ .  
• Une matrice de réflexion a la forme

$$= \begin{pmatrix} \cos\beta & \sin\beta & \sin\beta \\ -\sin\beta & \cos\beta & 0 \end{pmatrix}$$

Il a pour effet de refléter un point  $(x, y)$  sur la droite passant par l'origine dont la pente est  $1-\cos\beta/\sin\beta$ .

Nous illustrerons chacun d'entre eux, en présentant également l' objet vectoriel dans Sage.

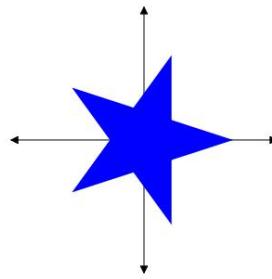
Nous avons déjà évoqué les vecteurs ; il s'agit essentiellement d'une matrice à une seule colonne. Un vecteur est similaire à un tuple ou une liste Sage, car il contient une suite ordonnée de nombres. Il diffère d'un tuple Sage par le fait qu'on lui associe certaines opérations mathématiques. Ce qui nous intéresse ici, c'est qu'il est possible de multiplier une matrice  $m \times n$  par un vecteur  $n \times 1$  ; le résultat est un vecteur  $m \times 1$ . On utilise généralement des matrices carrées, et le résultat de la multiplication d'une matrice  $n \times n$  par un vecteur  $n \times 1$  est un autre vecteur  $n \times 1$ , ce qui est pratique pour des applications répétées.

Sage permet d'utiliser des vecteurs comme points dans les commandes de traçage, telles que `point()`, `line()` et `polygon()`. Cela facilite l'illustration du fonctionnement des matrices de transformation.

Pour ce faire, nous travaillerons sur un polygone défini par les points suivants, dans cet ordre :

$$(1, 0), \quad \text{parce que } \frac{4\pi}{5}, \frac{4\pi \sin}{5}, \quad , \quad \text{parce que } \frac{8\pi}{5}, \frac{8\pi \sin}{5}, \quad , \quad \text{parce que } \frac{2p}{5}, \sin \frac{2\pi}{5}, \quad , \quad \text{parce que } \frac{6p}{5}, \sin \frac{6\pi}{5} \quad .$$

Définissez une liste  $V$  dont les éléments sont les vecteurs de ces valeurs. Tracez le polygone défini par ces points et vous obtiendrez une étoile à cinq branches presque parfaite :



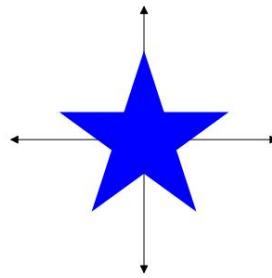
(Si vous obtenez un pentagone au lieu d'une étoile, assurez-vous que les points sont dans l'ordre ci-dessus. Sinon, .pop() et .insert() jusqu'à ce qu'ils le soient.) Nous disons « presque sympa » parce que c'est un peu décalé. Ne serait-il pas agréable de l'avoir pointé vers le haut ?

Nous pouvons résoudre ce problème. Nous avons la technologie. Comment procéder ? Nous pourrions effectuer une rotation de  $1/20$  d'une rotation complète, de  $2\pi/20$  ou de  $\pi/10$ . Nous avons affirmé précédemment que la matrice de rotation permettrait d'atteindre cet objectif, si seulement nous posions  $\alpha = \pi/10$ . Essayons. Nous allons créer une matrice de rotation correspondante  $M$ , puis l'utiliser pour créer une nouvelle liste,  $U$ , dont les points sont les rotations des points de  $V$  de  $\pi/10$ . Créer  $U$  est facile avec une liste en compréhension :

```
sage: M = matrix([[cos(pi/10), -sin(pi/10)], \
[sin(pi/10), cos(pi/10)]]) sage : U
= [M*v pour v dans V]
```

Tracez maintenant le polygone formé par  $U$  pour vérifier que nous avons bien redressé notre étoile :<sup>4</sup>

```
sage : polygone(U)
```



Et la réflexion ? Si on regarde l'étoile d'origine, on pourrait la refléter autour de l'axe des Y, qui a une pente...

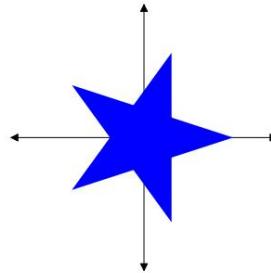
Hmm. Eh bien, c'est embêtant. L'axe des Y a une pente indéfinie. On obtient généralement une pente indéfinie en divisant par 0. Le dénominateur de la pente de la matrice de réflexion est  $\sin\beta$  ; donc si  $\sin\beta = 0$ , nous devrions avoir une réflexion par rapport à l'axe des Y. Cela fonctionne pour  $\beta = 0$  et  $\beta = \pi$  ; nous allons essayer la première solution en espérant que cela fonctionne.

---

<sup>3</sup> Si vous êtes de ces cinglés qui pensent que cette étoile est parfaite comme elle est et qu'il serait préférable de faire pivoter les axes, eh bien ! Vous avez un avenir en mathématiques. Si vous n'êtes pas de ces cinglés et que vous pensez qu'il est préférable de faire pivoter l'étoile... oui, vous avez peut-être aussi un avenir en mathématiques. Mais on vous surveille de près, juste pour être sûrs que vous ne dérapiez pas. Un peu comme cette étoile-là.

<sup>4</sup>Ou plutôt, si l'on en croit la note précédente, nous avons fait du tort à notre polygone.

```
sage : N = matrice([[cos(0), sin(0)], [sin(0), -cos(0)]])  
  
sage : W = [N*v pour v dans V]  
sage : polygone(W)
```



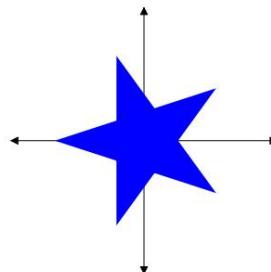
Eh bien, c'est embêtant. Ça n'a pas du tout tourné !

En fait, il s'est inversé ; on ne le voit simplement pas. Tout d'abord, la pente devient en fait  $1-\cos 0/\sin 0 = 1-1/0 = 0/0$  ; pour une pente verticale, il faut une valeur non nulle divisée par zéro. Un coup d'œil à N le confirme :

```
sage: N  
[ 1 0]  
[ 0 -1]
```

Si l'on considère le fonctionnement de la multiplication matricielle, l'effet de N sur tout vecteur est de préserver la valeur de x et d'inverser la valeur de y. Nous obtenons donc une inversion verticale plutôt qu'horizontale. Il semble que nous souhaitions utiliser  $\beta = \pi$ .

```
sage : N = matrice([[cos(pi), sin(pi)], [sin(pi), -cos(pi)]])  
sage : W = [N*v pour v dans  
V] sage : polygone(W)
```



Super!

Pour une matrice de réflexion, les points le long de l'axe de réflexion restent fixes : si l'on appliquait la valeur N corrigée au vecteur  $(0, 2)$ , on obtiendrait le vecteur  $(0, 2)$ . Il s'agit d'un cas particulier de vecteur propre, un vecteur qui reste sur la même ligne après multiplication par une matrice. Chaque matrice possède ses propres vecteurs propres, et nous pouvons les « découvrir » grâce à la méthode `.eigenvalues_right()`. Dans ce cas :

```
sage : N.vecteurs_propres_droite() [(-1,
[(1, 0)], 1), (1, [(0, 1)], 1)]
```

C'est beaucoup d'informations, alors voyons ce que cela nous dit.

La liste contient un tuple par vecteur propre. Chaque tuple contient trois éléments :

- Premièrement, cela nous donne ce qu'on appelle une valeur propre. Les valeurs propres vous indiquent comment la taille du vecteur change ; il restera sur la même droite passant par l'origine, mais la taille changera. La relation de base entre une matrice  $M$ , un vecteur propre  $e$  et la valeur propre correspondante  $\lambda$  est que  $Me = \lambda e$ . Vous pouvez également extraire les valeurs propres d'une matrice à l'aide de la méthode `.eigenvalues()`.
- Deuxièmement, cela donne une liste de vecteurs propres qui correspondent à cette valeur propre. Dans la plupart des cas, vous vous attendriez à ne voir qu'un seul vecteur propre par valeur propre, mais il peut arriver que vous en obteniez plusieurs.
- Enfin, cela donne la multiplicité de la valeur propre. Cela se rapporte à l'idée suivante : les valeurs propres sont les racines  $\lambda_1, \lambda_2, \dots, \lambda_m$  d'un polynôme appelé polynôme minimal de la matrice. Ce polynôme se factorise linéairement comme  $(x - \lambda_1)(x - \lambda_2) \dots$  Chaque puissance d'un facteur linéaire distinct est la multiplicité de la valeur propre correspondante. Ce point ne nous intéresse pas dans le cadre de l'application actuelle.

On espère généralement que les vecteurs propres  $e_1, \dots, e_m$  soient linéairement indépendants ; c'est la seule façon d'écrire

$$a_1 e_1 + \dots + a_m e_m = 0 \text{ est si}$$

$a_1 = \dots = a_m = 0$ . Nous pouvons le voir dans les vecteurs propres de  $N$ , puisque

$$\begin{matrix} & & & 0 \\ a_1 & 1 & 0 & + a_2 & 1 & = 0 \\ & & & & & \end{matrix} \quad \begin{matrix} a_1 \\ a_2 \end{matrix} = 0 \quad a_1 = 0, a_2 = 0.$$

Malheureusement, toutes les matrices de dimension  $n$  n'ont pas  $n$  valeurs propres distinctes ; si vous regardez la matrice d'échelle

$$S = \begin{matrix} 3 & 0 \\ 0 & 3 \end{matrix},$$

vous constaterez qu'il n'a qu'une seule valeur propre de multiplicité 2, car il traite chaque point exactement de la même manière.

Comme nous l'avons mentionné, chaque vecteur propre reste sur la même droite passant par l'origine lorsque nous lui appliquons la matrice. En termes de points, cela signifie que les vecteurs propres définissent une ligne de points qui peuvent se déplacer sur cette ligne, soit en inversant leur direction, soit en changeant de taille, mais les points restent sur la même droite passant par l'origine. Dans le cas de la matrice de réflexion  $N$ , les vecteurs propres sont  $(1, 0)^T$ , qui se trouve sur l'axe des  $x$ , et  $(0, 1)^T$ , qui se trouve sur l'axe des  $y$ . La valeur propre de  $(1, 0)^T$  est  $-1$  ; cela signifie que  $(1, 0)$  se déplace vers  $(-1, 0)$ , qui se trouve toujours sur l'axe des  $x$ . La valeur propre de  $(0, 1)^T$  est  $1$  ; cela signifie que  $(0, 1)$  se déplace vers  $(0, 1)$ , qui se trouve toujours sur l'axe des  $y$ . Ce n'est pas le cas pour les points situés sur d'autres droites passant par l'origine ; le point  $(3, 5)$ , par exemple, se déplace vers le point  $(-3, 5)$ , qui se trouve sur une droite complètement différente passant par l'origine. Bien sûr, nous nous attendons à cela puisque  $N$  est une matrice de réflexion.

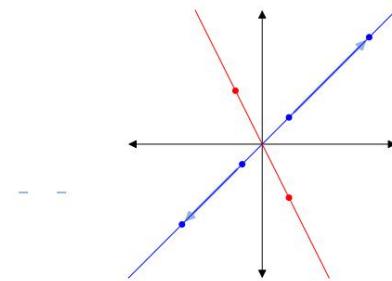
Pour conclure cette enquête, considérons une matrice différente,

$$M = \begin{matrix} 3 & 1 \\ 2 & 2 \end{matrix}.$$

Utilisez Sage pour vérifier que ses vecteurs propres sont

$$\begin{pmatrix} 1 & 1 \\ 1 & -2 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 1 & -2 \\ -2 & 1 \end{pmatrix},$$

avec les valeurs propres correspondantes 4 et 1. Cela signifie que nous nous attendons à ce que les points sur la même ligne que  $\begin{pmatrix} 1 & 1 \\ 1 & -2 \end{pmatrix}$  restent sur cette ligne, mais soient redimensionnés par un facteur de 4, tandis que les points sur la même ligne que  $\begin{pmatrix} 1 & -2 \\ -2 & 1 \end{pmatrix}$  restera au même endroit :



Comme vous pouvez l'imaginer, les choses deviennent un peu étranges avec une matrice de rotation, car elle déplace chaque point du plan, sauf l'origine, vers une droite différente passant par l'origine. Dans ce cas, vous constaterez que ses vecteurs propres ont des valeurs complexes. Essayez !

### Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Sage dispose de fonctionnalités permettant de trouver des solutions exactes et approximatives aux équations.
2. Sage peut trouver la solution exacte à n'importe quelle équation, donc la principale raison d'approximer les solutions c'est qu'ils sont plus faciles à regarder.
3. La commande `solve()` trouve des solutions exactes et approximatives à une équation.
4. La méthode `.roots()` n'est pas aussi efficace que la commande `find_root()`, même lorsque nous spécifions un anneau d'approximations de nombres, comme `RR` ou `CC`.
5. La multiplicité d'une racine n'est pas liée à la géométrie de la courbe.
6. Les résultats de la commande `solve()` sont dans un format similaire pour les inégalités et les systèmes de équations.
7. Sage décide parfois comment résoudre un problème en fonction de l'anneau dans lequel se trouvent ses valeurs.
8. Vous devez toujours spécifier l'anneau d'une matrice lors de sa création.
9. Vous pouvez extraire les valeurs propres des résultats de `eigenvectors_right()` et `eigenvectors_left()` commandes.
10. Les valeurs propres n'ont qu'une importance théorique.

Choix multiple.

1. Laquelle des commandes et méthodes suivantes ne produit pas de solutions exactes par défaut ?
  - A. `résoudre()`
  - B. `.racines()`
  - C. `.valeurs propres()`
  - D. `find_root()`
2. Le résultat de `solve()` lorsqu'on lui donne une équation dans une variable est : A. la valeur d'une solution approximative B. une liste de listes de solutions et de multiplicités C. une liste d'équations linéaires qui décrivent les solutions

D. une liste de listes d'équations linéaires, chacune décrivant une solution 3. Le résultat de solve()  
lorsqu'on lui donne plusieurs équations dans plus d'une variable est :

- A. la valeur d'une solution approximative B. une liste de listes de solutions et de multiplicités C. une liste d'équations linéaires qui décrivent les solutions D. une liste de listes d'équations linéaires, chacune décrivant une solution 4. Le résultat de find\_root() lorsqu'on lui donne une équation dans une variable est : A. la valeur de la solution approximative B. une liste de listes de solutions et de multiplicités C. une liste d'équations linéaires qui décrivent les solutions D. une liste de listes d'équations linéaires, chacune décrivant une solution 5. Le résultat de .roots() lorsqu'il est appliqué à une équation dans une variable est : A. la valeur de la solution approximative B. une liste de listes de solutions et de multiplicités C. une liste d'équations linéaires qui décrivent les solutions D. une liste de listes d'équations linéaires, chacune décrivant une solution

6. Laquelle des commandes et méthodes suivantes nécessite que vous spécifiez les points de terminaison d'un intervalle qui contient une solution ?

- A. résoudre()
- B. find\_root()
- C. .roots()
- D. aucune des réponses ci-dessus

7. Laquelle des méthodes suivantes pour une matrice renvoie ses valeurs propres ?

- A. valeurs propres()
- B. vecteurs propres\_droite()
- C. tout ce qui précède D.
- aucun de ce qui précède 8.

Les vecteurs v1 , v2 , ..., vn sont linéairement indépendants lorsque :

- A. Nous pouvons trouver une matrice M dont les vecteurs propres sont v1 ,v2 ,...,vn .
- B. L'ensemble {v1 ,v2 ,...,vn } définit un espace vectoriel.
- C. La seule façon pour que  $a_1 v_1 + a_2 v_2 + \dots + a_n v_n = 0$  est si  $a_1 = a_2 = \dots = a_n = 0$ .
- D. Les vecteurs déclarent leur indépendance par rapport à un espace vectoriel et créent leur propre sous-espace.

9. Pourquoi devrions-nous nous attendre à ce qu'une matrice de réflexion ait deux vecteurs propres linéairement indépendants ?

- A. Chaque matrice bidimensionnelle possède deux vecteurs propres linéairement indépendants.
- B. Une réflexion déplace toujours un point vers un deuxième point, et c'est ainsi que nous comptons le nombre de vecteurs propres linéairement indépendants.
- C. Une réflexion est comme un miroir, nous devrions donc nous attendre à ce que le deuxième vecteur propre reflète le d'abord.
- D. Les points de deux droites passant par l'origine restent sur ces droites : l'axe de symétrie, et la ligne perpendiculaire à celle-ci.

10. Pourquoi s'attend-on à ce que les valeurs propres d'une matrice de rotation aient des valeurs complexes ?

- A. Une matrice de rotation déplace des points sur le plan réel vers des points sur le plan complexe, et vice versa.
- B. Le seul point du plan réel qui reste sur la même ligne passant par l'origine est le l'origine elle-même.
- C. Les valeurs propres sont toujours complexes ; nous ne le remarquons simplement pas lorsque leurs valeurs sont réelles.

D. Les problèmes complexes nécessitent des solutions complexes.

Programmation.

1. Utilisez `solve()` pour trouver les solutions des équations polynomiales génériques de degré trois et quatre. Combien y en a-t-il ? (Ne comptez pas à la main ! Utilisez Sage pour compter les solutions !)
2. La commande `solve()` ne renvoie pas les solutions d'une équation unique à une variable selon l'ordre de la norme complexe. Écrivez une procédure qui accepte une équation comme argument, appelle `solve()` pour la résoudre, puis trie les solutions selon la norme. Conseil : Vous pouvez utiliser ou adapter la procédure clé `by_norm()` définie à la page 113.
4. Écrivez une procédure qui accepte une inégalité comme argument, la résout, puis produit un graphique de ses solutions sur la droite numérique, semblable à celle de la p. 137. 5. (a)

Écrivez le pseudo-code d'une procédure qui accepte deux fonctions  $f$  et  $g$ , résout leurs intersections, puis calcule l'aire totale entre  $f$  et  $g$ . (Nous trouvons l'aire totale en additionnant la valeur absolue des intégrales pour chaque intervalle défini par les intersections de  $f$  et  $g$ .)

(b) Implémentez votre pseudo-code en tant que procédure Sage qui utilise la commande `roots()` pour trouver des solutions exactes dans (c).

Écrivez une procédure interactive qui permet à l'utilisateur de spécifier les fonctions  $f$  et  $g$ , ainsi qu'une couleur. Il appelle ensuite le code Sage que vous avez écrit dans la partie (b) pour déterminer l'aire totale entre  $f$  et  $g$ , représente  $f$  et  $g$  en noir avec une épaisseur de 2, remplit l'aire entre eux avec la couleur spécifiée par l'utilisateur, et enfin écrit l'aire au-dessus des courbes.

6. Nous avons utilisé la méthode `.roots()` pour trouver les racines et les multiplicités de

$$f(x) = (x - 1)(x + 2)^2(x - 4)^3.$$

Créez un graphe de  $f$  sur un intervalle incluant les trois racines, mais suffisamment petit pour ne pas perdre les détails de la courbe de  $f$  autour d'elles. Tracez ce tracé en noir, d'épaisseur 1. Ajoutez-y trois autres tracés de  $f$ , dont les valeurs minimales et maximales de  $x$  sont voisines d'une racine différente. Tracez ces tracés en rouge, d'épaisseur 3. Maintenant que vous avez ce tracé, décrivez une similarité géométrique entre une racine de multiplicité  $m$  et le graphe de  $x^m$ .

7. Bien que Sage dispose d'une commande `implicit_plot()`, il lui manque une commande `implicit_diff()` pour effectuer une différentiation implicite. (Vous pouvez consulter la définition de la différentiation implicite dans votre livre de calcul.) Écrivez une procédure pour ce faire, en suivant les étapes suivantes : (a) Déplacez tout ce qui est dans  $f$  d'un côté de l'équation. Assurez-vous que  $f$  est une fonction de  $x$ .

et  $y$ .

(b) Définissez  $yf$  comme une fonction implicite de  $x$  à l'aide de la commande `yf=function('yf')(x)`. (c) Remplacez  $y$  dans  $f$  par  $yf$ . Indice : si vous avez défini  $f$  comme une fonction de  $x$  et  $y$ , il suffit de redéfinir  $f$  comme une fonction de  $x$  et  $yf$ . (d) Soit  $df$  la dérivée de  $f$ . (e) Résolvez  $df$  pour `diff(yf)`. (f)

Renvoie la solution — non pas

la liste des équations, mais le membre de droite. Pour in-

position, le résultat de `implicit_diff(y==cos(x*y))` devrait être `-sin(x*yf(x))*yf(x)/(x*sin(x*yf(x)) + 1)`.

## CHAPITRE 7

## Prise de décision

Comme mentionné précédemment, l'une des questions fondamentales des mathématiques est la suivante :

comment pouvons-nous trouver la racine 1 d'un polynôme ?

Comme cela est plus ou moins impraticable dans de nombreux cas, nous nous tournons vers la question

connexe : comment pouvons-nous approximer la racine d'un polynôme ?

Il existe plusieurs façons de procéder, mais nous allons décrire une méthode permettant d'« approximer » la racine « exactement ». Ce n'est pas vraiment une contradiction, car notre approximation consiste en un intervalle précis  $(a, b)$  de nombres rationnels dans lequel le polynôme possède au moins une racine. La valeur sera donc une approximation, dans la mesure où nous ne savons pas avec certitude quel  $c \in (a, b)$  est la racine, mais il s'agira d'une approximation exacte, car la solution est exempte de toute erreur.<sup>2</sup>

En bonus, la technique que nous allons regarder à volonté :

- fonctionnent pour toute fonction continue, pas seulement pour les polynômes ; et •  
dans de rares cas,<sup>3</sup> nous donnent la valeur exacte !

Cette technique est la méthode de la bissection, et elle est basée sur un fait important du calcul :

**THÉORÈME DES VALEURS INTERMÉDIAIRES** . Si une fonction  $f$  est continue sur l'intervalle  $[a, b]$ , alors pour toute valeur de  $y$  comprise entre  $f(a)$  et  $f(b)$ , on peut trouver  $c \in (a, b)$  tel que  $f(c)$  soit cette valeur de  $y$ .

Par exemple,  $f(x) = \cos x$  est continue partout, donc elle est certainement continue sur l'intervalle  $[\pi/6, \pi/3]$ . Or,  $f(\pi/6) = 3/2$ , tandis que  $f(\pi/3) = 1/2$ , et  $2/2$  se situe entre  $1/2$  et  $3/2$ , donc il doit y avoir un  $c \in [\pi/6, \pi/3]$  tel que  $f(c) = 2/2$ .

## La méthode de la bissection

Notre exemple avec  $\cos x$  ne semblera pas impressionnant ; après tout, vous saviez déjà que  $f(\pi/4) = 2/2$ .

C'est vrai, mais considérons ce scénario : si

- 
- $f(a)$  est positif et  $-f(b)$   
est négatif alors — 0  
se  
situe entre  $f(a)$  et  $f(b)$ , donc — une  
racine  $c$  doit se situer entre  $a$  et  $b$ .

Par exemple, supposons que nous voulions identifier la racine de  $f(x) =$

$$\cos x - x.$$

---

<sup>1</sup>Au cas où vous l'auriez oublié, une « racine » est une valeur d'une expression qui, une fois substituée dans l'expression, donne 0.

<sup>2</sup>En revanche, les réponses en virgule flottante sont une approximation précise, plutôt qu'une approximation exacte, dans la mesure où la virgule flottante nous donne un nombre qui est légèrement erroné.

<sup>3</sup>Très rares, mais ils existent.

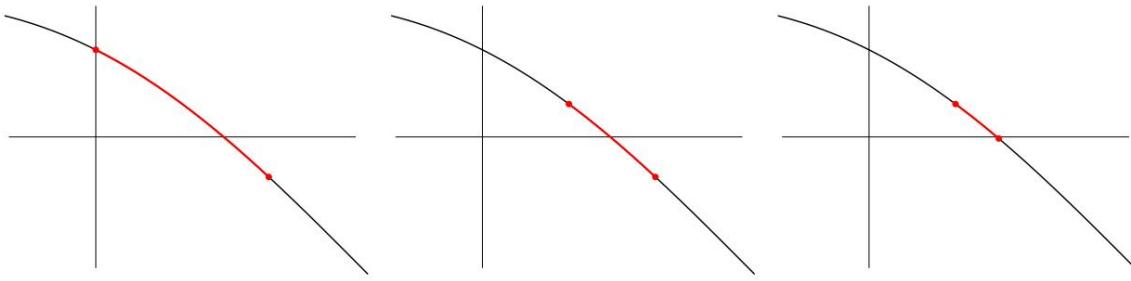


FIGURE 1. La méthode de bisection utilise le théorème de la valeur intermédiaire pour réduire l'intervalle contenant une racine de  $(0, 1)$  jusqu'à  $(1/2, 3/4)$ ... et plus encore !

Si vous le tracez dans Sage, le graphique suggère qu'une racine se situe entre  $x = 0$  et  $x = 1$ . Mais peut-on se fier à ce graphique ? Effectivement ; comme vous l'avez appris en calcul, •

cosinus  $x$  est continu ; •  $x$

est un polynôme, et tous les polynômes sont continus ; et • les sommes, différences et produits de fonctions continues sont continus.

Donc  $f$  est continue, et le théorème des valeurs intermédiaires confirme l'inspection visuelle.

Bien sûr, savoir qu'il existe une racine entre 0 et 1 ne renforce pas vraiment la confiance.

Nous aimerais localiser cette racine plus précisément que cela. Comment ?

Dans la méthode de bisection, nous divisons l'intervalle  $[a, b]$  en deux parties. Dans ce cas, nous le divisons en  $[0, 1/2]$  et  $[1/2, 1]$ . Nous testons ensuite la nouvelle extrémité de la fonction :  $f(1/2) > 0$ . Nous savions déjà que  $f(0) > 0$  ; le théorème des valeurs intermédiaires ne garantit donc pas qu'une racine soit comprise entre 0 et  $1/2$ . En revanche,  $f(1) < 0$  ; puisque 0 est compris entre  $f(1/2)$  et  $f(1)$ , le théorème des valeurs intermédiaires garantit qu'une racine soit comprise entre  $1/2$  et 1. Nous remplaçons donc  $[0, 1]$  par  $[1/2, 1]$ .

Un intervalle de taille  $1/2$  n'est toujours pas très impressionnant, mais il n'y a aucune raison pour que nous ne puissions pas continuer. Divisez à nouveau l'intervalle en deux parties :  $[1/2, 3/4]$  et  $[3/4, 1]$ . Le nouveau point final satisfait  $f(3/4) < 0$  ; la racine doit donc être comprise entre  $1/2$  et  $3/4$ . La figure 1 illustre chacune des trois premières étapes du processus.

On peut persister dans cette voie aussi longtemps qu'on le souhaite. À chaque fois, l'incertitude diminue de moitié, soit répéter cela 20 fois nous donne un intervalle de largeur  $1/2^{20} \approx 1,0 \times 10^{-6}$  difficile à , 0,000001. C'est un peu faire à la main, mais il est simple à répéter pour un ordinateur ; nous avons déjà vu comment faire cela.

Mais comment devrions-nous dire à l'ordinateur de décider quel point final remplacer par le point médian ?

Entrez la structure de contrôle if/else if/else :

```

si condition1
    que faire si la condition1 est vraie
sinon si condition2
    que faire si la condition2 est vraie
...
sinon que faire si aucune des conditions n'est vraie

```

Cela dirige l'exécution en fonction des conditions répertoriées, permettant au calcul de se dérouler d'une manière appropriée à la situation.

Toutes les parties de la structure ne sont pas nécessaires. Seul un if doit apparaître ; après tout, else n'a aucun sens sans if. En revanche, vous pouvez donner du sens à un if sans else ; c'est souvent utile si une valeur particulière nécessite un « massage » supplémentaire avant que l'algorithme puisse continuer.

Dans notre cas, nous voulons que l'algorithme remplace le point final par le même signe que le point médian. Cela implique le pseudo-code suivant :

```

algorithme Méthode_de_Bisection
entrées
    • a, b
    • f , une fonction telle que :
        – f est continue sur [a, b] – f
        (a) et f (b) ont des signes opposés • n, le
            nombre de bissections à effectuer sorties • [c,
d] [a,
b] telles que – d – c = 1/2 n
(b – a), et – une racine de f
se trouve dans [c, d]

faire soit c = a et d = b
répéter n fois
    soit e = 1/2 (c + d)
    si f (c) et f (e) ont le même signe
        remplacer c
        par
        e sinon
    remplacer d par e retourner [c, d]

```

Malheureusement, cela n'est pas encore suffisant pour être implémenté dans Sage, car une question

demeure : comment décider si  $f(c)$  et  $f(e)$  ont le même signe ?

Nous envisageons deux voies.

Une méthode astucieuse à laquelle vous avez peut-être pensé est basée sur l'observation pré-algèbre selon laquelle deux nombres réels ont le même signe si et seulement si leur produit est positif :

$$(-2) \times (-2) > 0 \text{ et } 2 \times 2 > 0 \quad \text{mais} \quad (-2) \times 2 < 0 \text{ et } 2 \times (-2) < 0.$$

Nous pouvons donc remplacer le test

si  $f(c)$  et  $f(e)$  ont le même signe

avec

si  $f(c) f(e) > 0$

Et c'est tout. C'est assez simple à implémenter en code Sage, car Sage propose des mots-clés qui correspondent presque parfaitement à notre pseudo-code : if, elif et else. Comme on peut s'y attendre, les commandes que Sage doit exécuter dans chaque cas doivent apparaître en dessous, en retrait. Notre pseudo-code se traduit donc comme suit.

```
sage : def méthode_de_bisection(a, b, n, f, x=x) :
    c, d = a, bf(x) =
    f pour i
        dans la plage(n) : #
            calculer le point médian, puis comparer e = (c
            + d)/2 si f(c)*f(e) >
        0 :
            c = e
        autre:
            d = e
    retour (c, d)
```

Attention aux deux points ! Si vous oubliez l'un d'eux par inadvertance, Python générera une erreur.

```
sage: ...
    si f(c)*f(e) > 0
        c = e
    ...
SyntaxError : syntaxe non valide
```

Une fois que vous l'avez correctement tapé, vérifions que cela fonctionne.

```
sage : méthode_de_bisection(0, 1, 20, cos(x) - x) (387 493/524 288,
774 987/1 048 576)
```

Mais il existe encore une voie plus excellente.

## logique booléenne

Jusqu'à présent, nous avons utilisé les identifiants Sage Vrai et Faux sans beaucoup de commentaires ; après tout, leur signification est assez évidente, sauf si vous les avez réaffectés. Pourtant, ces deux idées sont à la base de la plupart des calculs pratiques. Un domaine appelé logique booléenne étudie les méthodes logiques possibles à partir des deux concepts fondamentaux Vrai et Faux .

---

<sup>4</sup>Remarquez la distinction entre les identifiants Sage des termes à signification fixe (vrai et faux) et les constantes mentales de la logique booléenne (Vrai et Faux).

Quatre opérations « fondamentales ». Les quatre opérations fondamentales de la logique booléenne sont les opérateurs « ou », « et », « xor ».<sup>5</sup> Elles sont régies par les tables de « vérité » suivantes, qui indiquent le comportement des variables ayant pour valeur « Vrai » ou « Faux » sous les opérations données.

x	pas x
Vrai	Faux
Faux	Vrai

x	yx ou y
Vrai	Vrai
Vrai	Faux
Faux	Vrai
Faux	Faux

x	yx et y
Vrai	Vrai
Vrai	Faux
Faux	Vrai
Faux	Faux

x	yx xor y
Vrai	Faux
Vrai	Vrai
Faux	Vrai
Faux	Faux

Après avoir précisé la signification de ces termes, nous tenons à souligner qu'ils devraient vous être intuitifs ; la seule différence réelle entre leur signification précise et votre compréhension intuitive réside dans les termes « or » et « xor ». Nous ne trouverons généralement pas xor nécessaire dans ce texte, mais il a son utilité.

Opérateurs booléens dans Sage. Ces concepts se traduisent tous immédiatement dans Sage : Vrai, Faux, Non, Ou, Et et Xor.

Grâce à leur aide, nous pouvons désormais réécrire notre code Sage avec la logique booléenne plutôt qu'avec une astuce mathématique. Vérifier que deux nombres réels c et e ont le même signe revient à tester que

$$(f(c) > 0 \text{ et } f(e) > 0) \text{ ou } (f(c) < 0 \text{ et } f(e) < 0)$$

et cela se traduit directement dans le code Sage

$$(f(c) > 0 \text{ et } f(e) > 0) \text{ ou } (f(c) < 0 \text{ et } f(e) < 0)$$

<sup>5</sup> Abrévation de exclusif ou, dont le nom dérive du fait que, en général, vous pouvez soit couper le gâteau, soit L'avoir, mais pas les deux. Formulé différemment, vous pouvez avoir exclusivement l'une ou l'autre option.

En quoi est-ce différent du « or » classique ? La règle ne s'applique pas aux anniversaires, car on peut couper son gâteau, le manger, ou les deux. (Se plaindre que ses parents n'ont pas acheté assez de cadeaux est aussi une option, mais ne nous emballons pas.) Puisque l'on peut avoir les deux, le « or » classique est considéré comme un « or » « inclusif ».

Vous vous demandez peut-être pourquoi nous abrégeons « exclusive » en « xor ». C'est parce que les informaticiens, et plus particulièrement les ingénieurs, sont violenement allergiques à l'écriture de tout mot de plus de trois ou quatre lettres. ([Vraiment.](#)) Tu penses Je fais ça va ? — Bon, d'accord, on exagère un peu, mais seulement un peu.)

6Voici un exemple du mensonge flagrant promis dans la préface. Comme promis, cela sonne bien mieux que la vérité. Le mensonge consiste à qualifier les opérations de « fondamentales », car la liste contient au moins une opération redondante, ce qui rend au moins l'une d'entre elles non « fondamentale ». En particulier, on peut réécrire un xor b comme [a et (non b)] ou [(non a) et b]. On peut donc déjà réduire la liste à ou, et, et non.

Mais ce n'est pas tout ! On peut aussi réécrire a et b comme non [(non a) ou (non b)]. (Ce dernier fait est connu sous le nom de lois de DeMorgan.) On peut donc réduire la liste des opérations logiques à ou et non. S'il est normal que le nombre d'opérations fondamentales sur deux objets (Vrai et Faux) soit de deux (non et ou), personne sensé ne le fait. Il est bien mieux de vous mentir et de vous dire que les quatre opérations sont « fondamentales ».

Vous vous demandez peut-être si les parenthèses sont vraiment nécessaires ici. Dans ce cas, non, car Sage teste toujours « and » avant de tester « or ». En général, cependant, il est recommandé d'indiquer explicitement l'ordre des opérations, car il est très important de savoir si vous voulez dire « and ».

```
sage : Faux et Faux ou Vrai
Vrai

sage : Faux et (Faux ou Vrai)
FAUX

sage : (Faux et Faux) ou Vrai
Vrai
```

Assurez-vous de bien comprendre pourquoi nous obtenons ces résultats.

Nous pouvons maintenant énoncer un code Sage différent pour la méthode de bisection.

```
sage : def méthode_de_bisection(a, b, n, f, x=x) : c, d = a, bf(x) = f pour i
dans la plage(n) :

# calculer le point médian, puis comparer e = (c + d)/
2 si (f(c) > 0 et f(e) >
0) ou (f(c) < 0 et f(e) < 0) :
    c = e
autre
    d = e
retour (c, d)
```

Une autre relation booléenne, et une mise en garde concernant son utilisation. Jusqu'à présent, nous nous sommes appuyés sur les relations booléennes `==`, `<`, `>`, `<=` et `>=`. Vous avez peut-être remarqué que nous n'avons pas indiqué d'opérateur pour l'inégalité. Techniquement, nous pouvons traduire le pseudo-code.

```
a = b
```

comme

```
sage : non (a == b)
```

Mais ce n'est pas très élégant. Sage nous propose un symbole différent dans ces circonstances, `!=`, qui ressemble à une façon approximative de barrer un signe égal, puis de laisser tomber de l'encre.

L'égalité et l'inégalité peuvent être dangereuses lors des comparaisons. Les nombres exacts ne posent aucun problème, mais si votre valeur est à virgule flottante, l'ordinateur pourrait croire à tort que deux nombres sont différents alors qu'ils ne le sont pas. Par exemple :

```
sage : 14035706228479900. - 14035706228479899.99 != 0 Faux
```

C'est une conclusion très étrange, due à l'arrondissement des nombres à virgule flottante. D'un autre côté, rappelons-nous, p. 109, le résultat de la méthode d'Euler, dont nous avons conclu qu'il était égal à 8.

```
sage : 8 - 7.99999999999999 != 0 Vrai
```

Imaginez un algorithme qui se termine seulement s'il a effectivement calculé 0. Nous avons probablement une égalité ici, mais l'ordinateur ne la détecte pas encore (voire pas du tout).

Dans ce cas, la solution consiste à vérifier que la différence entre les valeurs est suffisamment faible, plutôt que de vérifier leur égalité. Par exemple, si une valeur inférieure à dix décimales suffit, le test suivant corrigera le précédent.

```
sauge : 8 - 7,99999999999999 > 10**(-10)
FAUX
```

## Briser une boucle

Il y aura des cas où il sera inutile de continuer une boucle. Prenons l'exemple de l'application de la méthode de bisection à une fonction dont la racine est un nombre rationnel avec une puissance de 2 au dénominateur. Par exemple,  $f(x) = 4x + 3$  a une racine en  $x = -3/4$ . Même si l'on répète la méthode de bisection 20 fois, il est fort probable qu'elle se termine beaucoup plus tôt. Supposons, par exemple, que l'on commence sur l'intervalle  $(-1, 1)$ . La première bisection donne l'intervalle  $(-1, -1/2)$ , tandis que la seconde donne l'intervalle  $(-1, -3/4)$ . À ce stade, nous avons déjà trouvé une racine  $f(-3/4) = 0$  ; il faudrait donc abandonner, mais l'algorithme ne permet pas de le tester. Au lieu de cela, il continue pendant 18 itérations supplémentaires, se terminant par l'apparence peu maniable  $(-393217/524288, -3/4)$ .

Une meilleure approche consisterait à reformuler la boucle afin qu'elle s'arrête dès qu'elle trouve une racine. Tester cela est simple dans Sage, mais comment lui indiquer d'arrêter la boucle ? Une solution consisterait à placer une instruction `return` dans la boucle, mais ce n'est pas une bonne idée si un algorithme recherche une valeur pour ensuite effectuer un calcul avec elle avant de retourner la boucle.

C'est là qu'intervient le mot-clé `break`. Une instruction `break` indique à Sage d'arrêter la boucle en cours et de poursuivre à partir de la première instruction située hors du bloc indenté de la boucle. Un `break` ne s'applique qu'une seule fois ; ainsi, si la boucle est imbriquée dans une autre boucle, Sage n'en sortira pas.

Nous pouvons maintenant reformuler notre procédure `method_of_bisection()` comme suit.

```
sage : def méthode_de_bisection(a, b, n, f, x=x) : c, d = a, bf(x) = f pour i dans
        la plage(n) :
            # calculer le point médian, puis comparer e = (c + d)/
            2 si (f(c) > 0 et f(e) >
            0) ou (f(c) < 0 et f(e) < 0) :
                c = e
            elif f(e) == 0:
                c = d = e
                casser
            sinon :
                d = e
            retour (c, d)
```

Cette fois, lorsque le code calcule  $e == -3/4$ , il détermine que • la condition pour

la première instruction if est fausse, car ni  $f(e) > 0$  ni  $f(e) < 0$  ; mais • la condition pour la deuxième instruction if est vraie, car  $f(e) == 0$ .

Le code doit donc attribuer  $-3/4$  à c et d, puis exécuter l' instruction break . La première instruction en dehors du bloc indenté de la boucle est return (c, d), le code renverra donc le tuple  $(-3/4, -3/4)$ .

```
sage : méthode_de_bisection(-1, 1, 20, 4*x + 3) (-3/4, -3/4)
```

## Exceptions

Les exceptions sont l'un des concepts les plus récents de la programmation informatique et sont liées à la prise de décision, ne serait-ce que parce qu'elles peuvent remplacer les instructions if/else dans de nombreuses situations.

Nous en discutons brièvement ici.

Supposons que vous ayez un algorithme qui doit diviser :

soit  $c = n/d$

Prenons l'exemple d'un algorithme de division pour certains objets mathématiques. Cette affirmation anodine cache un danger mathématique : que se passe-t-il si  $d = 0$  ?

Nous pourrions bien sûr protéger l'énoncé avec une sentinelle if/else :

si  $d = 0$ ,

soit  $c = n/d$ ,  
sinon

faites quelque chose d'intelligent dans le cas  $d = 0$  — réprimandez l'utilisateur, dites

Bien sûr, le code peut contenir d'autres variables avec leurs propres potentiels de valeurs dangereuses, conduisant au code laid suivant :

```

si condition1
    si condition2
        si condition3
            si condition4
                ...
                sinon si condition5
                    ...
                    autre
                    ...
                    sinon si condition6
                        ...
                        autre
                        ...

```

Ce n'est pas seulement laid, c'est difficile à lire. Les informaticiens ont un nom technique pour cela : la pyramide de la mort.<sup>7</sup>

En pseudo-code, nous essayons de lister les problèmes avant qu'ils ne surviennent, en spécifiant dans la section des entrées que, par exemple,  $d = 0$ . Par exemple, un pseudo-code qui s'attend à prendre une dérivée à un point arbitraire dirait probablement quelque chose comme ceci :

```

entrées
    • f , une fonction différentiable partout sur la droite réelle

```

... mais nous ne le faisons pas systématiquement, surtout lorsque cela devrait être évident compte tenu de la finalité de l'algorithme. Après tout, il s'agit de pseudo-code, censé être lisible. Si nous spécifions chaque contrainte, la section des entrées deviendrait une fastidieuse récitation de formules. Les mathématiciens se fient souvent à l'intuition du lecteur.

Comme ce point de sortie n'est pas présent dans le code réel, nous adoptons une approche différente, appelée gestion des exceptions. Cette gestion repose sur un bloc `try/except`. Comme son nom l'indique, il teste un bloc de code indenté sous l'instruction `try` et, en cas d'erreur, applique un autre bloc de code apparaissant sous une instruction `except`. `try` et `except` sont des mots-clés ; ils ne peuvent donc pas être utilisés comme identifiants.

L'utilisation précise est la suivante :

```
sage : essayez :
    première_instruction_d'essai
    deuxième_instruction_d'essai
    ...
sauf ExceptionList :
    première_exception_déclaration
    deuxième_exception_déclaration
```

Lorsque Sage rencontre un tel bloc de code, il essaie first\_try\_statement, second\_try\_statement, ... . S'il peut effectuer toutes les tâches répertoriées sous l'instruction try , il ignore le bloc except entièrement, et continue.

Cependant, si Sage rencontre une erreur, il la compare aux exceptions répertoriées dans la liste des exceptions. Si l'une d'elles correspond, Sage exécute first\_exception\_statement, second\_exception\_statement, et ainsi de suite, jusqu'à la fin, sauf si une autre erreur se produit. Dans ce cas, Sage abandonne et renvoie l'erreur au client. Voici un

exemple que vous pouvez tester sans même écrire de procédure :

```
sage : essayer :
1/0
sauf ZeroDivisionError :
    Infini
```

Si vous tapez ceci dans une cellule ou sur la ligne de commande, puis l'exécutez, Sage essaie d'abord de diviser 1 par 0. Cela ne fonctionnera évidemment pas, et Sage génère une erreur ZeroDivisionError. Notre clause except détecte cette erreur ; Sage passe donc dans ce bloc et trouve l'instruction Infinity. Ne soyez donc pas surpris du résultat : 9.

---

Si vous ne connaissez pas le type d'erreur, si vous souhaitez traiter toutes les erreurs possibles avec le même code ou si vous êtes particulièrement paresseux, il est possible de détecter toutes les erreurs en omettant l'exception `ExceptionType`. Les auteurs de ce texte se sentent souvent paresseux, mais pour des raisons pédagogiques, nous pensons qu'il est préférable d'éviter cette pratique.

Vous souhaiterez peut-être parfois traiter les informations renvoyées par l'exception. Dans ce cas, vous pouvez utiliser la construction

```
sauf ExceptionType comme e:
```

et puis regardons les détails de e. Nous n'entrions pas dans ces possibilités.

<sup>9</sup> Il est également possible d'« imbriquer » des clauses try... except , c'est-à-dire de placer une clause try... except dans une autre. Vous pouvez le faire. Ceci dans une clause try ou except . Par exemple :

```
sage : essayer :
1/0
sauf ZeroDivisionError : essayez :

    0**Infini sauf
    NotImplementedError : print('Pas de
    dé')
```

+Infini

Plutôt que de traiter toutes les exceptions dans un seul bloc `except`, il est possible de traiter les exceptions d'un bloc `try` dans plusieurs blocs `except` suivants. Alignez chaque clause `except` avec la clause `try`, indiquez l'erreur précise pour chaque exception et fournissez le code indenté en conséquence. Cela permet d'éviter que les structures `try/except` ne se transforment en une pyramide de la mort.

En plus de détecter les erreurs, une procédure peut invoquer un autre mot-clé, `raise`, pour « éléver » comporte ses propres erreurs. Son utilisation est la suivante :

sage : éléver ErrorType(message)

Pour le message, vous pouvez utiliser n'importe quelle chaîne. Pour le type d'erreur, vous pouvez lever n'importe quelle exception du type approprié ; les deux suspects habituels sont :

- `TypeError`, lorsque l'entrée est du mauvais type : par exemple, l'algorithme attend un matrice, mais a reçu un numéro ; et
- `ValueError`, où l'entrée a le bon type, mais la mauvaise valeur : par exemple, l'algorithme attend un nombre différent de zéro, mais a reçu zéro.

Nous n'utiliserons pas le mot-clé `raise` en général, mais vous devez savoir que ce mécanisme est la manière dont les auteurs de Sage vous communiquent les erreurs.

Essayons cette approche sur quelque chose de plus substantiel.

Un exemple « normal ». Un exercice de la page 104 vous demandait d'écrire une procédure Sage pour calculer la droite normale à une fonction mathématique  $f$  en  $x = a$ . La procédure que vous avez écrite devrait fonctionner dans la plupart des situations.

```
sage : était('t')
t
sage : normal_line(t**2, 1, t) 1/2*t + 1/2
```

Cependant, selon toute vraisemblance, votre code rencontrerait l'erreur suivante :

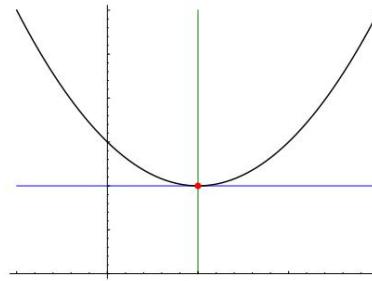
```
sage : normal_line((t - 1)**2 + 2, 1, t)
ZeroDivisionError : division symbolique par zéro
```

Eh bien, bien sûr, cela se produirait : •

- la droite normale est perpendiculaire à la droite tangente ; •
- une droite perpendiculaire est une réciproque négative ; et
- la pente de la droite tangente à  $(t - 1)^2 + 2$  à  $t = 1$  est 0.

« Réciproque » signifie « division » et division par zéro signifie `ZeroDivisionError`.

Ce problème est inévitable. Il nous faut trouver une solution, et il se trouve que nous en avons au moins deux. Mais il faut d'abord réfléchir au problème sous-jacent : la dérivée est nulle. Dans ce cas, la droite normale est en réalité la droite verticale  $x = a$ . Illustrons cela avec la fonction  $f(t) = (t - 1)^2 + 2$  à  $t = 1$  :



Une ligne verticale n'est pas une fonction ; une solution consiste donc à lever notre propre exception. Quelque chose comme ça, par exemple ?

```
raise ValueError('La ligne normale est verticale.')
```

Cela fonctionne, et `ValueError` semble être l'exception appropriée, puisque le type est très bien (une fonction, et une fonction différentiable, pour démarrer) mais sa valeur est inappropriée.

Le problème avec cette approche est qu'elle signale une erreur alors que la situation est réellement récupérable ! Après tout, une droite normale existe ; ce n'est simplement pas une fonction. Une approche plus appropriée pourrait consister à renvoyer une équation pour la droite normale.

renvoie  $x=a$

Cela pourrait rendre votre algorithme d'origine incohérent si vous renvoyiez une fonction (ce qui était probablement le cas). Nous pouvons néanmoins rendre l'algorithme cohérent en modifiant le cas normal pour renvoyer l'équation  $y = m(x - a) + f(a)$  au lieu du membre de droite seul.

Alors, comment implémenter cela dans Sage ? Là encore, deux options s'offrent à nous. La première consiste à utiliser une condition `if/else`. structure comme « garde » autour du calcul de  $m$ . Ainsi, aucune exception ne se produit.

```
sage : définition de ligne_normale_avec_garde(f, a, x=x) :
    f(x) = f df(x)
    = diff(f, x) si df(a) == 0 : #
        résultat horizontal
        perpendiculaire à vertical = x == a

    autre:
        m = 1/df(a)
        résultat = y == m*(x - a) + f(a)
        résultat de retour
```

L'autre approche consiste à utiliser une structure `try/except` pour intercepter une `ZeroDivisionError`.

```
sage : définition de ligne_normale_avec_catch(f, a, x=x) :
    f(x) = f df(x)
    = diff(f, x) essayez : m = 1/df(a)

    résultat = y == m*(x - a) +
        f(a) sauf ZeroDivisionError :

    # résultat horizontal perpendiculaire à vertical = x == a

    résultat de retour
```

Essayez-les tous les deux pour voir que les deux procédures fonctionnent avec les lignes normales verticales et obliques :

```
sage: ligne_normale_avec_garde((t - 1)**2 + 2, 0, t) y == -1/2*t + 3 sage:
ligne_normale_avec_garde((t
- 1)**2 + 2, 1, t) t == 1

sage: ligne_normale_avec_catch((t - 1)**2 + 2, 0, t) y == -1/2*t + 3 sage:
ligne_normale_avec_catch((t
- 1)**2 + 2, 1, t) t == 1
```

Quelle approche est « meilleure » ? La deuxième version est généralement recommandée pour deux raisons : • La logique

est plus fluide, donc plus lisible. La construction if/else n'explique pas clairement pourquoi nous testons  $df(a) == 0$ , mais simplement que nous le testons. Elle oblige le lecteur à réfléchir davantage à ce que nous faisons. En revanche, la construction try/except indique clairement qu'une erreur peut se produire (la seule raison d'utiliser un try) ; si oui, quelle est cette erreur (ZeroDivisionError) ; et de plus, que faire si elle se produit (le bloc except). • Rappelons que l'interface de Sage utilise Python. En termes de langages informatiques, les exceptions de Python sont efficaces. L'utilisation d'exceptions entraîne une légère pénalité ; dans nos tests, la garde if/else est environ 20 % plus rapide que la capture try/except lorsque la division par zéro se produit réellement.

Ne tirez pas de conclusions hâtives ! La division par zéro est rare, il faut donc se demander comment ces approches se comparent en temps normal. Il s'avère que la fonction try/except catch est plus de 30 % plus rapide que la fonction if/else guard !

Un exemple plus complexe. À la page 281, nous décrivons la méthode de Dodgson pour calculer un déterminant. Cette méthode est un algorithme implantable à l'aide d'une boucle for ; votre instructeur l'a peut-être déjà utilisée. C'est un algorithme intéressant ; si vous ne l'avez pas encore essayé, essayez-le vite.

Malheureusement, il présente des problèmes difficiles à résoudre. Un autre algorithme de calcul des déterminants repose sur l'élimination gaussienne. Il présente également des problèmes, mais ceux-ci sont faciles à résoudre. Commençons par le pseudo-code ci-dessous, qui compte les lignes et les matrices à partir de 1.

algorithme déterminant gaussien  
entrées

- M, une matrice  $n \times n$

sorties

- $\det M$

soit N une copie de M  
soit d = 1

pour i {1, ..., n - 1}

pour j {i + 1, ..., n} si

$N_j, i = 0$  soit

$a = N_j, i$

multiplier la ligne j par  $N_i, i$

soustraire a fois la ligne i de la ligne j diviser

d par  $N_i, i$  renvoyer

$d \times N_{1,1} \times N_{2,2} \times \dots \times N_{n,n}$

Avant de le mettre en œuvre, voyons-le en action sur

$$M = \begin{matrix} 3 & 2 & 1 & 5 \\ & 4 & 3 \\ & 6 & 3 & 4 \end{matrix}$$

L'algorithme commence par copier M dans N et définir d = 1. Il parcourt ensuite toutes les lignes de N sauf la dernière, en stockant la valeur de la ligne dans i.

Avec i = 1, la boucle externe trouve une boucle interne avec j = 2. Comme  $N_{2,1} = 5 = 0$ , l'algorithme stocke 5 dans a, multiplie la ligne 2 par  $N_{1,1} = 3$ , soustrait a = 5 fois la ligne 1 de la ligne 2, puis divise d par  $N_{1,1} = 3$ , ce qui donne  $d = 1 \div 3 = 1/3$ . La matrice résultante est

$$N = \begin{matrix} 3 & 2 & 1 \\ & 0 & 2 & 4 & 6 \\ & 3 & 4 \end{matrix}$$

La boucle interne définit ensuite j = 3. Comme  $N_{3,1} = 6 = 0$ , l'algorithme exécute à nouveau le bloc le plus interne sur la ligne 3, obtenant

$$N = \begin{matrix} 3 & 2 & 1 & 0 & 2 \\ & 4 & 0 & -3 & 6 \end{matrix}$$

Il a également modifié d, de sorte que  $d = 1/9$ . Jusqu'ici, tout va bien. L'algorithme a bouclé la boucle interne sur j lorsque i = 1.

La boucle externe continue jusqu'à i = 2 et trouve une boucle interne avec j = 3. Comme  $N_{3,2} = -3 = 0$ , l'algorithme stocke -3 dans a, multiplie la ligne 3 par  $N_{2,2} = 2$ , soustrait a = -3 fois la ligne 2 de la ligne 3,

puis divise d par  $N[i,i] = 2$ , ce qui donne  $d = (1/9) \div 2 = 1/18$ . La matrice résultante est

$$\begin{matrix} & 3 & 2 & 1 & 0 \\ N = & 2 & 4 & 0 & 0 \\ & 24 \end{matrix}$$

L'algorithme a terminé la boucle interne sur j lorsque  $i = 2$ . C'était la dernière valeur de i, donc les boucles sont terminées ; l'algorithme renvoie

$$d \times N[1,1] \times N[2,2] \times N[3,3] = 1/18 \times (3 \times 2 \times 24) = 8.$$

Soit Sage, soit une méthode plus traditionnelle confirmera que  $\det M = 8$ .

Nous pouvons implémenter cela dans Sage dans le code suivant :

```
sage : définition du déterminant gaussien (M) :
N = copie(M)
d = 1
n = N.nrows() pour i
dans la plage(n-1) : pour j dans
la plage(i+1,n) :
sinon (N[j,i] == 0) :
    # effacer la colonne sous cette ligne
    d = d/N[i,i] a = N[j,i]
    N.set_row_to_multiple_of_row(j,j,N[i,i])
    N.add_multiple_of_row(j,i,-a) renvoie
d*prod(N[i,i] pour i dans la plage(n))
```

Si nous l'essayons sur les matrices restantes données dans l'exercice de la page 281, nous constatons que cela fonctionne pour la troisième matrice, mais pas pour la seconde. Qu'est-ce qui ne va pas ?

ZeroDivisionError : division rationnelle par zéro

La division n'intervient qu'à un seul endroit de l'algorithme : d par  $N[i,i]$ . Nous avons dû rencontrer un 0 sur la diagonale principale.

Comment cela est-il arrivé ? Nous avons commencé avec

$$\begin{matrix} & 1 & -4 & 1 & 2 & -1 & 4 \\ M = & 4 & 1 & 3 & 2 & 5 & 2 & -1 \\ & 3 & 3 & 4 \end{matrix},$$

et le premier passage à travers la boucle extérieure nous laisse avec

$$\begin{matrix} & 1 & -4 & 1 & 2 & 0 & 0 & 5 \\ N = & 3 & 0 & 15 & 0 & -2 & 0 \\ & 13 & 0 & -5 \end{matrix}.$$

Et voilà ! Lors du deuxième passage dans la boucle, l'algorithme divise d par 0, un désastre.

Contrairement à la méthode de Dodgson, une solution simple est disponible. Rappelons que l'élimination gaussienne permet d'échanger les lignes, ce qui réorganise simplement les équations correspondant à chaque ligne. Si nous trouvons une autre ligne sous  $i = 2$  avec un élément non nul dans la colonne 2, nous pouvons échanger cette ligne avec la ligne 2 et procéder comme précédemment. Dans l'exemple ci-dessus, la ligne 3 a un élément non nul dans la colonne 2 ; nous l'échangeons donc avec la ligne 2, ce qui donne

$$N = \begin{array}{r} 1 -4 1 2 0 15 \\ 0 -2 0 0 5 3 0 \\ 13 0 -5 \end{array} .$$

Avec un élément non nul à la ligne 1, colonne 1, nous pouvons maintenant reprendre l'algorithme. Lorsque  $j = 3$ , il n'y a rien à faire ; la seule boucle interne qui fonctionne est lorsque  $j = 4$ , obtenant

$$N = \begin{array}{r} 1 -4 1 \quad 2 \\ 0 15 0 -2 \\ 0 0 5 \\ 3 0 0 0 -49 \end{array}$$

et  $d = 1/15$ . Rien ne se produit non plus sur la valeur suivante de  $i$ , donc l'algorithme renvoie

$$\frac{1}{15} \times [1 \times 15 \times 5 \times (-49)] = -245.$$

Encore une fois, nous pouvons vérifier en utilisant Sage ou un algorithme traditionnel de calcul des déterminants que  $\det M$  est ... euh, 245.

Quoi ? On a eu -245 !

## PANIQUE!

Une fois que nous avons fini de paniquer, réfléchissons à ce qui aurait pu changer, en particulier à ce qui aurait pu faire que le panneau se détériore.

La seule chose qui a changé, c'est que nous avons échangé les lignes. Rappelez-vous, d'après votre expérience avec les matrices, que l'échange de lignes modifie le signe du déterminant. Voilà ! Notre méthode pour « sauver » l'algorithme impliquait d'échanger les lignes ; il fallait donc s'assurer de multiplier le déterminant par  $-1$ . Ce n'est pas trop compliqué ; il suffit de multiplier  $d$  par  $-1$ .

Avant d'implémenter cet algorithme amélioré, nous devons nous demander : que se serait-il passé si nous n'avions pas trouvé d'élément non nul dans la colonne 2 parmi les lignes situées en dessous ? Dans ce cas, la structure de la matrice indique que le déterminant est nul. Ceci couvre toutes nos hypothèses.

Nous pourrions utiliser une structure de contrôle `if/else` pour implémenter l'algorithme modifié, ce qui fonctionnerait parfaitement (voir les exercices). Nous utilisons cette fois une structure `try/except`, notamment pour illustrer son fonctionnement. Cependant, plutôt que de regrouper tout ce contenu dans une seule procédure, divisons les tâches en procédures distinctes, chacune traitant un sous-problème de petite taille.

Une procédure intitulée `clear_column()` pourrait gérer la tâche ordinaire de réduction de lignes. une chose impliquant la sauvegarde d'une division zéro pourrait entrer dans une procédure intitulée, `unzero()`.

```

sage : def unzero(M, d, i) : # utiliser
    pour échanger des lignes lorsque M[i,i] = 0 n =
    M.nrows() #
    rechercher la ligne kw/non nulle dans la colonne j
    pour k dans la plage(i+1,n) :
        sinon (M[k,i] == 0) :
            M.swap_rows(k,i) d = -d

    casser
    # M[i,i] == 0 ? colonne vide ("échec") si M[i,i] == 0 : succès = Faux
    sinon :

    d = d/M[i,i]
    succès = Vrai
    retour réussi, M, d

```

La procédure `clear_column()` pourrait alors gérer la tâche simplifiée d'effacement des éléments restants de la colonne.

```

sage : def clear_column(M, d, i) : n = M.nrows()

    # effacer la colonne sous cette ligne
    pour j dans la plage (i+1, n) :
        sinon (M[j,i] == 0) : essayez : d =
            d/
            M[i,i] sauf
            ZeroDivisionError : # recherche du pivot
            dans la ligne inférieure succès, M, d = unzero(M,
                d, i)
            sinon succès :
                raise ZeroDivisionError('Le résultat est 0')
            a = M[j,i]
            M.set_row_to_multiple_of_row(j, j, M[i,i])
            M.add_multiple_of_row(j, i, -a) renvoie M, d

```

Cela nous permet de simplifier considérablement la procédure `gaussian_determinant()`, la rendant beaucoup plus compréhensible.

```
sage : définition du déterminant gaussien (M) :
N = copie(M)
d =
1
n = N.nrows() #
effacer les colonnes de gauche à droite pour i
dans la plage (n-1) : essayez : N,
d =
    clear_column(N, d, i) sauf ZeroDivisionError :
renvoie 0

renvoie d*prod(N[i,i] pour i dans la plage(n))
```

La procédure renvoie désormais le déterminant correct pour toute matrice que vous lui envoyez.

```
sage : M = matrice([[1,-4,1,2],[-1,4,4,1],[3,3,3,4],[2,5,2,-1]]) sage :
déterminant_gaussien(M)
```

### Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Sage propose la structure de contrôle if / else pour prendre des décisions.
2. Sage génère une SyntaxError si vous oubliez d'inclure les deux points à la fin d'un état if ou else. ment.
3. La logique booléenne est basée sur les trois valeurs Vrai, Faux et Parfois.
4. Vrai et (non (Faux ou Vrai))
5. Vrai ou (non (Faux et Vrai))
6. Vrai et (non (Faux xor Vrai))
7. Vrai ou (non (Faux xor Vrai))
8. Vous devez toujours utiliser le terme inclusif ou parce que **vous faites preuve de discrimination**. c'est toujours et partout mauvais.
9. Si un algorithme peut prendre trois actions possibles, la seule façon de l'implémenter est de la structure suivante :

```
si condition1 :
    bloc 1
autre:
    si condition2 :
        bloc 2
    autre:
        bloc 3
```

10. Une capture try/except est moins efficace qu'une garde if/else , mais cette pénalité en vaut la peine pour des raisons de lisibilité.

Choix multiple.

1. La meilleure structure de contrôle pour une décision générale utilise quel(s) mot(s) clé(s) ?

A. try/except B. def

C. if/elif/

else D. for/break 2.

Quelles valeurs

une instruction booléenne dans Sage peut-elle avoir ?

A. Activé,

Désactivé B. Vrai,

Faux C. Vrai,

Faux D.

1, -1 3. Que signifie « imbriquer » des structures de contrôle ?

A. Placer une structure de contrôle (comme une instruction if ) à l'intérieur d'une structure de contrôle différente (comme une instruction for ).

B. Placer une structure de contrôle (comme une instruction if ) à l'intérieur de la même structure de contrôle (une autre instruction if ).

C. Programmer de telle manière que les structures de contrôle ne soient pas nécessaires.

D. Construire deux structures de contrôle, une maison agréable et confortable où ils pourront élever une famille des structures de contrôle.

4. Qu'indiquent les commandes à exécuter lorsque la condition d'une instruction if est vraie ?

A. deux points à la fin de la ligne B.

indentation des commandes C. accolades

{ et }

D. à la fois un deux-points et une indentation

5. Quel mot clé indique qu'une boucle doit se terminer plus tôt car elle est effectivement terminée ?

A.

abandonner

B.

interrompre C. quitter D. revenir

6. Quel symbole indique à Sage que deux expressions ne sont pas égales ?

A. /= B. !

= C. #

D.

Aucun ; vous devez utiliser not (a == b)

7. Prendre la racine carrée d'un nombre négatif produit quel type d'exception ?

A. ZeroDivisionError B.

TypeError C.

ValueError D.

Pourquoi devrait-il produire une exception ?

8. La meilleure structure de contrôle pour surveiller une erreur spécifique utilise quel(s) mot(s) clé(s) ?

A. try/except B. def

C. if/elif/

else D. for/break 9.

Laquelle des

actions suivantes n'est pas exécutée dans une instruction try... except ?

A. Sage exécute chaque instruction indentée sous l' instruction try qui ne génère pas d'erreur erreur.

- B. Si aucune erreur ne se produit dans les instructions répertoriées sous le bloc `try` , Sage ignore les instructions en retrait sous le bloc `except` .
- C. Si une erreur se produit dans le bloc `try` et que l'instruction `except` suivante répertorie cette erreur, ou aucune, Sage exécute chaque instruction indentée sous l' instruction `except` qui ne génère pas d'erreur.
- D. Si une erreur se produit dans les instructions répertoriées sous le bloc `except` et que l'instruction `except` suivante répertorie cette erreur, Sage exécute chaque instruction indentée sous l' instruction `except` suivante qui ne génère pas d'erreur.
10. Traditionnellement, que signifie la « pyramide du destin » ?
- une situation où nous devons imbriquer les instructions `try` très profondément
  - une situation où un `if` a de très nombreux autres `if`
  - une situation où nous devons imbriquer les instructions `if` très profondément
  - encore une autre suite dans la série Indiana Jones
11. Une bonne raison d'utiliser des parenthèses lorsqu'une condition comporte plusieurs opérateurs booléens est que :
- ...cela clarifie la logique de la condition à quiconque la lit.
  - ...il évite les bugs subtils dus à l'ordre dans lequel Sage évalue et, ou, et non.
  - tout ce qui précède
  - aucun de ces éléments

Réponse courte.

- Comparez et opposez les moments appropriés pour utiliser `if/elif/else` par opposition aux moments appropriés. Il est parfois difficile d'utiliser `try/except/raise`.
- Comment modifieriez-vous le pseudo-code `Method_of_Bisection` pour tester que  $f(a)$  et  $f(b)$  ont des signes différents avant de démarrer la boucle et de renvoyer  $(a, b)$  si'ils ne le font pas ?
- Si quelqu'un devait modifier l'implémentation de `method_of_bisection()` pour générer une exception, tion dans le cas où  $f(a)$  et  $f(b)$  étaient du même signe :
  - Quel type d'exception devraient-ils utiliser, `TypeError`, `ValueError` ou autre chose d'autre ensemble?
  - Quel type de structure devraient-ils utiliser, si/sinon ou essayer/sauf ?
- Écrivez une procédure `abmatrix()` qui accepte en entrée un entier positif  $n$  et deux objets  $a$  et  $b$ , puis renvoie la matrice  $n \times n$   $C$  où  $C$  est la matrice ci-dessous.

$$\begin{array}{c}
 n \text{ colonnes} \\
 \hline \hline \\
 abababba \cdots \\
 \hline \hline \\
 abababba \\
 \vdots \quad \ddots \\
 \hline \hline
 \end{array}$$

n lignes

Puisque  $a$  et  $b$  peuvent être symboliques, vous devez créer cette matrice sur l'anneau symbolique. (a) Calculez  $\det C$ . (b)

Pourquoi ce résultat est-il logique ? Indice : Pensez à l'indépendance linéaire, ou plutôt à la absence de celle-ci.

Programmation.

1. Écrivez une procédure qui calcule le plus grand élément d'une liste. Sage possède une procédure `max` intégrée , mais utilisez une boucle ici.
2. Écrivez des procédures Sage qui :
  - (a) renvoient le plus grand élément d'une matrice ; (b)
  - renvoient le plus petit élément d'une matrice ; (c)
  - renvoient la somme de tous les éléments d'une matrice.
3. Écrivez le pseudo-code d'une procédure Sage qui accepte en entrée une fonction mathématique  $f$  et un nombre réel  $a$ , puis renvoie un tuple avec deux valeurs : si  $f$  augmente à  $x = a$  (vrai si c'est le cas) et si  $f$  est concave vers le haut à  $x = a$  (vrai si c'est le cas).
4. Écrivez une procédure `lp()` qui accepte comme arguments deux listes  $L$  et  $M$ , vérifie qu'elles ont la même longueur et, si c'est le cas, renvoie une liste  $N$  de même nombre d'éléments, où  $N[i]$  est le produit de  $L[i]$  et  $M[i]$ . Si les deux listes n'ont pas la même longueur, la procédure doit générer une erreur `AttributeError`. Par exemple, le résultat de `LP([1,3,5],[2,4,6])` serait `[2,12,30]`, tandis que l'invocation `LP([1,3],[2,4,6])` généreraît l'erreur.
5. Lorsqu'on demande à Sage de résoudre une fonction trigonométrique, il ne renvoie qu'une seule solution. Par exemple, la solution de  $\sin 2x = 1/2$  est en réalité  $\{\pi/12 + \pi k\} \cup \{5\pi/12 + \pi k\}$ . Écrivez une procédure pour ajouter des solutions périodiques à une équation trigonométrique de la forme  $\sin(ax + b) = 0$ .
 

Astuce : Une partie de votre programme devra résoudre  $ax + b = 0$ . Vous pouvez obtenir cette information grâce à la méthode `.operands()`. Pour comprendre son fonctionnement, assignez  $f(x) = \sin(2*x + 3)$  puis saisissez `f.operands()`.
6. Dans ce problème, vous écrirez une procédure interactive qui approxime des intégrales définies dans
 

Il existe trois façons

de procéder. (a) Adaptez le pseudo-code `Left_Riemann_approximation` pour écrire un pseudo-code pour l'approximation de Riemann utilisant des extrémités droites. Traduisez-le en code Sage sans utiliser de compréhensions. Vérifiez que votre code produit des approximations précises. (b)

Adaptez le pseudo-code `Left_Riemann_approximation` pour écrire un pseudo-code pour l'approximation de Riemann utilisant des points médians. Traduisez-le en code Sage sans utiliser de compréhensions. Vérifiez que votre code produit des approximations précises. (c) Écrivez une procédure interactive avec les objets d'interface suivants :

  - une zone de saisie pour une fonction  $f(x)$  ; • une zone de saisie pour une extrémité gauche  $a$  ; • une zone de saisie pour une extrémité droite  $b$  ; • un curseur pour un nombre  $n$  d'approximations, avec une valeur minimale de 10 et une valeur maximale. 200, et un pas de 10 ; et enfin,
  - un sélecteur avec les options « approximation à gauche », « approximation à droite » et « point médian » approximation."

Le corps de la procédure appellera la procédure Sage que vous avez écrite pour l'approximation demandée et renverra sa valeur.
7. En général, un utilisateur de la méthode de bisection ne sait pas à l'avance combien d'étapes la boucle principale de l'algorithme doit effectuer. Il a en tête la précision décimale nécessaire entre les points de terminaison : par exemple, ils doivent être identiques au millième près. Une façon d'implémenter cela est de remplacer l'entrée  $n$  par un entier positif  $d$  spécifiant le nombre de chiffres. Nous pouvons ensuite calculer  $n$  à partir de  $d$  dans le corps de l'algorithme. (a) Modifiez le pseudo-code de la méthode de bisection afin qu'il accepte  $d$  au lieu de  $n$  et calcule  $n$  comme toute première étape.

Astuce : En général, nous arrondissons  $\log_{10} n$  à l'entier supérieur suivant pour trouver le nombre de chiffres

en n. ( $\log_{10} 2 \approx 0 \rightarrow 1$ ,  $\log_{10} 9 \approx 1 \rightarrow 1$ ,  $\log_{10} 10 = 1 \leftarrow 1$ , ...) Donc si l'algorithme est divisé par 10, vous pourriez le répéter d fois pour obtenir une précision de d chiffres. Hélas ! l'algorithme divise par 2, vous devez donc utiliser un  $\log_2$ . La précision avec laquelle vous devez l'utiliser, nous vous laissons la tâche, mais vous devrez également considérer b – a, pas seulement d. (b) Implémentez le pseudo-code dans le code Sage.

8. Écrivez des procédures Sage qui acceptent en entrée une matrice M et renvoient True si M satisfait l'une des conditions suivantes ces conditions, et False sinon : • `is_square()` : M est

carré • `is_zero_one_negative()` :

chaque entrée de M est 0, 1 ou -1 • `sums_to_one()` : chaque ligne et chaque colonne de M est égale à 1

Écrivez ensuite une procédure Sage acceptant une matrice M en entrée et renvoyant « Vrai » si les trois conditions sont remplies, et « Faux » dans le cas contraire. Rendez cette procédure modulaire afin qu'elle invoque les trois procédures précédentes au lieu de répéter leur code !

## CHAPITRE 8

## Se répéter indéfiniment

Nous avons précédemment parlé de la répétition « définitive », qui se produit lorsque l'on sait exactement combien de fois il faut répéter une tâche. Nos exemples comprenaient la méthode d'Euler pour approximer la solution d'une équation différentielle, les sommes de Riemann pour approximer une intégrale et la vérification de la formation d'un corps par les éléments d'un anneau fini. Plus tard, nous avons utilisé les mêmes idées pour trouver les racines d'une fonction continue par la méthode de la bisection et pour modifier des matrices.

Toutes les boucles ne sont pas définies ; il est parfois difficile de savoir d'emblée combien de fois une tâche doit être répétée. La méthode de Newton en est un exemple. Elle peut être plus rapide que la méthode de la bisection, mais elle utilise des tangentes ; elle exige donc que la fonction soit différentiable, et pas seulement continue.

(Quelle est la différence entre une fonction continue et une fonction différentiable, demandez-vous ? On peut envisager la différence de manière géométrique : le graphe d'une fonction continue est « ininterrompu », tandis que celui d'une fonction différentiable est « lisse ». Une fonction lisse est nécessairement ininterrompue, tandis qu'une fonction ininterrompue peut ne pas l'être. En fait, les fonctions différentiables sont toujours continues, tandis que les fonctions continues ne sont pas toujours lisses. Ainsi, la méthode de la bisection fonctionne partout où la méthode de Newton fonctionne, mais peut-être pas aussi rapidement ; tandis que la méthode de Newton ne fonctionne pas partout où la méthode de la bisection fonctionne. D'ailleurs, la méthode de Newton pourrait ne pas fonctionner partout où elle fonctionne<sup>1</sup>, selon votre négligence dans la configuration.)

L'idée de la méthode de Newton repose sur deux faits.

- La droite tangente à une courbe se déplace dans la même direction que cette courbe.
- Il est facile de trouver la racine d'une droite :  $mx + b = 0$  implique  $x = -b/m$ .

En les combinant, on constate que si l'on commence près de la racine d'une fonction, puis que l'on suit la tangente jusqu'à sa racine, on ne devrait pas s'éloigner trop de la racine de départ – et, avec un peu de chance, on sera plus proche qu'au départ. On peut répéter ce processus autant de fois que nécessaire jusqu'à obtenir la précision souhaitée en chiffres. Voir la figure 1.

Comment déterminer si l'on a atteint la précision souhaitée ? L'astuce habituelle consiste à arrondir chaque approximation au nombre de chiffres souhaité et à ne s'arrêter que lorsque l'on obtient deux fois la même valeur. Comme vous pouvez l'imaginer, il n'est pas évident de déterminer à l'avance le nombre d'étapes nécessaires pour y parvenir. Il est plus simple de tester, après chaque itération, si l'approximation précédente et l'approximation actuelle concordent avec le nombre de chiffres spécifié. Cependant, jusqu'à présent, nous n'avons aucun moyen de le faire.

---

<sup>1</sup>Cela peut paraître une curieuse combinaison de mots, mais c'est la vérité pure et dure, comme nous le montrerons bientôt. Si vous pensez que cela n'a pas de sens, consacrez-y quelques paragraphes et vous verrez. (Modulo, une certaine liberté artistique dans le sens des mots, bien sûr.)

FIGURE 1. Principe de base de la méthode de Newton : commencer près d'une racine, suivre la tangente et terminer à un point (idéalement) plus proche de la racine. Répéter aussi longtemps que souhaité.

#### Implémentation d'une boucle indéfinie

Lorsqu'on ne sait pas combien de fois répéter, mais qu'on peut appliquer une condition comme celle ci-dessus qui simplifie en Vrai ou Faux, on opte pour ce qu'on appelle une boucle while. Le pseudo-code d'une boucle while est le suivant :

condition pendant

et est suivi d'une liste d'instructions indentées. L'algorithme exécute chaque instruction dès que la condition est vraie au début de la boucle ; il ne teste pas la condition et s'arrête en cours de boucle si la condition devient soudainement vraie après l'une des instructions. Ceci se traduit directement en code Sage : utilisez le mot-clé `while` , suivi d'une condition booléenne et du signe deux-points.

condition while :

suivi d'une liste d'instructions indentées. Si vous le souhaitez, vous pouvez utiliser une commande `break` dans une boucle `while` , comme avec une boucle `for` , bien que nous ne le recommandions pas.

Pour implémenter la méthode de Newton, nous devons suivre deux valeurs : l'approximation actuelle de la racine, par exemple `a`, et l'approximation suivante de la racine, par exemple `b`. L'utilisateur spécifie `a`, et nous calculons `b`. Tant que `a` et `b` diffèrent sur au moins un des chiffres spécifiés, nous continuons la boucle.

Une petite complication se pose ici. Une boucle `while` teste l'égalité au début de la boucle, plutôt qu'à la fin. Or, nous calculons `b` à l'intérieur de la boucle ; nous ne pouvons donc pas connaître sa valeur au début, sauf si nous effectuons un calcul supplémentaire avant la boucle ! Cela gaspillerait de la place dans le programme et le rendrait un peu plus difficile à lire.

Pseudo-code. Pour garantir que l'algorithme effectue au moins une approximation, nous exploitons la nature du problème et créons une valeur manifestement erronée pour  $b$ . Dans ce cas, il serait extrêmement étrange que quelqu'un nous demande d'arrondir au chiffre des unités ; nous utilisons donc  $b = a + 2$  à la place. Dans ce cas, il serait extrêmement étrange que  $a$  et  $b$  concordent jusqu'à  $d$  décimales.

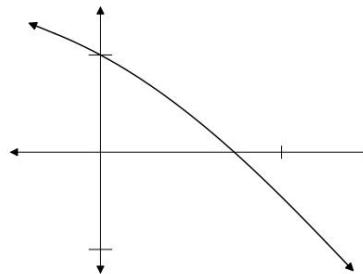
Pour décrire l'idée en pseudo-code, nous utilisons une troisième variable,  $c$ , comme espace réservé temporaire pour l'approximation la plus récente de la racine. L'utilisation d'un espace réservé nous permet d'organiser le calcul de manière à pouvoir passer des valeurs les plus anciennes aux plus récentes à chaque passage dans la boucle.

```

algorithme Méthode de Newton entrées
• f , une
fonction différentiable autour de a • a, une approximation d'une
racine de f • d, le nombre de chiffres pour
approximer une racine de f près de a sorties • une racine de f , corrigée à d chiffres
décimaux
soit
b = a + 2, c = a tant que
les d premiers chiffres décimaux de a et b diffèrent soit a = c soit b
la racine
de la droite tangente à f en x = a soit c = b
retourner un

```

Voyons comment cela fonctionne avec le même exemple que celui utilisé avec la méthode de bisection :  $f(x) = \cos x - x$ .  $x$  et  $\cos x$  sont tous deux différentiables, donc leur différence l'est aussi ; nous pouvons continuer. L'inspection du graphique montre qu'une racine devrait se trouver entre  $x = 0$  et  $x = 1$  ; comme  $x = 1$  est plus proche, nous commencerons l'algorithme avec  $a = 1$  et  $d = 3$ .



L'algorithme initialise  $b$  à 3 et  $c$  à 1 avant d'entrer dans la boucle. • Au début du premier passage dans la boucle, l'algorithme vérifie que les trois premiers chiffres de  $a = 1$  et  $b = 3$  diffèrent. Il attribue à  $a = 1$ , puis à  $b$  la racine de la droite tangente à  $\cos x - x$ . Pour trouver cela, nous avons besoin de l'équation de la droite tangente ; cela nécessite un point,

<sup>2</sup>Une autre façon plus sûre de procéder est de définir  $b = a + 10^{-(d-1)}$ , Mais nous n'avons pas envie d'expliquer cela, si ce n'est que si quelqu'un est assez bizarre pour spécifier  $d = 0$ , alors cela nous donnerait  $b = a + 101$ , et si quelqu'un disait quelque chose de plus raisonnable comme  $d = 3$ , alors  $b = a + 10^{-2}$ , de sorte que  $b - a = 10^{-2} > 10^{-3} = d$ . Et ainsi de suite. Au-delà de cela, nous laissons au lecteur le soin de comprendre pourquoi  $10^{-(d-1)}$  est une si bonne idée — à moins que l'instructeur ne lui assigne l'exercice correspondant...

Nous avons at3  $(1, \cos 1 - 1) \approx (1, -0,459698)$ , et une pente, que nous ne connaissons pas. La pente de la tangente est la valeur de la dérivée ; nous calculons donc

$$f(x) = -\sin x - 1$$

et évaluons  $f(1) \approx -1,84147$ . Notre tangente est donc  $y + 0,459698 = -1,84147(x - 1)$ .

Pour trouver la racine, définissez  $y = 0$  et résolvez pour  $x$ , en obtenant la nouvelle approximation  $b = c = 0,7503638678$ .

- Au début du deuxième passage dans la boucle, l'algorithme vérifie que les trois premiers chiffres de  $a = 1$  et  $b = 0,75038678$  diffèrent. Il attribue à  $a = 0,75038678$ , puis à  $b$  la racine de la droite tangente à  $\cos x - x$ . Nous connaissons déjà la dérivée  $f(x)$ , et évaluons  $f(0,75038678) \approx -1,6736325442$ . Notre droite tangente est donc  $y + 0,0000464559 = -1,6736325442(x - 0,75038678)$ . (Notez que la valeur  $y = 4,6 \times 10^{-5}$  est déjà très, très proche de 0.) Pour trouver la racine, définissez  $y = 0$  et résolvez pour  $x$ , obtenant la nouvelle approximation  $b = c = 0,7391128909$ .
- Au début du troisième passage, l'algorithme vérifie que  $a = 0,75038678$  et  $b = 0,7391128909$  diffèrent par leurs 3 premiers chiffres décimaux. Il attribue à  $a = 0,7391128909$ , puis à  $b$  la racine de la tangente à  $\cos x - x$ . En passant outre les détails, nous obtenons la nouvelle approximation  $b = c = 0,7390851334$ .
  - Au début du quatrième passage, l'algorithme remarque que les trois premiers chiffres décimaux de  $a = 0,7391128909$  et  $b = 0,7390851334$  concordent. La boucle while se termine et l'algorithme n'a plus qu'à renvoyer  $a \approx 0,7391128909$ .

Remarquez la rapidité avec laquelle la méthode de Newton a atteint son objectif ! Comment se compare-t-elle à la méthode de la bisection ? Rappelons que la méthode de la bisection divise l'intervalle par deux à chaque étape. Ainsi, selon le choix de  $a$  et  $b$ , nous aurions dû exécuter  $n$  étapes telles que  $2^{-n}(b - a) < 10^{-3}$  ou  $10^{-3}/|b-a| = 3\log_2 10 + \log_2(b-a) \approx 10 + \log_2(b-a)$ . La meilleure estimation que nous pouvons faire pour  $(a, b)$  en observant le graphique est probablement  $(1/2, 3/4)$ , donc  $\log_2(b-a) = \log_2 1/2 = -1$ . Par conséquent, la méthode de la bisection a besoin de  $n > 9$  pour approximer la racine au millième. L'utilisation de la méthode de Newton a nécessité moins d'un tiers du temps.

Code Sage. Sage ne dispose pas de commande permettant de tester immédiatement si deux nombres diffèrent par leurs 3 premiers chiffres décimaux ; nous devons donc trouver comment procéder par nous-mêmes. Ce n'est toutefois pas trop difficile ; rappelons que la commande `round()` arrondit un nombre au nombre de chiffres spécifié. Il suffit donc d'arrondir  $a$  et  $b$  à trois chiffres, puis de vérifier leur égalité.

Nous devons également indiquer à Sage de résoudre l'équation de la tangente à la courbe. Nous avons expliqué, page 83, comment construire l'équation d'une tangente ; nous ne le répéterons donc pas ici. La commande `solve()` nous fournira la racine, mais rappelons que `solve()` renvoie une liste de solutions sous forme d'équations. Nous devons donc extraire la solution en utilisant `[0]` pour obtenir le premier élément de la liste, et la méthode `.rhs()` pour obtenir le membre de droite de l'équation.

Ces considérations suggèrent le code Sage suivant :

---

3. Eh bien, environ. Nous devons recourir à des valeurs à virgule flottante, car dans ce cas, les valeurs exactes deviennent insignifiantes. rapide et ralentit énormément le calcul.

```
sage : def newtons_method(f, a, d, x=x) : f(x) = f df(x) = diff(f, x)
        b, c = a + 2,
        a # boucle jusqu'à ce que la
        précision souhaitée soit
        trouvée while round(a, d) != round(b, d) :

            a = c
            m = df(a) #
            trouver la racine de la tangente sol =
            solve(m*(x - a) + f(a), x) b = sol[0].rhs() c = b

        retourner un
```

Comment cela fonctionne-t-il sur l'exemple que nous avons spécifié précédemment ?

```
sage: newtons_method(cos(x) - x, 1, 3) ((cos(cos(1)/(sin(1) + 1) +
+ sin(1)/(sin(1) + 1)) + sin(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) + 1)))*sin(1) +
cos(1)*sin(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) + 1)) + cos(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) +
1))/((sin(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) + 1)) + 1)*péché(1) + péché(cos(1)/
(péché(1) + 1) + péché(1)/(péché(1) + 1)) + 1)
```

Oups ! Cela illustre bien ce que nous voulions dire par approximation de nos réponses. Modifions le code pour que l'affectation de `b` dans la boucle devienne

```
sage: b = round(sol[0].rhs(), d+2)
```

L'arrondi à `d+2` décimales devrait garantir une précision constante sur les `d` premières décimales et garantir une approximation, au lieu de l'expression symbolique complexe que nous avons élaborée précédemment. Nous utiliserons également une première approximation en virgule flottante, 1.

```
sage : newtons_method(cos(x) - x, 1., 3) 0.73911
```

Relation entre boucles définies et indéfinies. Il s'avère que les boucles définies ne sont en réalité qu'un type particulier de boucles indéfinies, et que toute boucle `for` peut être implémentée comme une boucle `while`.

Dans Sage, par exemple, nous pouvons passer par une liste `L` en utilisant une boucle `while` comme suit :

```
sage : i = 0 sage :
while i < len(L) : # faire quelque chose
    avec L[i] i = i + 1
```

Les collections non indexées comme les ensembles sont un peu plus complexes. Supposons que S soit un ensemble ; nous pouvons le copier (`copy()`) , en extraire des éléments (`pop()`) et les exploiter. Une fois terminé, S est toujours là.

```
sage : T = copy(S) sage :
while not len(T) == 0 : t = T.pop() # faire
    quelque chose
    avec t
```

Néanmoins, il est utile, pour quelqu'un qui lit un programme, de distinguer une boucle définie d'une boucle indéfinie. Dans le second cas, par exemple, il est tout à fait possible que les lignes indentées sous `while not len(T) == 0:` modifient T. Dans ce cas, la boucle n'est pas définie ; même une boucle `while` de ce type ne peut être définie que si la collection n'est pas modifiée.

#### Qu'est-ce qui pourrait mal se passer ?

Nous avons dit que les boucles `for` étaient « définies » car elles parcourront une collection finie bien spécifiée. Cela garantit leur terminaison. Nous avons vu que les boucles `while` fonctionnent différemment : elles continuent tant qu'une condition est vraie, donc elles peuvent ne jamais se terminer. Voici un exemple très simple (à éviter à la maison, les enfants !) :

```
sage : tant que Vrai : a = 1
```

Dans cette boucle, la condition est spécifiée comme étant vraie. Elle ne changera jamais. Rien dans la boucle ne change le fait que la condition est vraie, et la boucle `while` ne se terminera jamais. C'est un bon moyen de bloquer l'ordinateur, et si vous avez été assez maladrois pour ignorer notre avertissement (vraiment, n'essayez pas ça à la maison, les enfants !), vous devrez arrêter Sage en cliquant sur le bouton Arrêter (nuage), en sélectionnant le menu Action, puis en cliquant sur Interrompre (autre serveur), ou en maintenant la touche Ctrl enfoncee et en appuyant sur C (ligne de commande).

Ce n'est pas le seul endroit où cela peut se produire ; des conditions soigneusement préparées peuvent également mal tourner. La méthode de Newton peut conduire à une boucle infinie lorsque nous faisons ce qui suit (une dernière fois : n'essayez pas cela à la maison, les enfants !) :

```
sage : f(x) = x**3 - 2*x + 2 sage :
newtons_method(f(x), 0, 3)
```

---

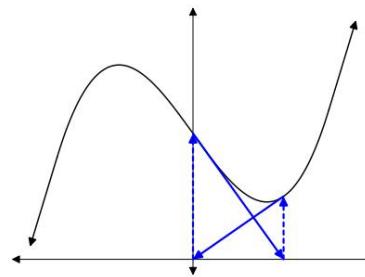
<sup>4</sup>Techniquement, on pourrait modifier T même dans une boucle `for` , mais la signification du mot rend moins probable qu'une personne Cela ferait l'affaire. Il est toutefois courant d'exécuter une boucle `while` sur une collection modifiée pendant la boucle.

Dans ce cas, la dérivée est  $f'(x) = 3x^2 - 2$ . La droite tangente à  $f$  en  $x = 0$  est  $y = -2x + 2$  ; sa racine est  $x = 1$ . La droite tangente à  $f$  en  $x = 1$  est  $y = (x - 1) + 1$  ; sa racine est  $x = 0$ . Nous sommes de retour au point de départ... oups !

Le lecteur qui examine cet exemple plus attentivement peut objecter qu'il n'est pas réaliste, dans la mesure où le choix de  $x = 0$  semble évidemment imprudent, étant quelque peu éloigné d'une racine à

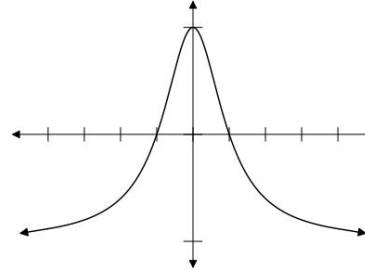
$$-\frac{3\sqrt[3]{81}2 + 3\sqrt[3]{81} + 3\sqrt[3]{9 - 57}}{273\sqrt[3]{9 - 57}} \approx -1,7693.$$

C'est encore plus évident à partir du graphique, qui illustre l'oscillation des approximations entre  $x = 0$  et  $x = 1$  :



Bien que cela soit vrai, il est également vrai que commencer à  $x = 0,5$  vous conduira également à cette même oscillation ; il faut simplement plus d'étapes avant que l'algorithme ne commence à osciller entre 0 et 1.

S'égarer. Un exemple de boucle infinie théorique se produit avec  $f(x) = 1/(1+x^2) - 1/2$ .



Supposons que nous commençons la méthode de Newton à  $x = 3$ , ce qui n'est pas trop loin de la racine à  $x = 1$ . Dans ce cas, Sage s'arrête soudainement avec l'erreur,

```
sage : newtons_method(f(x), 3., 3)
IndexError : index de liste hors limites
```

Si vous consultez les informations supplémentaires fournies par Sage, vous constaterez qu'il se plaint de la ligne `b = round(sol[0].rhs(), d+2)`. En résumé, IndexError signifie qu'une entrée n'existe pas dans la liste à l'index spécifié. L'index apparaît entre crochets : [0] dans ce cas, ce qui signifie que Sage ne trouve pas d'entrée dans `sol`. Cela suggère qu'il n'y a pas de solution.

Que s'est-il passé ? Si vous tracez les calculs manuellement, ou si vous placez un `print(a)` au début de la boucle `while`, vous verrez les approximations osciller brutalement entre des valeurs négatives et positives de plus en plus grandes :

`3, -3,66667, 8,58926, -149,80063, 840240,29866, -1,48303 × 1017, 8,15439 × 1050`

Cette erreur est due au fait que nous travaillons avec des approximations, et les nombres deviennent si grands que l'ordinateur décide qu'il n'y a pas de solution et `solve()` abandonne, renvoyant une liste vide. Essayez de tracer les tangentes sur le graphique pour voir comment cela se produit.

Le lecteur pourrait objecter que  $x = 3$  ne semble pas être un point particulièrement convaincant pour démarrer la méthode de Newton. Certes, mais  $x = 0,09$  est moins inquiétant, et son effet sera similaire.

Un phénomène similaire peut se produire dans la méthode de Newton lorsqu'on vise une racine, mais aboutit à une autre. Ce texte n'étant pas un ouvrage d'analyse numérique, nous n'entrerons pas dans les détails, ni dans d'autres difficultés liées à la méthode de Newton ; nous encourageons le lecteur intéressé à consulter un ouvrage sur ce sujet.

### Division des entiers gaussiens

Nous nous tournons vers un anneau intéressant appelé les entiers gaussiens, notés [i] en abrégé. Les entiers gaussiens ont la forme  $a+bi$ , où  $a$  et  $b$  sont des entiers. Cela diffère des nombres complexes, qui ont la forme superficiellement similaire  $a+bi$ , à la différence que pour les nombres complexes  $a$  et  $b$  sont des nombres réels. En bref, • 2

+  $3i$  est à la fois complexe et un entier gaussien, tandis que  $\bullet 2 + i/3$  et  $2 + i\sqrt{3}$  sont complexes, mais pas des entiers gaussiens.

L'addition, la soustraction et la multiplication des entiers gaussiens sont identiques à celles des nombres

$$\begin{aligned} \text{complexes : } & (a + bi) + (c + di) = (a + c) + \\ & (b + d)i \\ & (a + bi) - (c + di) = (a - c) + (b - d)i \\ & (a + bi)(c + di) = (ac - bd) + (ad + bc)i. \end{aligned}$$

Pour la division, cependant, nous ne voulons pas passer des entiers gaussiens aux nombres complexes, comme Sage pourrait nous le donner par défaut :

sage :  $(2 + 3i) / (1 - i)$   
 $5/2i - 1/2$

Nous souhaiterions plutôt un moyen de calculer un quotient et un reste, à l'instar des opérateurs // et % proposés par Sage pour les entiers. Plus précisément, nous souhaiterions un algorithme qui se comporte de manière similaire à la division d'entiers :

**THÉORÈME DE DIVISION DES ENTIERS.** Étant donnés un entier  $n$ , appelé dividende, et un entier  $d$  non nul, appelé diviseur, on peut trouver un entier  $q$ , appelé quotient, et un entier  $r$  non négatif, appelé le reste, tel que

$$n = qd + r \quad \text{et} \quad r < |d|.$$

Rappelons que la valeur absolue d'un nombre réel nous donne une idée de sa taille ; l'analogie pour les nombres complexes est la norme  $z$ . Traduit en entiers gaussiens, le théorème deviendrait

**THÉORÈME DE DIVISION DES ENTIERS GAUSSIENS .** Étant donnés un entier gaussien  $z$ , appelé dividende, et un entier gaussien non nul  $d$ , appelé diviseur, on peut trouver un entier gaussien  $q$ , appelé quotient, et un autre entier gaussien  $r = qd$  , appelé le reste, tel que

$$+ r. \quad \text{et} \quad r < d.$$

Ce théorème ne dit rien sur le fait que le reste soit non négatif, car « négatif » ne l'est pas. cela a vraiment du sens pour les entiers gaussiens.

Le théorème de division des entiers gaussiens est vrai, et nous allons le prouver en construisant un algorithme qui fait ce qu'il prétend. Mais comment construire un tel algorithme ? La division d'entiers est une soustraction répétée, et vous pouvez créer un algorithme de division parfaitement correct de la manière suivante :

```

algorithme divide_integers
entrées
    • n, d    tels que d = 0 sorties
    • q    et
          r    tels que n = qd + r et r < |d|
soit r = |n|, q = 0 si n
et d ont le même signe soit s = 1
sinon
soit
s = -1 tant
que r < 0 ou r ≥ |d| ajoute
s à q
remplace r par r - sd
renvoie q, r

```

Cet algorithme utilise s pour « avancer » le quotient dans la bonne direction vers n. Par exemple, si nous divisons  $-17$  par  $5$ , alors r et q prennent les valeurs suivantes au début de chaque boucle (q diminue car  $s = -1$ ) :

r	-17	-12	-7	-2	3		
q	0	-1	-2	-3	-4		

Une fois que  $r = 3$  et  $q = -4$ , l'algorithme remarque que  $r < |d|$ , donc la boucle while se termine, nous donnant l'expression correcte

$$-17 = qd + r = (-4) \times 5 + 3.$$

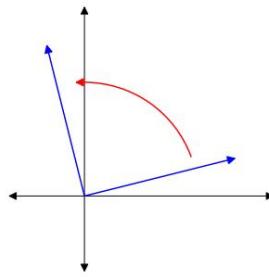
Il est logique que la division des entiers gaussiens fonctionne de la même manière. Notre stratégie générale sera donc la suivante :

```

soit r = z, q = 0
tant que r ≥ d
    « augmenter q » en utilisant une valeur s
    appropriée remplacer
    r par r - sd renvoyer q, r

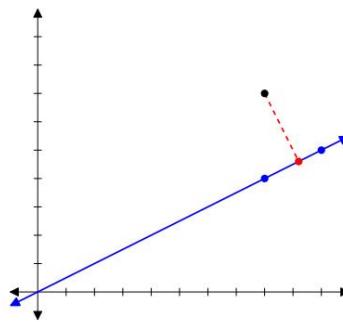
```

Il nous faut maintenant comprendre ce que signifie « augmenter q ». La difficulté réside dans le fait que l'on peut « augmenter » q dans deux directions : la direction réelle et la direction imaginaire. Comme vous l'avez vu dans l'introduction au plan complexe, page 77, multiplier un nombre complexe par  $i$  le fait pivoter de  $90^\circ$  dans le sens inverse des aiguilles d'une montre.



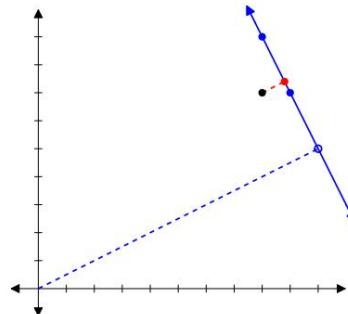
Il est donc possible d'« augmenter  $q$  » de deux manières différentes.

Nous allons résoudre ce problème en deux étapes : d'abord, nous minimisons la distance en multipliant par des entiers ; ensuite, nous minimisons la distance en multipliant par des multiples entiers de  $i$ . Par exemple, supposons que nous divisons  $8+7i$  par  $2+i$ . Le diviseur se trouve sur la droite de pente  $1/2$  ; nous progressons donc le long de cette droite dans la direction appropriée jusqu'à ce que tout autre pas augmente la distance. D'après la géométrie, nous savons que cela signifie se rapprocher le plus possible de l'altitude tombée de  $(8, 7)$  à la ligne  $y = x/2$  :



Malheureusement, le point le plus proche est  $(91/5, 43/5)$  (en rouge), qui correspond à un nombre complexe, mais pas à un entier gaussien. L'entier gaussien le plus proche est le point bleu situé au-delà du point rouge ; la seule façon de le vérifier est de comparer la norme de chaque point à mesure que l'on se déplace le long de la droite.

L'étape suivante consiste à partir de notre entier gaussien minimal le long d'une droite perpendiculaire à la droite bleue, correspondant à la multiplication répétée de  $d$  par  $i$ . Nous progressons à nouveau le long de la droite jusqu'à ce que tout autre pas augmente la norme.



Le point le plus proche sur la droite contenant des valeurs entières est  $(9, 7)$ , ce qui correspond à  $9+7i$ . Nous avons obtenu cette valeur après avoir multiplié par 5, puis par  $i$  :

$$(2 + i)(5 + i) = 10 + 2i + 5i - 1 = 9 + 7i.$$

Le reste est  $-1$ , dont la norme est plus petite que celle du diviseur.

Écrire ceci sous forme de pseudo-code nous donne ce qui suit.

```

algorithme divide_gaussian_integers entrées
• z, d
    [i] tels que d = 0 sorties • q, r    [i]
    tels que z
        = qd + r et r < d

    faire soit r = z, q = 0 si
        r - d < r soit s = 1
        sinon soit
            s =
            -1 tant que
            r - sd < r ajouter s à q
            remplacer r
            par r - sd si r - id < r
            soit s = i sinon soit s
            = -i tant
            que
            r - sd < r
            ajouter s à q remplacer r
            par r - sd
            retourner q, r

```

Comme prévu, cet algorithme utilise une instruction if pour vérifier d'abord s'il est préférable d'avancer ou de reculer d'un facteur réel. Il choisit ensuite  $s$  comme étant égal à 1 ou  $-1$ , selon la valeur la plus appropriée. La boucle while avance ensuite dans la direction appropriée, diminuant la norme jusqu'à ce que des étapes supplémentaires l'augmentent. Nous ne pouvons pas nous fier au test  $r \geq d$  comme pour les entiers, car une minimisation supplémentaire pourrait être nécessaire après la première étape, car  $r > d$  reste inchangé.

C'est pourquoi la boucle vérifie plutôt  $r - sd < r$ ; en gros, cela se traduit par « avancer davantage réduit la norme de  $r$  ».

L'algorithme utilise ensuite une seconde instruction if pour vérifier s'il est préférable d'avancer ou de reculer d'un facteur imaginaire. Il choisit  $s$  comme étant  $i$  ou  $-i$ , selon la valeur la plus appropriée. La boucle while avance ensuite dans la direction appropriée, diminuant la norme jusqu'à ce que des étapes supplémentaires l'augmentent. Nous pourrions effectivement tester  $r \geq d$  ici, mais par souci de cohérence, nous avons utilisé la même structure de base.

Cela se traduit par le code Sage de la figure 2. Comment fonctionne-t-il sur certains exemples ?

```

sage : diviser_entiers_gaussiens(4 + I, 1 - I)
(2*I + 1, 1)
sage : diviser_entiers_gaussiens(8 + 7*I, 2 + I)
(5 + 1, -1)

```

```

sage : def divide_gaussian_integers(z, d) : r, q = z, 0 # dans quel
        sens réel faut-il
        marcher ? si norm(r - d) < norm(r) :

            s = 1
        autre:
            s = -1
        # boucle pour passer
        à l'étape suivante tant que norm(r - s*d) < norm(r) :
            q = q + sr = r -
            s*d
        # quel chemin imaginaire parcourir ? si norm(r -
        l*d) < norm(r) :
            s = je
        autre:
            s = -l
        # boucle pour passer
        à l'étape suivante tant que norm(r - s*d) < norm(r) :
            q = q + s
            r = r - s*d
        retour q, r

```

FIGURE 2. Code Sage pour la division des entiers gaussiens

Nous avons déjà vérifié le deuxième calcul en l'examinant plus tôt ; le premier n'est pas difficile à faire soi-même.

Une retraite bien ordonnée. Avec la méthode de Newton, nous avons rencontré une boucle infinie avec des combinaisons malheureuses d'une fonction  $f$  et d'une estimation initiale  $a$ . Est-il possible de trouver deux entiers gaussiens tels que les boucles while de `divide_gaussian_integers()` deviennent infinies ?

Pas dans ce cas. La différence réside dans le fait que les boucles sont construites de manière à décrire la condition à l'aide d'un entier positif qui décroît après chaque passage réussi. Dans les deux boucles, cette valeur est  $z - qd$ . Pourquoi ? L'algorithme initialise  $r = z$ , puis rapproche  $q$  de  $z$  si et seulement si cela réduit la distance entre  $z$  et  $qd$  ; autrement dit,  $z - qd$ . La norme est toujours un entier, et les entiers positifs bénéficient d'une propriété appelée « propriété de bon ordre » :

Chaque sous-ensemble non vide de possède un plus petit élément.

Si la boucle était infinie, on pourrait considérer que toutes les valeurs de  $z - qd$  ont un sous-ensemble de . L'algorithme effectue la boucle si cette distance diminue, et la propriété de bon ordre garantit que cela ne peut se produire qu'un nombre fini de fois, ce qui contredit l'hypothèse d'une boucle infinie.

Si possible, il est judicieux, lors de l'utilisation de boucles while, de les formuler de manière à exploiter leur propriété de bon ordre, garantissant ainsi leur terminaison. Cette méthode est possible dans presque toutes les circonstances, même si les conséquences peuvent être indésirables.

Une façon de procéder avec la méthode de Newton, par exemple, est de définir une variable `max_iterations` sur

Un grand nombre, diminuez-le d'une unité à chaque passage dans la boucle, puis reformulez la condition de la boucle while pour qu'elle s'arrête dès que `max_iterations` atteint 0. Vous pouvez même choisir de lever une exception. Nous concluons cette section avec une méthode `terminating_newtons_method()` modifiée qui illustre cette technique.

```
sage : def méthode_terminante_de_newtons(f, a, d, x=x) :
    f(x) = f df(x) =
    diff(f, x) # a + 2 devrait être trop
    loin pour une terminaison prématuée b, c = a + 2, a

    max_iterations = 100
    # boucle pour la précision souhaitée, jusqu'au seuil tant que max_iterations > 0 et
    round(a, d) != round(b, d) :
        a = c
        m = df(a) sol =
        résoudre(m*(x - a) + f(a), x) b = sol[0].rhs()

        c = b
        max_iterations = max_iterations - 1
        # erreur ? signaler si oui si max_iterations
        == 0 :
            raise ValueError('Trop d'itérations : ') + 'essayer une valeur initiale
            différente' renvoie un
```

Essayez avec l'exemple non terminal d'avant ( $f = x$  se comporte  $x^3 - 2x + 2$ ,  $a = 0$ ) et voyez comment cela beaucoup mieux).

### Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Dans Sage, le mot-clé `while` implémente une boucle indéfinie.
2. Une boucle `while` s'arrête dès que la condition devient fausse, même si elle n'a pas terminé toutes les déclarations en retrait en dessous.
3. La manière la plus intelligente de parcourir une collection est d'utiliser une boucle `while`.
4. Une boucle indéfinie est un type particulier de boucle définie.
5. Il est impossible de créer une boucle infinie en utilisant une boucle indéfinie.
6. Il est impossible d'empêcher toute boucle indéfinie de devenir une boucle infinie.
7. La méthode de Newton trouve toutes les racines que la méthode de bisection trouvera.
8. La méthode de Newton trouve certaines racines que la méthode de la bisection ne trouvera pas.
9. La division gaussienne est plus compliquée que la division complexe car elle nécessite un quotient et un reste.
10. La propriété de bon ordre stipule que chaque sous-ensemble des nombres naturels a un minimum élément.

Choix multiple.

1. Une boucle qui itère sur une collection spécifique est :

A. boucle définie B.

boucle indéfinie C. boucle

infinie D. boucle

spécifique 2. Une

boucle qui se répète un nombre spécifique de fois est :

A. boucle définie B.

boucle indéfinie C. boucle

infinie D. boucle

spécifique 3. Une

boucle qui se répète selon qu'une condition reste vraie ou fausse est : A. boucle définie B. boucle indéfinie

C. boucle infinie D.

boucle spécifique 4. La

principale raison

d'utiliser une boucle

indéfinie au lieu d'une boucle définie est : A. Il est plus facile de garantir la terminaison d'une boucle

indéfinie.

B. Il est plus difficile de garantir la terminaison d'une boucle indéfinie.

C. Pour décider de se terminer, la boucle doit tester une condition booléenne.

D. Les boucles définies ne sont qu'un type particulier de boucle indéfinie, il est donc plus cohérent d'utiliser exclusivement des boucles indéfinies.

5. Laquelle des propositions suivantes caractérise le mieux l'idée derrière la méthode de Newton ?

A. Utilisez la ligne tangente pour trouver une nouvelle approximation, espérons-le plus proche de la racine correcte.

B. Utilisez la ligne tangente pour trouver une nouvelle approximation, certainement plus proche de la racine correcte.

C. Divisez l'intervalle en deux à plusieurs reprises jusqu'à ce que les points finaux correspondent aux 4 premiers chiffres décimaux.

D. Tracez le long de la courbe avec une précision extrême, en zoomant et en vous déplaçant d'avant en arrière jusqu'à obtenir le résultat correct.

6. Pour laquelle des fonctions suivantes serait-il préférable d'utiliser la méthode de la bissectrice ?

tion pour trouver une racine, plutôt que la méthode de Newton ?

A.  $|x + 3|$  B.

$\sin(x + \pi)$

C.  $(x - 3)^2$

D.  $\ln(x - 3)$

7. Dans quelles conditions la méthode de Newton peut-elle ne pas trouver de racine ?

A. la fonction est discontinue B. la

fonction est non différentiable C. le point

de départ s'éloigne de la racine D. tout ce qui précède

8. Laquelle des interprétations géométriques suivantes est correcte de ce qui se passe lorsque vous mul-

Est-ce un nombre complexe par  $-i$  ? (Notez qu'il est négatif !)

A. Le nouveau nombre correspond à une rotation de  $90^\circ$  dans le sens des aiguilles d'une montre dans le plan complexe.

B. Le nouveau nombre correspond à une rotation de  $90^\circ$  dans le sens inverse des aiguilles d'une montre dans le plan complexe.

C. Le nouveau nombre correspond à une rotation de  $180^\circ$  dans le plan complexe.

D. Le nouveau numéro est dans la même direction que le numéro d'origine, seulement plus long.

9. Pour laquelle des paires de nombres suivantes la division gaussienne devrait-elle échouer ? (Le dividende vient en premier, le

diviseur en second)

A.  $4 + i, 1 + i$

B.  $0, 1 + i C.$

1 + i, 0 D.

aucune des réponses ci-dessus

10. La raison pour laquelle nous avons généré une ValueError dans la dernière version de la méthode de Newton est que : A. Nous considérons que la fonction était une mauvaise valeur, car elle n'est pas continue en a.  
 B. Nous considérons le point de départ comme une mauvaise valeur, car la fonction n'est pas différentiable là.  
 C. Les valeurs sont devenues trop importantes et Sage n'a pas pu trouver de solution.  
 D. Nous considérons que le point de départ est une mauvaise valeur, car il semble pris dans une boucle infinie.

Réponse courte.

1. Considérez la procédure Sage suivante :

```
sage : déf aw(n) : t = 0
        i = 1
        tant que i <= n :
            t = t + i**2
            i = i + 1
        retour t
```

(a) Calculez  $aw(4)$  à la main, en montrant votre travail. (b)

Réécrivez  $aw(n)$  comme une expression mathématique sur une seule ligne, en utilisant la notation de sommation. (c) Traduisez  $aw(n)$  en une nouvelle procédure  $af(n)$ , qui fait la même chose, mais utilise un pour boucle à la place.

(d) Les boucles while et for sont inefficaces. Comment serait-il plus judicieux de les implémenter ?  
 ment cette procédure ?

2. La suite de Syracuse est définie de la manière suivante. Soit  $S_1$  , et

$$S_{i+1} = \begin{cases} S_i/2, & \text{Si } S_i \text{ est pair ;} \\ 3S_i + 1, & \text{Si } S_i \text{ est impair.} \end{cases}$$

La suite se termine lorsque  $S_i = 1$ . On pense que la suite se termine par  $S_i = 1$  pour chaque valeur de  $S_1$  , mais personne ne le sait réellement. Calculez manuellement la suite pour plusieurs valeurs différentes de  $S_1$  et remarquez qu'elle semble toujours atteindre  $S_i = 1$ .

3. La division des polynômes univariés  $f$  par  $g$  fonctionne de la manière suivante (ici,  $\deg(r)$  signifie « degré de  $r$  » et  $lc(r)$  signifie « coefficient dominant de  $r$  ») :

• soit  $r = f$  ,  $q = 0$  • tant que

$\deg(r) \geq \deg(g)$   $\deg(r)$

$-\deg(g)$  – soit  $t = x$

$$, c = lc(r)/lc(g)$$

– remplacer  $r$  par  $r - ctg$

– ajouter  $ct$  à

$q$  • renvoyer  $q$ ,  $r$

- (a) Choisissez 5 paires de polynômes et appliquez-leur l'algorithme de division. Vous pouvez utiliser Sage pour accélérer les soustractions, mais avec la commande Division de Sage, vous ne pourrez pas résoudre au moins un des problèmes suivants. (b) Pensez-vous que cet algorithme exploite la propriété de bon ordre ? Pourquoi ?

(c) Existe-t-il un modèle quant au nombre de fois où l'algorithme parcourt la boucle ? (d) La boucle pourrait-elle être écrite comme une boucle for ? Si oui, comment ? Sinon, pourquoi ?

4. Montrez comment ces boucles for peuvent être réécrites en utilisant une boucle while dans chaque cas :

(un)

pour k dans la plage(num) :  
 <corps> # éventuellement en fonction de k

(b)

pour elem dans L : # L est une collection (liste/tuple/ensemble) #  
 <corps> dépendant de elem

5. Est-il possible de réécrire chaque boucle while en boucle for ? Expliquez !

Programmation.

- Dans la réponse courte n° 1, vous avez travaillé avec la séquence Syracuse. Écrivez un pseudo-code expliquant comment l'implémenter sous forme de programme. Ce pseudo-code doit accepter un argument pour S1 , définir s = S1 , appliquer la formule à s pour trouver sa valeur suivante et se terminer lorsque s = 1. Implémentez votre pseudo-code. (Vous pouvez tester si s est pair en utilisant l'opérateur % de Sage , car un nombre est pair lorsque le reste après division par 2 est 0.)
- L'algorithme traditionnel de division entière (p. 183) trouve le reste non négatif le plus proche. Dans de nombreux cas, autoriser un reste négatif laisserait une distance plus petite entre n et d. Par exemple, la division traditionnelle aurait

$$-17 = -4 \times 5 + 3$$

tandis que permettre des restes négatifs aurait

$$-17 = -3 \times 5 + (-2).$$

Il est possible de formuler un algorithme qui effectue cette division du « plus petit reste ». (a) Essayez quelques exemples où vous utilisez des soustractions répétées pour effectuer cette division du « plus petit reste ». division.

(b) Comment apprendriez-vous à un ami à effectuer cette division ? Réfléchissez aux instructions précises.

(c) Implémentez votre algorithme en tant que programme Sage et testez-le.

- Lors de la mise en œuvre de la méthode de bisection, nous avons utilisé un logarithme pour déterminer le nombre d'étapes requis. C'était la seule façon d'utiliser une boucle définie. Il est possible de reformuler l'algorithme avec une boucle indéfinie, similaire à la méthode de Newton : terminer l'algorithme dès que a et b correspondent au nombre de chiffres spécifié. (a) Faites ceci. Réécrivez le pseudo-code, puis le code Sage pour voir cela en action. (b) Quelle méthode préférez-vous et pourquoi ? (c) Quelle méthode pensez-vous être plus facile à lire, à comprendre et/ou à modifier pour quelqu'un d'autre ? Encore une fois, pourquoi ?

- Dans la réponse courte n° 3, vous avez travaillé sur la division de polynômes univariés. Écrivez le pseudo-code complet, c'est-à-dire nommez l'algorithme, spécifiez précisément les entrées et les sorties. Implémentez-le ensuite dans un programme Sage et testez-le sur les mêmes exemples que dans ce problème.

5. Un entier positif  $n$  est premier si aucun entier de 2 à  $n$  ne divise  $n$ .

(a) Écrivez une procédure Sage qui accepte une valeur de  $n$  en entrée et trouve le plus petit diviseur positif non trivial de  $n$ .<sup>5</sup> Si le nombre est premier, la procédure doit générer une erreur ValueError. (b)

Écrivez une autre procédure Sage qui accepte un entier positif  $k$  en entrée et renvoie une liste de tous les nombres premiers inférieurs ou égaux à  $k$ . Demandez à la procédure d'appeler la réponse à la partie (a).

(c) La conjecture de Goldbach affirme que tout nombre pair est la somme de deux nombres premiers. Bien qu'elle ait été vérifiée jusqu'à des valeurs inimaginables, personne n'a encore prouvé que la conjecture de Goldbach est vraie pour tous les entiers positifs. Écrivez une procédure qui accepte un entier positif  $m$  en entrée, génère une erreur ValueError si  $m$  n'est pas pair ; sinon, elle trouve deux nombres premiers dont la somme est égale à  $m$ . Demandez à la procédure d'invoquer la réponse à la partie (b).

6. Le plus grand diviseur commun de deux entiers est le plus grand entier qui les divise tous les deux.

Un algorithme permettant de calculer le plus grand diviseur commun était déjà connu d'Euclide d'Alexandrie il y a plus de 2000 ans :

```

algorithme pgcd
entrées
    • a, b
sorties •
    le plus grand commun diviseur de a et b

soit m = |a|, n = |b| tant
que n = 0 soit
    d = n
    remplacez n par le reste de la division de m par n remplacez
        m par d
    retour m

```

(a) Implémentez ceci en tant que code Sage et vérifiez qu'il donne les plus grands diviseurs communs corrects pour plusieurs grands nombres.

(b) Choisissez une petite paire d'entiers non nuls et parcourez l'algorithme, étape par étape, montrant comment il arrive au pgcd.

7. L'algorithme d'Euclide étendu ne trouve pas seulement le pgcd de deux entiers non nuls  $a$  et  $b$ , il trouve les nombres  $c$  et  $d$  tels que

$$\text{pgcd}(a, b) = ac + bd.$$

Par exemple,  $\text{pgcd}(4, 6) = -1 \times 4 + 1 \times 6$ . Nous pouvons le décrire comme suit :

---

<sup>5</sup>Un diviseur positif n'est pas trivial s'il n'est pas égal à 1.

```
algorithme algorithme_euclidien_étendu
entrées
    • a, b
sorties •
    c, d    telles que pgcd(a, b) = ac + bd

soit m = |a|, n = |b| soit
    s = 0, c = 1, t = 1, d = 0 tant
    que n = 0
        soit q le quotient de la division de m par n
        soit r le reste de la division de m par n soit m
        = n et n = r soit w = c,
            alors c = s et s = w - qs soit w = d, alors
            d = t et t = w - qt renvoie c, d
```

Implémentez ceci en tant que code Sage et vérifiez qu'il donne les valeurs correctes de  $c$  et  $d$  pour plusieurs grandes valeurs de  $a$  et  $b$  ; c'est-à-dire, vérifiez que les valeurs  $c$  et  $d$  qu'il renvoie satisfont  $ac + bd = \gcd(a, b)$ .

## CHAPITRE 9

## Se répéter de manière inductive

Une autre forme de répétition est appelée récursivité. Ce terme signifie littéralement « répéter ». car l'idée est qu'un algorithme résout un problème en le décomposant en cas plus simples, en invoquant sur ces cas plus petits, puis de reconstituer les résultats. Ces cas plus petits diviser leurs dossiers en dossiers encore plus petits. Cela peut paraître un peu risqué : qu'est-ce qui nous empêche de frapper une chaîne infinie ?

Lors de l'organisation d'une récursivité, nous essayons de la configurer de telle manière que cette stratégie crée une chaîne des problèmes qui mènent à l'élément le plus petit possible. Si nous pouvons les relier aux nombres naturels, alors c'est encore mieux, car les nombres naturels bénéficient d'une propriété appelée propriété de bon ordre :

Chaque sous-ensemble non vide de possède un plus petit élément.

(Si cela vous semble familier, c'est parce que nous y avons fait référence à la page 186 du chapitre 8.)

## Récursivité

Un exemple classique. De nombreux problèmes se traduisent naturellement par la récursivité. Un exemple est le triangle de Pascal, qui ressemble à ceci :

$$\begin{array}{ccccccc}
 & & & 1 & & & \\
 & & 1 & & 1 & & \\
 & 1 & & 2 & & 1 & \\
 1 & & 3 & & 3 & & 1 \\
 1 & & 4 & & 6 & & 4 & & 1 \\
 & \vdots & & \vdots & & \ddots & \\
 \end{array}$$

Les lignes de ce triangle apparaissent à de nombreux endroits, comme dans le développement des binômes :

$$\begin{aligned}
 (x+1)^0 &= 1 \\
 (x+1)^1 &= 1x + 1 \\
 (x+1)^2 &= 1x^2 + 2x + 1 \\
 (x+1)^3 &= 1x^3 + 3x^2 + 3x + 1 \\
 (x+1)^4 &= 1x^4 + 4x^3 + 6x^2 + 4x + 1 \\
 &\vdots && \vdots && \ddots
 \end{aligned}$$

Prenez un moment pour essayer de déduire le motif du triangle.

Vous le voyez ? Sinon, prenez un autre moment pour essayer de le comprendre.

Vous avez sans doute remarqué que chaque ligne est définie de la manière suivante : la ligne du haut est la ligne 1. la rangée suivante, la rangée 2, et ainsi de suite.

- Le premier et le dernier élément de chaque ligne sont tous deux 1.
- Les éléments restants satisfont  $q_j = p_{j-1} + p_j$ , où  
 $q_j$  est l'élément de l'entrée  $j$  sur la ligne  $i$ , et

–  $p_{j-1}$  et  $p_j$  sont les éléments des entrées  $j - 1$  et  $j$  sur la ligne  $i - 1$ .

Pour connaître les entrées de la ligne  $i$ , nous devons d'abord connaître les entrées de la ligne  $i - 1$ . Nous pouvons parcourir ces entrées pour obtenir les entrées suivantes. Nous savons combien il y a d'entrées dans cette ligne :  $i - 1$ , mais nous pouvons également nous renseigner à partir de la ligne elle-même.<sup>1</sup> Nous pouvons donc appliquer une boucle définie. Cela nous donne le pseudo-code récursif suivant pour calculer la ligne  $i$  :

```

algorithme pascals_row
entrées
    • i      , la rangée souhaitée du triangle de Pascal
sorties •
    la séquence de nombres de la ligne i du triangle de Pascal

faire si i =
    1 résultat =
    [1] sinon si i
        = 2 résultat =
        [1,1]
        sinon prev = pascals_row(i -
            1) résultat =
            [1] pour j   (2, 3,...,i - 1)
                ajouter prevj-1 + prevj au résultat
            ajouter 1 au résultat
            renvoyer le résultat

```

Cela se traduit par le code Sage suivant :

```

sage : def pascals_row(i) :
    résultat = [1]
    elif i == 2:
        résultat = [1, 1]
    autre:
        # calculer d'abord la ligne précédente prev =
        pascals_row(i - 1) # cette ligne commence
        par 1...
        résultat = [1]
        # ...ajoute deux au-dessus du suivant dans cette ligne...
        pour j dans la plage(1, i - 1) :
            result.append(prev[j-1] + prev[j])
        # ... et se termine par 1
        résultat.append(1)
    résultat de retour

```

---

<sup>1</sup>Dans le pseudo-code, nous aurions pu écrire « |prev| » pour l'indiquer. Dans le code Sage, nous aurions simplement pu écrire prev.len().

Nous pouvons le vérifier sur les lignes que nous avons vues ci-dessus, et en générer davantage que vous pouvez vérifier à la main :

```
sage: pascals_row(1) [1]
sage: pascals_row(5) [1, 4,
6, 4, 1] sage:
pascals_row(10) [1, 9, 36, 84,
126, 126, 84, 36, 9, 1]
```

Un exemple si classique qu'il en paraît médiéval. L'exemple suivant est l'un des problèmes fondamentaux des mathématiques.

À quelle vitesse une population de lapins va-t-elle croître ?



Tu dois demander?!?

Vous pensez qu'on plaisante, mais non.<sup>2</sup> Comme vous pouvez le deviner, ces lapins ne sont pas des lapins ordinaires ; ils obéissent à des lois de reproduction particulières : • Au

commencement, il y avait un couple, mais ils étaient immatures. • Chaque couple de lapins met une saison pour arriver à maturité. • Une fois mature, chaque couple produit un nouveau couple chaque saison. • Les lapins sont immortels. Ils n'ont pas de prédateurs et tirent leur énergie du soleil, ce qui fait qu'ils ne manquent jamais de nourriture. Étonnamment, ils ne se lassent jamais des nouveaux enfants qui les réveillent la nuit, rampent sur eux et, généralement, les gênent dans des activités plus nobles, comme la contemplation de la forme du lapin idéal.<sup>3</sup>

La formule du nombre de paires de lapins dans une saison est définie par le nombre des deux saisons précédentes :

$$b_n = b_{n-1} + b_{n-2} .$$

C'est un parfait exemple de récursivité. Il est naturel d'essayer de l'implémenter de manière récursive avec une procédure acceptant un entier pour le nombre de saisons écoulées. Si c'est 1 ou 2, la réponse est simple : 1. Sinon, nous calculons le nombre de lapins des deux saisons précédentes.

<sup>2</sup>Au moins un journal est dédié à ces lapins.

<sup>3</sup>Référence gratuite (et non pertinente) à la philosophie platonicienne incluse sans frais supplémentaires.

```

algorithme funny_bunnies
entrées
    • n      , le nombre de saisons
sorties •
    le nombre de couples de lapins après n saisons

faire si n = 1 ou n = 2
    soit résultat =
        1
    sinon soit résultat = funny_bunnies(n - 1) + funny_bunnies(n - 2)
retourner résultat

```

Pour traduire cela en code Sage, gardez à l'esprit que range() n'itère pas sur la dernière valeur, nous avons donc besoin que sa limite soit  $n-1$  :

```

sage : def funny_bunnies(n) : si n == 1 ou n
    == 2 :
        résultat = 1
    autre:
        résultat = funny_bunnies(n-1) + funny_bunnies(n-2)
    résultat de retour
sage : [funny_bunnies(i) pour i dans la plage(2,10)] [2, 3, 5, 8, 13, 21, 34]

```

C'est une liste de nombres très intéressante. Elle est si fascinante qu'elle porte le nom d'un mathématicien qui l'a décrite au Moyen Âge, Léonard de Pise, plus connu sous le nom de Fibonacci<sup>4</sup>.

<sup>4</sup>Fibonacci a vécu à une époque connue sous le nom de Haut Moyen Âge et ne se souciait pas tant des lapins que du fait que les gens utilisaient encore I, V, X, L, C, D et M pour écrire des nombres et faire de l'arithmétique — des chiffres romains, bien sûr.

Il réalisa que c'était très inefficace et qu'il serait préférable d'utiliser la méthode hindoue-arabe pour écrire les nombres : 0, 1, 9, etc. Le problème était que la plupart des gens étaient habitués à travailler avec une calculatrice à l'ancienne, appelée boulier 2, ..., et avaient formés à l'utilisation d'un boulier avec des chiffres romains. À l'époque, comme aujourd'hui, les gens étaient réticents à abandonner cette béquille informatique.

Pour contourner ce problème, il écrivit un livre intitulé *Liber Abaci* (« Le Livre de l'abaque »). Son objectif principal était de montrer comment résoudre les mêmes problèmes avec des nombres hindous-arabes sans utiliser de boulier, et de travailler de manière bien plus efficace. Travailler plus vite permettait de réaliser davantage d'affaires. Les lapins platoniciens étaient l'un des exemples qu'il utilisait dans son livre. C'était probablement la première fois que quelqu'un utilisait un lapin rigolo pour fabriquer des objets.

argent.

Oh, on n'a pas fini de vous faire râler, loin de là. Vous trouverez peut-être que *Liber Abaci* est un titre un peu fade — certainement pas aussi accrocheur que le slogan « Penser différemment », même si Fibonacci aurait eu de meilleures raisons d'utiliser ce slogan que quiconque l'a popularisé aujourd'hui. Mais à l'époque, on s'intéressait davantage au fond qu'à la forme. Fibonacci a donc réussi à sévir les gens des chiffres romains et à associer son nom à une suite de nombres qui, ironiquement, avait été étudiée par des mathématiciens indiens (hindous) bien avant que Leo ne soit lui-même un nombre de Fibonacci.

Ce n'est pas mal quand même, pour une bande de personnes d'âge moyen intellectuellement en retard.

(Et, oui, nous reconnaissons l'ironie de faire référence à un boulier comme une « béquille informatique » dans un texte dédié à faire des problèmes de mathématiques avec le système d'algèbre informatique Sage, mais cette note de bas de page est conçue pour être riche en ironie.)

Appelons-la la suite de Fibonacci. On peut la décrire ainsi :<sup>5</sup>

$$F_{\text{suivant}} = F_{\text{total}} + F_{\text{mature.}}$$

Cela signifie que « le nombre de lapins la saison prochaine est la somme du nombre total de lapins et du nombre de lapins adultes ». On pourrait également dire que le nombre total de lapins est le nombre « actuel » de lapins, tandis que le nombre de lapins adultes est le nombre de lapins que nous avions la saison dernière. Cela signifie que « le nombre de lapins la saison prochaine est la somme du nombre de lapins de cette saison et du nombre de lapins de la saison dernière ». On pourrait donc écrire :

$$F_{\text{suivant}} = F_{\text{courant}} + F_{\text{précédent.}}$$

Un exemple que vous ne pouvez pas réellement programmer. Un autre exemple est la technique de preuve appelée induction. Le principe de l'induction est que :

- si vous

pouvez prouver : –

- qu'une propriété s'applique à 1, et –
- chaque fois qu'elle s'applique à  $n$ , elle s'applique également à  $n + 1$ ,
- alors la propriété s'applique à tous les entiers positifs.

L'idée est que si nous pouvons montrer que la première puce est vraie, alors la deuxième puce est vraie pour tout entier positif  $m$  parce que •

- si elle est vraie pour  $m - 1$ , alors elle est vraie pour  $m$  ; et •
- si elle est vraie pour  $m - 2$ , alors elle est vraie pour  $m - 1$ , et
- ...
- si c'est vrai pour 1, alors c'est vrai pour 2.

Nous savons que c'est vrai pour 1 (c'est dans la première puce), donc la chaîne d'implications nous dit que c'est vrai pour  $m$ , qui était un entier positif arbitraire.

### Alternatives à la récursivité

La récursivité est assez courante et relativement simple à mettre en œuvre, mais elle n'est pas toujours la meilleure approche. Il n'est pas difficile de calculer le triangle de Pascal jusqu'à la 900e ligne, et vous pourriez aller plus loin si Sage ne limitait pas le nombre d'applications de récursivité :

```
sage : P = pascals_row(900) sage : len(P)
900

sage : P[-3]
403651

sage : P = pascals_row(1000)
RuntimeError : profondeur de récursivité maximale dépassée lors de l'appel d'un
objet Python
```

---

<sup>5</sup>  $F$  signifie « drôle », pas « Fibonacci ». Pourquoi posez-vous cette question ?

Cette limite existe pour une bonne raison, soit dit en passant ; la récursivité peut utiliser une quantité importante de mémoire d'un type particulier,<sup>6</sup> et une récursivité trop profonde peut également indiquer une boucle infinie, ou du moins que quelque chose a terriblement mal tourné.

Avec la suite de Fibonacci, les choses se gâtent assez rapidement, mais différemment. Essayez de calculer le n-ième nombre de Fibonacci pour des valeurs de n vraiment pas très grandes, et vous constaterez que la longueur des problèmes augmente rapidement. Sur les machines des auteurs, par exemple, on observe déjà un décalage notable à n = 30 ; à n = 35, le temps est de plusieurs secondes ; et à n = 40, on pourrait à peine se préparer une tasse de café avant la fin. On pourrait penser pouvoir contourner ce décalage avec un ordinateur plus rapide, et on se trompe, on se trompe lourdement. Le problème est qu'il y a trop de branches à chaque étape. Pour mieux comprendre, étendons le calcul du 7e nombre de Fibonacci à l'un des deux cas de base :

$$\begin{aligned}
 F7 &= F6 + F5 \\
 &= (F5 + F4) + (F4 + F3) \\
 &= [(F4 + F3) + (F3 + F2)] + [(F3 + F2) + (F2 + F1)] \\
 &= [(F3 + F2) + (F2 + F1)] + (F2 + F1) + F2 + [(F2 + F1) + F2] + (F2 + F1) \\
 &= [[[F2 + F1] + F2] + (F2 + F1)] + (F2 + F1) + F2 \\
 &\quad + [[(F2 + F1) + F2] + (F2 + F1)]
 \end{aligned}$$

Douze F ne sont pas les cas de base F1 et F2 ; chacun d'eux nécessite une récursivité. C'est beaucoup, mais ça ne paraît pas si mal. En revanche, considérons le nombre suivant :

$$F8 = F7 + F6 .$$

Nous avons déjà compté 12 récursions pour F7 , et vous pouvez regarder vers le haut pour voir que F6 en nécessite 7. Donc F8 nécessitera  $12 + 7 + 1 = 20$  récursions. (Il nous en faut une supplémentaire pour le développement de F7 .) Vous pouvez immédiatement constater que le nombre de récursions de la suite de Fibonacci croît à la manière de Fibonacci :

$$0, 0, 1, 2, 4, 7, 12, 20, 33, 54, \dots$$

Déjà F10 nécessite  $54 \approx 5 \times 10$  récursions ; remarquez le saut de taille par rapport à F7 , qui ne nécessitait « que »  $12 \approx 2 \times 7$ . Avec la suite de Fibonacci, le nombre de récursions augmente trop vite !

Calculer le 100e nombre de Fibonacci peut donc sembler hors de portée, mais il existe plusieurs solutions. Les deux premières approches reposent sur le fait que, lors du calcul de F7 , nous avons calculé F5 deux fois :

$$F7 = F6 + F5 = (F5 + F4) + F5 .$$

Nous avons ensuite calculé F4 trois fois :

$$F7 = [(F4 + F3) + F4] + (F4 + F3) .$$

... et ainsi de suite. Le problème n'est donc pas que la méthode de Fibonacci exige intrinsèquement une quantité déraisonnable de récursivité ; c'est que nous gaspillons beaucoup de calculs que nous pourrions réutiliser.

---

Pour ceux qui connaissent un peu l'informatique, nous parlons de la pile.

Que vous ayez envie d'en boire ou non est une autre histoire, mais avant que les auteurs ne se lancent dans une autre discussion sur les boissons contenant de la caféine, nous allons nous arrêter là.

Mise en cache. Une façon d'éviter de gaspiller des calculs est de les « mémoriser » dans ce qu'on appelle un cache. On peut l'envisager ainsi : on stocke les résultats précédents dans le cache. Pour implémenter le cache, on utilise une variable globale, c'est-à-dire une variable externe à une procédure, accessible depuis l'intérieur. En effet, si le cache était simplement local à la procédure, les mêmes valeurs ne seraient pas disponibles lors de la résolution du sous-problème.

Nous décrivons deux méthodes pour mettre en cache les résultats. L'une d'elles consiste à le faire explicitement, avec une liste dont la  $i$ ème valeur est le  $i$ ème nombre de Fibonacci. Nous initialisons la liste par  $F = [1, 1]$ . Chaque fois que nous calculons un nombre de Fibonacci, nous vérifions d'abord si  $F$  contient déjà cette valeur. Si c'est le cas, nous la renvoyons sans récursivité. Sinon, nous la calculons, puis l'ajoutons à  $F$  à la fin. Par exemple, pour le calcul de  $F_5$  :

- Nous devons calculer  $F_4$  et  $F_3$ . Nous n'avons ni l'un ni l'autre, nous effectuons donc une récursivité.

Pour  $F_4$ , nous devons calculer  $F_3$  et  $F_2$ . Nous n'avons pas  $F_3$ , – alors on récure.

Pour – Il faut calculer  $F_2$  et  $F_1$ . Nous avons les deux, donc nous calculons  $F_3 =$  qui est  $F_3, F_2+F_1 = 1+1 = 2$  et stockons cela à la fin de la liste  $F$ . Nous le , maintenant  $[1, 1, 2]$ . renvoyons ensuite,

Pour – nous l'avons déjà, nous renvoyons donc  $F_2 = 1$ .

$F_2$ , – Nous avons maintenant  $F_3$  et  $F_2$ , nous calculons donc  $F_4 = F_3 + F_2 = 2 + 1 = 3$ , stockons cette valeur à la fin de la liste et la renvoyons.

– Pour  $F_3$ , nous l'avons maintenant dans la liste (elle a été calculée ci-dessus), nous la renvoyons donc simplement.

- Nous avons maintenant  $F_4$  et  $F_3$ , donc nous calculons  $F_5 = F_4 + F_3 = 3 + 2 = 5$ , stockons cette valeur au fin de la liste, et la renvoyer.

Cela nous a évité d'avoir à calculer  $F_4$  plus d'une fois et  $F_3$  plus d'une fois.

De plus, la définition des nombres de Fibonacci est pratique pour cette méthode : lorsque nous avons déjà  $F_1, F_2, \dots$ , pseudo-code pour cela, nous  $F_{n-1}$  dans la liste, nous pouvons donc le stocker au bon endroit.<sup>8</sup> Pour écrire du introduisons un nouveau « mot en gras » pour notre pseudo-code : global. Cela en tête d'une liste de variables globales utilisées par l'algorithme ; pour plus de clarté, nous le listons après les entrées et les sorties.

---

<sup>8</sup>Toutes les séquences récursives n'ont pas cette chance. Si la séquence était définie par  $S_n = S_{n-2}+S_{n-3}$ , par exemple, il faudrait s'assurer que le cache a la bonne longueur avant de le stocker. Nous avons donc un peu de chance avec la suite de Fibonacci.

```

algorithme cached_fibonacci
entrées
    • n      avec n ≥ 1
sorties •
    Fn , le n-ième nombre de Fibonacci
globales
    • F    , une liste de nombres de Fibonacci (dans l'ordre)

faire si n ≤ |F|
    | soit résultat =
    Fn
    sinon soit résultat = cached_fibonacci(n - 1) + cached_fibonacci(n - 2)
    ajouter le résultat à F
    renvoyer le résultat

```

Pour transformer cela en pseudo-code, nous utilisons le mot-clé Sage « global », qui indique que F est une variable externe à la procédure. Ce n'est pas strictement nécessaire pour Sage, mais cela permet d'avertir les lecteurs qu'une variable globale est attendue.

```

sage : F = [1,1] sage :
def cached_fibonacci(n) :
    global F # cache des valeurs précédentes # si déjà
    calculé, le renvoyer si n <= len(F) :

        résultat = F[n-1]
    autre:
        # besoin de récursivité : résultat décevant
        = cached_fibonacci(n-1) \+ cached_fibonacci(n-2)

        F.append(résultat)
    résultat de retour

```

Notez que nous avons dû ajuster un élément du pseudo-code : au lieu de renvoyer F[n], nous avons renvoyé F[n-1]. Nous avons procédé ainsi car le pseudo-code suppose que les listes commencent à l'index 1, alors que Sage commence ses index à l'index 0.

Vous pouvez tester et comparer les résultats entre les deux. Il deviendra très vite évident que cached\_fibonacci() est bien plus rapide que funny\_bunnies() dès que n dépasse le plus petit des nombres :

```
sage : cached_fibonacci(10) 55  
  
sage : cached_fibonacci(25) 75025  
  
sage : cached_fibonacci(35) 9227465
```

Pour vraiment enfoncez le clou, nous allons en calculer un qui prendrait beaucoup trop de temps avec l'implémentation récursive :

```
sage : cached_fibonacci(100)  
354224848179261915075
```

N'oubliez pas la première ligne du code ci-dessus, où nous définissons  $F = [1,1]$ . Si vous l'oubliez, vous rencontrerez une erreur. Si vous avez lu attentivement et n'avez pas commis cette erreur, nous allons simuler l'erreur en réinitialisant  $F$ , puis en exécutant `cached_fibonacci` :

```
sage : réinitialiser('F')  
sage : cached_fibonacci(10)  
NameError : le nom global « F » n'est pas défini
```

Si vous étiez encore un bon lecteur et que vous réinitialisiez  $F$  pour voir ce qui se passerait, et que vous souhaitez maintenant l'exécuter à nouveau, attribuez-le simplement à nouveau :

```
sage : F = [1,1] sage :  
cached_fibonacci(10) 55
```

Deux derniers conseils. Bien que cet algorithme soit mis en cache, il reste récursif ; nous n'avons pas modifié le fait que, lorsqu'un « cache miss » ( $n$  est supérieur à la taille de  $F$ ), l'algorithme doit s'auto-appeler. Comme Sage intègre une limite de récursivité, nous rencontrons toujours le même problème qu'avec `pascals_row` :

```
sage : cached_fibonacci(1000)  
RuntimeError : profondeur de récursivité maximale dépassée lors de l'appel d'un  
objet Python
```

Vous pouvez effectivement modifier cela, mais c'est dangereux ; nous ne vous expliquerons donc pas comment procéder. Il existe généralement de meilleures options ; pour en trouver une, consultez la section suivante.

Enfin, vous n'avez généralement pas besoin d'implémenter la mise en cache vous-même. Dans le cloud, Sage vous fournit un cache gratuitement ; saisissez simplement `@cached_function`, puis démarrez la définition de la procédure sur la ligne ci-dessous.

```

sage : @cached_function def
    decorated_fibonacci(n) :
        si n == 1 ou n == 2 :
            résultat = 1
        autre:
            résultat = decorated_fibonacci(n-1) + decorated_fibonacci(n-2)

        résultat de retour
sage : décoré_fibonacci(100)
354224848179261915075

```

Notez que nous avons obtenu le même résultat qu'avec notre cache maison.

Transformer la récursivité en boucle. Une autre façon d'éviter de gaspiller des calculs est de reformuler la récursivité en boucle. Soit bmat le nombre de paires de lapins matures, btot le nombre total de paires de lapins et bnnext le nombre de paires de lapins attendues pour la saison prochaine. Puisque les lapins sont immortels, bnnext comptera tous les btot des lapins actuels, mais comme chaque paire mature produit une nouvelle paire, bnnext devra également compter deux fois toutes les paires matures bmat pour tenir compte des nouvelles paires de la saison prochaine. Donc

$$\text{bnnext} = \text{btot} + \text{bmat}.$$

Une fois la saison suivante arrivée, btot correspondra au nombre de lapins adultes produisant des couples, tandis que bnnext correspondra au nouveau total de lapins. Nous pourrions donc remplacer bmat par btot et btot par bnnext pour calculer le total de la saison suivante. Et ainsi de suite. Puisque nous connaissons déjà les valeurs des deux premières saisons, nous n'avons jamais besoin de les compter. Sachant que n = 1 correspond à la première saison et n = 2 à la seconde, nous avons besoin de la boucle pour calculer btot pour n = 3, n = 4, etc. Cela suggère le pseudo-code suivant.

```

algorithme immortal_bunnies
entrées
    • n      , le nombre de saisons
sorties •
    le nombre de lapins « immortels » après n saisons

faire laisser bmat = btot =
1 répéter n – 2 fois
    laisser bnnext = btot + bmat
    laisser bmat =
    btot laisser btot =
bnnext retourner btot

```

Pour traduire cela en code Sage, gardez à l'esprit que range() n'itère pas sur la dernière valeur ; sa limite doit donc être n-1. Il est également plus « naturel » dans Sage de commencer à compter à n = 0 ; la boucle sera donc légèrement différente du pseudo-code :

```

sage : def immortal_bunnies(n) : b_mat = b_tot =
    1
    # boucle ascendante pour
    chaque élément dans la plage (n-1) :
        b_next = b_tot + b_mat
        b_mat, b_tot = b_tot, b_next
    retourner b_tot

sage : [immortal_bunnies(i) pour i dans la plage (2,10)] [2, 3, 5, 8, 13, 21, 34,
55]

```

Contrairement à la mise en cache, Sage ne propose pas de méthode automatique permettant de transformer une procédure récursive en procédure en boucle. Il nous appartient de le faire par nous-mêmes, en raisonnant à partir de la logique du problème. Considéré comme un sujet avancé en informatique, nous ne nous y attarderons pas davantage. Le lecteur intéressé trouvera des informations complémentaires dans les ouvrages d'informatique traitant de la « programmation dynamique ».<sup>9</sup>

Les valeurs propres et les vecteurs propres résolvent un dilemme du lapin. Vous ne les connaissez peut-être pas, même si vous les avez déjà vus en cours. Nous concluons ce chapitre par une illustration de leur immense utilité.

Comme vous l'avez probablement remarqué, même les implémentations non récursives de la suite de Fibonacci reposent sur sa description récursive, dont la nature récursive nécessite la connaissance de valeurs antérieures. Il serait souhaitable de disposer d'une formule indépendante des valeurs antérieures, du moins pas explicitement. Une telle formule est appelée forme fermée d'une suite.

Une autre façon de décrire cela est de le décrire comme une équation matricielle. Les matrices

$$\begin{matrix} F_{curr} \\ F_{prev} \end{matrix} \quad \text{et} \quad \begin{matrix} F_{next} \\ F_{curr} \end{matrix}$$

Représentent les informations nécessaires à la saison actuelle et à la saison suivante pour calculer les lapins de la saison suivante. La somme des nombres de la première matrice donne l'élément supérieur de la seconde matrice, tandis que le nombre supérieur de la première matrice donne le nombre inférieur de la seconde. Cette relation peut être décrite par l'équation matricielle suivante :  $F_{curr} F_{prev}$

$$\begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix} = \text{Fsivant } F_{curr}.$$

De plus, en multipliant par les puissances de la matrice

$$M = \begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix}$$

fait monter les chiffres plus haut dans la liste. À savoir,

$$\begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix}^5 = \begin{matrix} 13 \\ 8 \end{matrix},$$

---

<sup>9</sup>L'inventeur de la programmation dynamique a déclaré un jour avoir inventé ce terme parce que son entreprise répondait aux besoins d'un homme qui nourrissait « une peur et une haine pathologiques du mot recherche... Imaginez donc ce qu'il ressentait alors pour le terme mathématique. » Utiliser un terme comme « programmation dynamique » l'a aidé à échapper à la colère de son supérieur. Il a ajouté : « Je pensais que la programmation dynamique était un bon nom... même un député ne pouvait pas s'y opposer. »

Vous les reconnaîtrez comme les 4e et 5e chiffres de la liste produite par notre programme, soit F6 et F7 , qui correspondent au nombre de lapins des 6e et 7e saisons. En général,

$$\begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix}^{n-2} = \begin{matrix} F_n \\ F_{n-1} \end{matrix},$$

car après tout

$$\begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix}^1 = \begin{matrix} 2 \\ 1 \end{matrix}, \quad \begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix}^2 = \begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix}^1 = \begin{matrix} 1 & 2 \\ 1 & 1 \end{matrix} = 32,$$

et ainsi de suite.

Si nous pouvons trouver une formule simple pour les puissances de la matrice, nous devrions pouvoir l'utiliser pour trouver la forme fermée souhaitée. Il se trouve qu'un résultat très pratique d'algèbre linéaire s'avère utile ici. Un vecteur propre  $e$  d'une matrice  $M$  sur un anneau  $R$  est un vecteur relié à une valeur propre  $\lambda \in R$  tel que

$$Me = \lambda e;$$

autrement dit,  $M$  modifie la « longueur » du vecteur. Les vecteurs propres et leurs valeurs propres ont de nombreuses utilités, dont l'une nous vient maintenant en aide.

**Théorème de la composition propre .** Soit  $M$  une matrice  $n \times n$  de vecteurs propres linéairement indépendants  $e_1, \dots, e_n$  et de valeurs propres correspondantes  $\lambda_1, \dots, \lambda_n$ . On peut réécrire  $M$  sous la forme  $Q\Lambda Q^{-1}$  où

$$Q = (e_1 | e_2 | \cdots | e_n) \text{ et } \Lambda = \begin{matrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{matrix}.$$

(La notation pour  $Q$  signifie que la  $i$  ème colonne de  $Q$  est le  $i$  ème vecteur propre de  $M$ .)

Il est facile de vérifier cela dans Sage pour  $M$ . Calculons d'abord les vecteurs propres.

```
sage : evecs = M.eigenvectors_right()
sage : evecs
[(-0.618033988749895 ?, [(1, -1.618033988749895 ?)], 1), (1.618033988749895 ?, [(1, 0.618033988749895 ?)], 1)]
```

Que signifie tout cela ? Si vous lisez attentivement le texte d'aide...

```
sage : M.vecteurs_propres_droit ?
```

... vous verrez que la liste contient un tuple par vecteur propre, et le tuple lui-même contient la valeur propre, le vecteur propre et la multiplicité de la valeur propre.

Pour notre propos, les valeurs et vecteurs propres suffisent. Leur extraction est relativement simple, mais

- nous voulons des valeurs exactes, pas approximatives, et
- que signifient ces points d'interrogation, de toute façon ?

Vous serez peut-être surpris d'apprendre que les points d'interrogation n'indiquent pas d'incertitude. Ils indiquent que les nombres appartiennent à un domaine particulier appelé le domaine des nombres algébriques. Sage désigne ce domaine par le symbole AA.

```
sage : a, b = evecs[0][0], evecs[1][0] sage : type(a)
<classe
'sage.rings.qqbar.AlgebraicNumber'>
```

Les nombres algébriques sont le plus petit corps contenant toutes les racines de polynômes à coefficients entiers. On peut donc toujours trouver un polynôme « esthétique » qui les possède comme racine. De plus, on peut souvent les réécrire sous forme d'expression avec des radicaux. Les méthodes suivantes pour les nombres algébriques permettent d'effectuer ces tâches :

a.minpoly()	renvoie le polynôme de plus petit degré qui a comme racine
a.radical_expression()	renvoie une expression radicale équivalente à a

Vous pouvez trouver beaucoup plus de méthodes de la manière habituelle, en tapant le nom d'une variable dont la valeur est un nombre algébrique, en le suivant d'un point, puis en appuyant sur la touche Tab .

Pour l'instant, essayons-les sur notre vecteur propre actuel.

```
sage : a.minpoly() x^2 - x - 1

sage : a.radical_expression()
-1/2*sqrt(5) + 1/2 sage :
b.minpoly()
x^2 - x - 1

sage : b.radical_expression()
1/2*sqrt(5) + 1/2
```

Les deux nombres ont le même polynôme minimal,10 mais leurs expressions radicales sont différentes. Ils ne sont pas si différents, cependant ; nous les appellerons 11

$$\psi = \frac{1 - \sqrt{5}}{2} = \frac{1 + \sqrt{5}}{2} \text{ et}$$

Le deuxième en particulier est extrêmement célèbre, car il s'agit du **nombre d'or**. On le retrouve à de nombreux endroits en mathématiques, et sans doute dans de nombreuses situations du monde réel. Nos lapins de Fibonacci peuvent ne pas sembler particulièrement convaincants, mais ce nombre suggère qu'ils peuvent apparaître dans des situations réelles, et **c'est effectivement le cas**.

Nous revenons à notre problème principal, celui de trouver une forme fermée pour la suite de Fibonacci. Nous avons d'abord voulu vérifier le théorème de décomposition propre. Pour construire les matrices Q et  $\Lambda$ , nous avons extrait les données d' evecs à l'aide de l'opérateur crochet [], puis construit les matrices par compréhension. Assurez-vous de bien comprendre comment ces matrices sont construites.

---

Cela a du sens si vous avez étudié ce sujet. Sinon, surveillez la prochaine session d'algèbre linéaire près de chez vous !

11Quelques anecdotes. Puisque  $\psi$  est une racine de  $x^2 - x - 1$ , nous savons que  $\psi^2 - \psi - 1 = 0$ , donc  $\psi^2 = \psi + 1$ , ou  $\psi = \psi^2 - 1 = \psi(\psi - 1)$ , ou  $\psi^{-1} = \psi - 1$ . Notez que  $\psi = 1 - \psi^{-1}$ , donc  $-\psi$  et  $\psi$  sont en fait des réciproques.

```
sage : Q = matrix([e[1][0] pour e dans evecs]).transpose() sage : L = diagonal_matrix([e[0]
pour e dans E]) sage : M == Q*L*Q.inverse()
```

Vrai

Donc le théorème fonctionne. En quoi est-il utile à notre objectif ? Rappelez-vous que

$$\underset{1 \ 1}{M^{n-2}} = \frac{F_n}{F_{n-1}}.$$

Par substitution, cela devient

$$M^{n-2} = Q \Lambda Q^{-1} \underset{n-2 \text{ fois}}{\underbrace{Q \Lambda Q^{-1} \dots Q \Lambda Q^{-1}}}.$$

Les propriétés associatives nous permettent de réécrire ceci comme

$$M^{n-2} = (Q \Lambda) Q^{-1} Q^{-1} \dots Q^{-1} Q \Lambda.$$

Cela se simplifie en

$$M^{n-2} = Q \Lambda^{n-2} Q^{-1}.$$

De plus, il est « facile » de montrer que pour une matrice diagonale comme  $\Lambda$

$$k \Lambda = \begin{matrix} \lambda_1 & & & k \\ & \ddots & & \lambda_k \\ & & \lambda_2 & \\ & & & \ddots \\ & & & & \lambda_n \end{matrix} = \begin{matrix} & & & \lambda_1 \\ & & & \lambda_2 \\ & & \ddots & \\ & & & \lambda_n \end{matrix},$$

donc on peut écrire  $\Lambda_k$  sous une forme plus simple.

```
sage : Lk = matrice_diagonale([L[i,i]^n pour i dans la plage(L.nrows())])
TypeError : aucune coercition canonique de l'anneau symbolique au champ rationnel
```

Eh bien, c'est étrange. Pourquoi ne pas lui donner d'abord une expression radicale ?

```
sage : Lk = matrice_diagonale([L[i,i].expression_radicale()^(n-2) \
pour i dans la plage(L.nrows())])
scie : Lk
[(-1/2*sqrt(5) + 1/2)^(n-2) 0] [ 0 (1/2*sqrt(5) + 1/2)^(n-2)]
```

Cela a fonctionné. Sage semble vouloir une expression radicale pour continuer ; nous pouvons également nous y adapter avec Q.

```
sage : Q = matrice([[Q[0,0].expression_radicale(), \
                     Q[0,1].expression_radicale()], \
                     [Q[1,0].expression_radicale(), \
                     Q[1,1].expression_radicale()] \
])
voir : Q
[ 1] [-1/2*sqrt(5) - 1/2 1/2*sqrt(5) - 1/2]
```

Nous pouvons maintenant calculer  $Q \wedge kQ - 1$  directement.

```
sage : Q*Lk*Q.inverse()^*matrice([[1],[1]])
[ 1/20*sqrt(5)*(1/2*sqrt(5) + ...
```

... eh bien, c'est un peu le bazar. En fait, seul l'élément supérieur nous intéresse, et nous préférerions qu'il soit entièrement simplifié.

```
sage : (Q*Lk*Q.inverse())^*matrix([[1],[1]]))[[0,0].full_simplify()
1/5*sqrt(5)*((1/2*sqrt(5) + 1/2)^n - (-1/2*sqrt(5) + 1/2)^n)
```

Eh bien, c'est pratique ! C'est

$$\frac{1}{5} \frac{1}{5} \frac{1}{2} \frac{1}{15} + \frac{n}{2} \quad \frac{1}{2} \frac{1}{15} + \frac{n}{2},$$

donc avec une légère réécriture nous constatons que

$$F_n = \frac{1}{5} (n - \psi n),$$

une forme fermée terriblement élégante !

### Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Un algorithme récursif pour un problème est un algorithme dans lequel l'algorithme s'invoque lui-même sur un cas plus petit du problème.
2. Un algorithme récursif devrait essayer d'exploiter la propriété de bon ordre des entiers () .
3. Les rangées du triangle de Pascal apparaissent à de nombreux endroits, comme dans le développement des trinômes.
4. Étant donné que le triangle de Pascal n'a qu'une seule récursivité, il n'y a aucun risque de rencontrer une RuntimeError.
5. La croissance d'une population de lapins est l'une des propriétés fondamentales des mathématiques.
6. Chaque instruction récursive peut être résolue à l'aide d'une procédure facile à mettre en œuvre.
7. Si un problème peut être résolu par récursivité, alors l'implémenter par récursivité est la meilleure approche.
8. Une récursivité très profonde peut indiquer que quelque chose a mal tourné, comme une boucle infinie.
9. Vous pouvez contourner le retard dans le calcul du 40e nombre de Fibonacci en achetant un ordinateur.
10. Un cache est une variable locale qui stocke les résultats précédents de la procédure.

11. Sage fournit un décorateur `@cached_function` qui met automatiquement en cache la procédure précédente résultats.
12. Vous pouvez parfois transformer une procédure récursive en boucle, évitant ainsi les pénalités associées à la récursivité.
13. Comme le nombre d'or est propre aux nombres de Fibonacci, ils présentent un intérêt purement théorique.
14. La décomposition en valeurs propres nous permet de réécrire une matrice  $M$  en termes d'autres matrices qui sont elles-mêmes liées à des vecteurs que  $M$  redimensionne.
15. Vous pouvez obtenir une représentation radicale d'un nombre algébrique en utilisant la fonction `.radical_simplify()` méthode.

Choix multiple.

1. La propriété de bon ordre s'applique à lequel de ces ensembles ?
  - A. UN.
  - B.
  - C.
  - D.
2. Le triangle de Pascal est un bon exemple de récursivité car :
  - A. Il a de nombreuses applications.
  - B. Chaque ligne peut être définie en fonction de la ligne précédente.
  - C. C'est l'un des problèmes fondamentaux des mathématiques.
  - D. Il constitue une critique efficace de la philosophie du doute systématique de Descartes.<sup>12</sup>
3. Les nombres de Fibonacci sont un bon exemple de récursivité car :
  - A. Ils ont de nombreuses applications.
  - B. Chaque nombre peut être défini en fonction des nombres précédents.
  - C. Ils se rapportent à l'un des problèmes fondamentaux des mathématiques.
  - D. Si vous pouvez obtenir une séquence de nombres portant votre nom, [vous devez avoir raison — vous venez de devoir!](#)
4. Quelle technique de preuve est essentiellement une sorte de récursivité ?
  - A. contradiction
  - B. contraposée
  - C. induction
  - D. test d'hypothèse

5. Que se passe-t-il dans Sage si vous programmez accidentellement une récursivité infinie ?

  - A. l'ordinateur explode
  - B. le programme se bloque
  - C. RuntimeError
  - D. RecursionError
6. Pourquoi est-ce en fait une mauvaise idée d'implémenter la séquence de Fibonacci avec une récursivité naïve ?
  - A. Une implémentation naïve répète la grande majorité de ses calculs.
  - B. Sauf dans les cas de base, chaque récursivité double le nombre de calculs nécessaires.
  - C. Le temps nécessaire pour calculer les nombres augmente à peu près aussi vite que les nombres eux-mêmes.  
moi-même.
  - D. Tout ce qui précède.
7. Laquelle des alternatives suivantes à la récursivité devez-vous déterminer en analysant l'algorithme, plutôt qu'en utilisant une combinaison de commandes ou de variables Sage ?

---

<sup>12</sup>Références gratuites et non pertinentes à la philosophie moderne incluses sans frais supplémentaires.

A. mise en

cache B. recherche d'une  
forme fermée C.

induction D.

boucle 8. Lequel des énoncés suivants définit le mieux l'anneau des nombres algébriques ?

A. AA

B. l'ensemble de toutes les racines de polynômes à coefficients entiers C. le plus petit corps contenant les racines de polynômes à coefficients entiers D. le plus petit corps contenant tous les nombres irrationnels 9. La meilleure commande ou

méthode pour transformer un nombre algébrique en un nombre exact, mais facile à lire

l'expression numérique est :

A. .exactify()

B. .full\_simplify()

C. .minpoly()

D. .radical\_expression()

10. Laquelle des propositions suivantes décrit le mieux la signification d'un vecteur propre ?

A. un vecteur utile pour trouver des formes fermées de suites ; B. un vecteur que

la matrice redimensionne, mais qui reste dans la même direction ; C. un vecteur

linéairement indépendant des autres vecteurs propres ; D. un vecteur qui peut

être utilisé pour décomposer une matrice en une forme utile en calcul. Bonus : Étant donné la définition récursive d'une suite de lapins mathématiques, la meilleure façon de trouver une « forme fermée » pour décrire la suite est de : A.

acquérir des chiens ; B. les enfermer ; C. les enfermer ; D.

les habiller ; E. les

enfermer dans des  
combinaisons

spatiales, comme

Glenda :13



Réponse courte.

1. Pourquoi pascals\_row n'aurait-il pas autant de problèmes avec la récursivité que la séquence de Fibonacci ?

Autrement dit, quelle différence dans le pseudo-code (et donc dans le code Sage) rend pascals\_row plus propice à la récursivité que la séquence de Fibonacci ?

2. Un cache serait-il aussi utile pour pascals\_row que pour la suite de Fibonacci ? Pourquoi ?  
pas?

3. À la page 77, nous vous avons demandé de tracer  $x$ ,  $x^2$ ,  $\log_{10} x$  et  $e^x$  et classez-les du plus grand au plus petit. Recréez ce graphique sur l'intervalle  $[1, 15]$  en y ajoutant une courbe dont les points sont définis par les nombres de Fibonacci de 1 à 15. Classez à nouveau ces fonctions selon la croissance la plus rapide. Indice : rappelez-vous que la procédure line() trace une « courbe » définie par un ensemble fini de points. Vous devrez peut-être ajuster ymax afin de bien visualiser le tracé.

13Téléchargé depuis [le site Web Plan 9 de Bell Labs](#) et utilisé avec permission (nous pensons que la formulation est un peu vague).

## Programmation.

1. Supposons que  $y = x^m q$ , où  $x$  ne divise pas  $q$ . Par exemple, si  $y = 12$  et  $x = 2$ , alors  $m = 2$ . On appelle  $m$  la multiplicité de  $x$ , et on peut la calculer à l'aide de l'algorithme récursif suivant :

```

algorithme multiplicité entrées
    • x et y,
        deux objets tels que « x divise y » a du sens sorties • le nombre de fois que
        x divise y

faire si x ne divise pas y
    résultat = 0
sinon
    résultat = 1 + multiplicité( y/x )
    renvoyer le résultat

```

- (a) Implémentez ceci comme un programme Sage. Vérifiez qu'il donne les résultats corrects pour les éléments suivants :
- ant :
- (i)  $y = 12$ ,  $x = 2$  (ii)  $y = 12$ ,  $x = 3$  (iii)  $y = t$ ,  $x = t^4 + t^2$
- (b) Il est possible de résoudre ce problème de manière non récursive, en utilisant une boucle while. Écrivez le pseudo-code d'un tel algorithme, puis implémentez-le dans Sage. Testez votre programme sur les mêmes exemples.
2. Réécrivez pascals\_row pour qu'il utilise un cache manuel. N'utilisez pas la fonction `@cached_function` de Sage ; utilisez plutôt une liste dont la  $i$ ème entrée est également une liste : la  $i$ ème ligne du triangle de Pascal.
3. Nous pouvons compter la complexité de la récursivité d'une procédure avec une variable globale appelée `invocations`. Nous ajoutons deux lignes au début de la procédure récursive qui déclarent les invocations globales, puis nous leur ajoutons 1. Par exemple, nous pourrions utiliser `funny_bunnies()` comme suit :

```

sage : def funny_bunnies(n) : invocations
    globales invocations =
    invocations + 1
    si n == 1 ou n == 2 :
        résultat = 1
    sinon :
        résultat = funny_bunnies(n-1) + funny_bunnies(n-2)
    résultat de retour

```

Désormais, à chaque exécution, nous définissons d'abord les invocations à 0, puis nous exécutons le programme, puis nous affichons les invocations. Par exemple :

```
sage : invocations = 0 sage :
funny_bunnies(7) 13
```

```
sage : invocations 25
```

(Ce n'est pas la même chose que le nombre de récursions.)

- (a) Procédez ainsi pour les valeurs  $n = 7, 8, 9, 10$  et  $11$ . Diriez-vous que cette suite se comporte comme les nombres de Fibonacci, comme nous l'avons vu avec le nombre de récursions ? Si oui, trouvez une formule récursive pour décrire le modèle dans les nombres. Sinon, comment le décririez-vous ?
- (b) Modifiez le programme `pascals_row` pour calculer le nombre d'invocations, puis exécutez-le pour quelques valeurs de  $n$ . Dans ce cas, le nombre d'invocations est-il de type Fibonacci ? Si oui, trouvez une formule récursive pour décrire le motif dans les nombres. Sinon, comment le décririez-vous ?
4. John von Neumann a montré qu'il est possible de représenter tout nombre naturel en termes des symboles `{}` et :

$$\begin{aligned}0 &\leftrightarrow \\1 &\leftrightarrow \{\} \\2 &\leftrightarrow \{, \{\}\} \\3 &\leftrightarrow \{, \{\}, \{\{\}, \{\}\}\} \\\vdots \\n &\leftrightarrow \{n - 1, \{n - 1\}\}\end{aligned}$$

Comme vous pouvez le voir à partir de la dernière ligne en particulier, il s'agit d'une définition récursive :

$$n = (n - 1) \quad \{n - 1\}.$$

Implémentez ceci en tant que code Sage.

Astuce : Ce n'est pas compliqué, mais il faut être prudent. Tout d'abord, la nature récursive de l'algorithme implique que la procédure doit renvoyer un ensemble figé. Pour créer un ensemble figé `R` contenant un autre ensemble figé `S`, ainsi que d'autres éléments, créez d'abord `R` comme un ensemble mutable (`R = set()`), ajoutez-y `S`, ajoutez les autres éléments (en utilisant `.add()`, `.update()`, etc. selon le cas), et enfin convertissez `R` en ensemble figé.

5. Supposons que vous ayez un ensemble de  $n$  objets et que vous souhaitez lister ses sous-ensembles de  $m$  éléments (c'est-à-dire ses sous-ensembles contenant exactement  $m$  éléments).<sup>14</sup> L'algorithme suivant accomplirait cela pour vous :

---

<sup>14</sup>Cela peut sembler être le genre de question qui ne concerne que les mathématiciens les plus purs, mais c'est en fait lié à la raison pour laquelle vous ne semblez jamais acheter un ticket gagnant pour PowerBall.

combinaisons d'algorithme  
entrées

- S, un ensemble de n objets
- m, le nombre d'objets que vous souhaitez choisir parmi S, avec  $m \leq n$  sorties • les sous-ensembles de S de taille n

faire si  $|S| = m$

- soit résultat = {S}
- sinon
- soit résultat =
- pour s     S
- soit U = S\{s
- ajouter des combinaisons (U, m) au résultat
- renvoyer le résultat

(a) Implémentez ceci en tant que programme Sage.

Astuce : Ce n'est pas compliqué, mais il faut être prudent. Premièrement, la nature récursive de l'algorithme signifie que le résultat doit être un ensemble gelé. Cependant, créer un ensemble gelé contenant un ensemble S ne fonctionne pas avec `frozenset(S)`, car cela crée un ensemble gelé dont les éléments sont les éléments de S, convertissant ainsi S d'un ensemble mutable en un ensemble gelé, plutôt que de créer un ensemble gelé dont l'un des éléments est S. Pour créer un ensemble gelé dont l'élément est un autre ensemble, créez d'abord un ensemble mutable, ajoutez-lui un ensemble gelé, puis convertissez l'ensemble mutable en ensemble gelé. En résumé, la ligne du premier cas du pseudo-code se transforme en trois lignes de code Sage. (b) Soit  $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$  et évaluez la taille des combinaisons ( $S, i$ )

pour chaque  $i = 0, \dots, 8$ .

Les tailles obtenues apparaissent plus haut dans le texte. Où ?

(c) Pensez-vous qu'il s'agit d'un accident bizarre ou que cela soit vrai en général ? Qu'est-ce qui vous fait dire cela ?

## CHAPITRE 10

## Rendre vos images en 3D

Ce chapitre examine les objets et graphiques tridimensionnels que vous pourriez utiliser.

De nombreuses commandes bidimensionnelles bénéficient d'une extension 3D, avec certaines options également reprises. Comme précédemment, nous n'illustrerons pas toutes les options, car elles pourraient évoluer dans les futures versions de Sage.

La combinaison de graphiques s'effectue également par addition, et la commande `show()` s'applique également aux tracés 3D. Pour les tracés 3D, la commande `show()` ajoute une option permettant de sélectionner un visualiseur ou un moteur de rendu :

- Lorsque vous utilisez `show()` comme commande, telle que `show(p, ...)`, utilisez l' option de rendu pour sélectionner l'une des options suivantes : « webgl » (le plus rapide), « canvas » (peut mieux fonctionner avec la transparence) ou « tachyon ». Cette dernière option donne une image statique, tandis que 'webgl' et 'canvas' sont interactifs. • Lorsque vous utilisez `show()` comme méthode, comme `p.show(...)`, utilisez l' option viewer pour sélectionner l'une des options suivantes : 'jmol' (nécessite Java), 'java3d' (nécessite également Java), 'canvas3d' (navigateur uniquement, utilise JavaScript) et encore une fois 'tachyon'. Encore une fois, tous sauf le dernier sont interactifs.

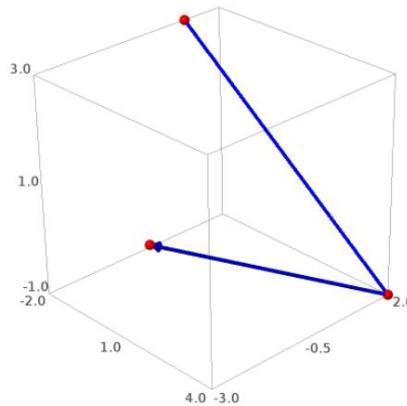
Qu'entend-on par « interactif » ? La nouveauté des graphiques 3D est leur manipulation : vous pouvez changer de point de vue ! Pour ce faire, cliquez pour activer l'affichage, puis faites glisser pour modifier l'angle de vue. Vous pouvez également zoomer en faisant défiler l'écran. D'autres options d'affichage sont disponibles via un menu accessible par un clic droit ; nous vous laissons les explorer.

### objets 3D

Des choses « droites ». Nombre des procédures graphiques 2D que nous avons étudiées précédemment ont une procédure correspondante en 3D : `point()`, `ligne()`, `flèche()`, `polygone()`. Comme toutes ces procédures nécessitent un ou plusieurs points, Sage reconnaîtra que les points sont donnés en 3D et les représentera graphiquement de manière appropriée. De nombreuses options sont conservées.

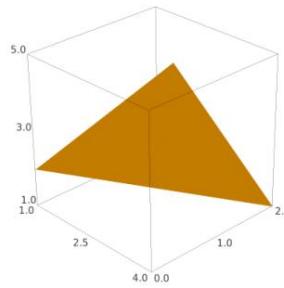
Si nous attribuons à la commande `point()` un ou plusieurs triplets (plutôt que des paires), elle tracera des points dont les coordonnées sont déterminées par un triplet listé. Les lignes sont également représentées graphiquement avec la commande `line()` . En 3D, la commande `line()` possède une option `arrow_head` ; si elle est définie sur `True`, la pointe de flèche n'est affichée qu'au dernier point.

```
sage : pts = ((-2,1,3),(4,2,-1),(2,-3,1)) sage : p1 =
point(pts,couleur='rouge',taille=20) sage : p2 =
ligne(pts,couleur='bleu',épaisseur=7,pointe_de_flèche=Vrai) sage : p1+p2
```



Les polygones sont représentés graphiquement en 3D comme en 2D avec la commande `polygon()`, qui nécessite une collection de points. Les seules options disponibles en 3D sont la couleur et l'opacité. Voici un triangle :

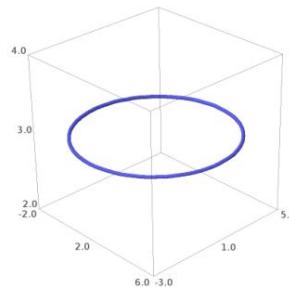
```
sage : polygone([(1,0,2),(3,1,5),(4,2,1)],couleur='orange')
```



Des courbes. La procédure « `circle` » permet également de créer des cercles en 3D, nécessitant là encore un point central et une taille. Les cercles représentés graphiquement avec la commande `circle()` seront placés parallèlement au plan  $xy$  ; pour d'autres orientations, vous devrez utiliser d'autres méthodes décrites plus loin.

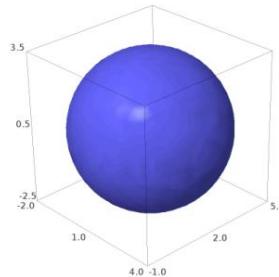
Il n'existe pas d'options de couleur de bord ou de couleur de face pour les cercles en 3D. Comme en 2D, si vous souhaitez un disque, définissez l'option de remplissage sur « `Vrai` ».

```
sage : cercle((2,1,3),4,epaisseur=10)
```



En 3D, l'ensemble de tous les points équidistants d'un point particulier est une sphère. On peut représenter graphiquement sphères en utilisant la commande `sphere()`, qui nécessite un centre et un rayon (taille).

```
sage : sphère(centre=(1,2,.5),taille=3)
```



### Tracés 3D de base

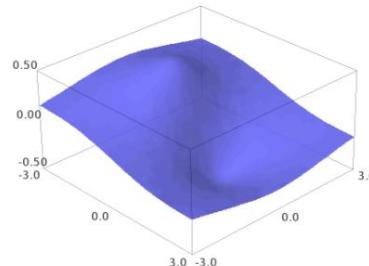
Si vous avez suivi des cours de calcul différentiel et intégral, n'auriez-vous pas souhaité que votre calculatrice portable puisse représenter graphiquement ces surfaces ? Heureusement, Sage le peut ! Nombre des procédures graphiques 2D étudiées précédemment ont une correspondance en 3D. Il existe plusieurs façons de représenter des relations tridimensionnelles :

- Coordonnées cartésiennes avec soit
  - $z$  en fonction de  $x$  et  $y$  ; – une équation implicite en termes de  $x$ ,  $y$  et  $z$ , mais pas nécessairement une fonction ; – avec  $x$ ,  $y$  et  $z$  en termes d'une troisième variable  $t$ , un paramètre ; – avec  $x$ ,  $y$  et  $z$  en termes de deux autres paramètres  $u$  et  $v$  ; • coordonnées sphériques ; et • coordonnées cylindriques.

Graphiques cartésiens génériques. La relation tridimensionnelle la plus courante définit la valeur d'une variable (souvent  $z$ ) en fonction de deux autres (généralement  $x$  et  $y$ ). Contrairement à la commande `plot()` 2D, la commande `plot3d()` ne permet pas de détecter les pôles ni d'afficher les asymptotes.

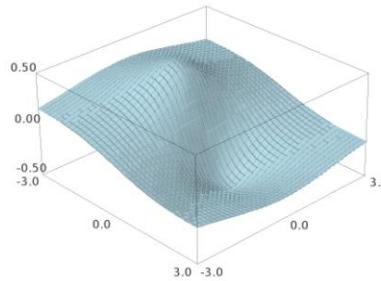
Dans les quelques exemples suivants, nous représentons graphiquement  $f(x, y) = -x/(1+x^2+y^2)$  avec deux styles de tracés différents et avec diverses options.

```
sage : f(x,y) = -x/(1+x^2+y^2) sage :
plot3d(f(x,y), (x,-3,3), (y,-3,3), opacité=.8)
```



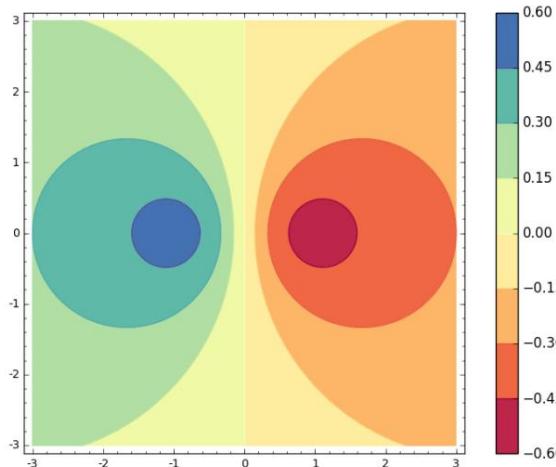
La définition de l' option de maillage sur True produit des graphiques avec une grille  $xy$  sur la surface, comme ceux de nombreux manuels :

```
sage : plot3d(f(x,y), (x,-3,3), (y,-3,3), opacité = 0,8, \ color='bleu clair',
mesh=True)
```



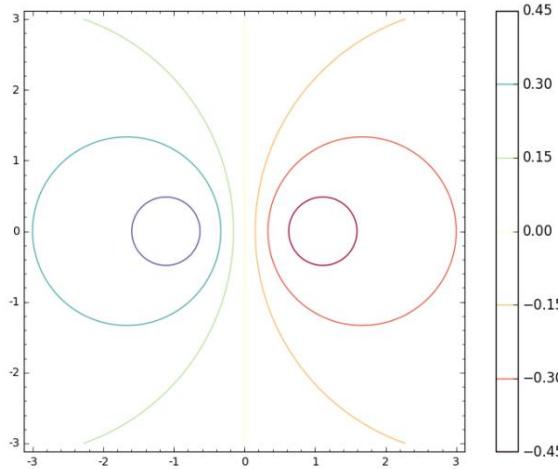
Lorsqu'une partie du graphique 3D est masquée, il peut être plus facile de visualiser la même surface à l'aide d'un « tracé de contour ». Imaginez ce tracé comme une vue aérienne du graphique, avec des points de même couleur ayant des valeurs  $z$  comprises dans la même plage, comme indiqué sur la barre de couleur à droite. Avec la carte de couleurs « Spectrale » utilisée ci-dessous, les points les plus bas sont en rouge et les points les plus hauts en bleu. Les niveaux de contour (les incrémentés de valeur  $z$  affichés sur la barre de couleur) ont été déterminés automatiquement, mais ils peuvent être spécifiés à l'aide d'une liste pour l'option « Contours » .

```
sage : contour_plot(f(x,y),(x,-3,3),(y,-3,3), cmap='Spectral', \ colorbar=True)
```



Les limites entre les régions de couleur sont appelées « courbes de niveau », ce qui signifie que  $f(x, y) = k$  pour une constante  $k$ . Si vous préférez n'afficher que les courbes de niveau, utilisez l'option `fill=False`. (Vous remarquerez que de nombreux auteurs de manuels scolaires utilisent cette option.)

```
sage : var('y') sage :
contour_plot(f(x,y), (x,-3,3), (y,-3,3), \ cmap='Spectral', fill=False,
colorbar=True)
```



Plans tangents : combinaison de tracés 3D. Représentons graphiquement une fonction  $f(x, y)$  et le plan tangent à  $f(x, y)$  en un point  $P_0 = (x_0, y_0, z_0)$ . Rappelons que le plan tangent dépend des dérivées partielles premières de  $f$  en  $P_0$  et est donné par

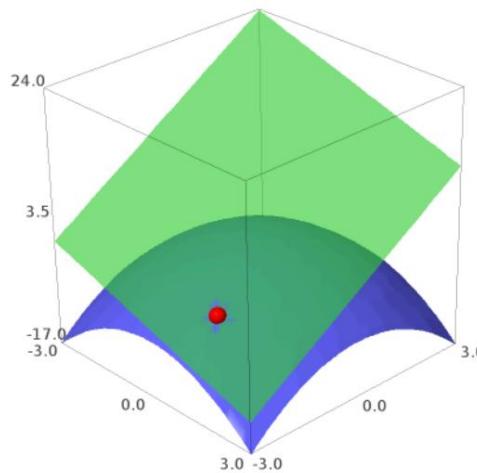
$$z = f_{x}(x_0, y_0)(x - x_0) + f_{y}(x_0, y_0)(y - y_0) + z_0,$$

Où  $f$  est la dérivée partielle première. On peut facilement écrire une procédure Sage pour calculer le plan tangent et représenter graphiquement  $f(x, y)$  et le plan tangent sur la région rectangulaire  $[a, b] \times [c, d]$  :

```
sage : def plot_f_tanplane(f,pt,a,b,c,d) : f(x,y)=f x0,y0=pt
        fx = diff(f,x)
        fy = diff(f,y)
        z0 = f(x0,y0)
        tanplane =
        fx(x0,y0)*(x-x0) +
        fy(x0,y0)*(y-y0) + z0 p1 = plot3d(f,(x,a,b),(y,c,d)) p2 = plot3d(tanplane,
        (x,a,b),(y,c,d), \
        couleur='vert citron',opacité=.6) p3 =
        point((x0,y0,z0),couleur='rouge',taille=30) renvoie p1 + p2
        + p3
```

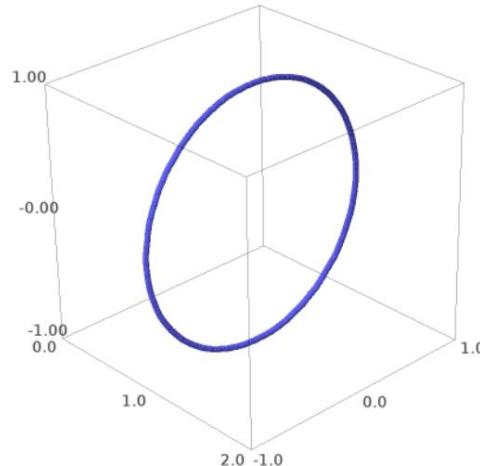
Nous pouvons maintenant générer des graphiques de plans tangents pour toute fonction dont les dérivées partielles sont définies. Par exemple :

```
sage : var('y')
sage : plot_f_tanplane(1-x^2-y^2,(1,-2),-3,3,-3,3)
```



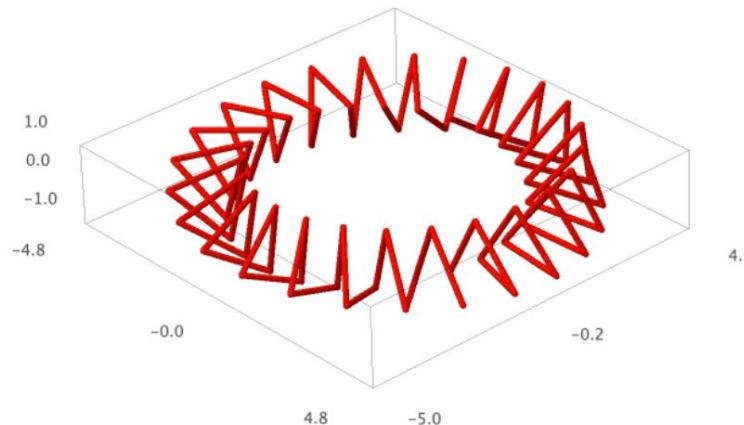
Paramétrage en 3D. Une autre façon de tracer des objets tridimensionnels est le paramétrage. Les courbes sont paramétrées dans une seule variable, souvent appelée  $t$ . Nous avons vu précédemment que la commande `circle()` trace toujours un cercle 3D parallèle au plan  $xy$ . Le paramétrage permet d'orienter les cercles d'autres manières, par exemple parallèlement au plan  $x$  et  $z$  :

```
sage : var('t') sage :
parametrique_plot3d((1,cos(t),sin(t)),(t,0,2*pi),epaisseur=10)
```



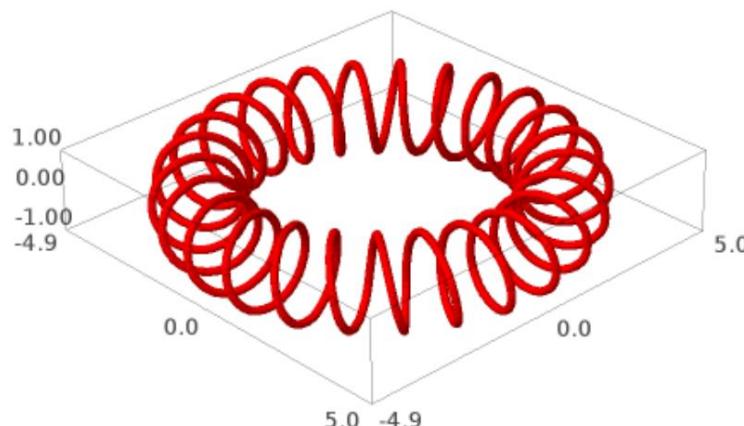
Le nombre de points à tracer par défaut est de 75, mais il est souvent nécessaire d'utiliser plus de points pour lisser le graphique. Voici une « spirale toroïdale » qui ne semble pas particulièrement spiralée.

```
sage : var('t') sage :
p = parametric_plot3d(((4+sin(25*t))*cos(t), (4+sin(25*t))*sin(t), cos(25*t)),
                      (t,0,2*pi), couleur='rouge', epaisseur=10) sage :
show(p,rapport_aspect=1)
```



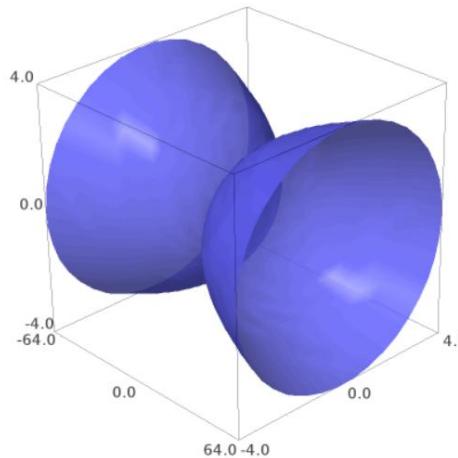
La raison est qu'il faut environ 200 points pour le rendre fluide.

```
sage : var('t')
sage : p = parametric_3d(((4+sin(25*t))2)cos(t), \ (4+sin(25*t))2sin(t),
    cos(25*t)), (t,0,2*pi), \ couleur='rouge', épaisseur=10,
    plot_points=200) sage : show(p,aspect_ratio=1)
```



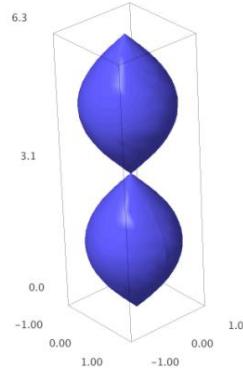
La même commande `parametric_plot3d()` permet de représenter graphiquement des surfaces paramétriques. La définition de surfaces paramétriques nécessite deux variables indépendantes, souvent appelées  $u$  et  $v$ . Par exemple, dans Sage, on peut représenter graphiquement la surface définie par  $x = u$ ,  $y = u \sin(v)$ ,  $z = u \cos(v)$  pour  $-4 \leq u \leq 4$  et  $0 \leq v \leq 2\pi$  :

```
sage : var('u v')
sage : paramétrique_plot3d((u^3,u*sin(v),u*cos(v)),\n    (u,-4,4), (v,0,2*pi), opacité=.8)
```



Coordonnées cylindriques et sphériques. Les surfaces données dans des systèmes de coordonnées cylindriques ou sphériques peuvent être considérées comme des paramétrisations particulières. À partir des coordonnées cylindriques  $(r, \theta, z)$ , nous les convertissons en coordonnées rectangulaires par  $x = r \cos\theta$ ,  $y = r \sin\theta$  et  $z = z$ . Voici la surface  $r = \sin(z)$ , pour  $0 \leq \theta \leq 2\pi$  et  $0 \leq z \leq 2\pi$ .

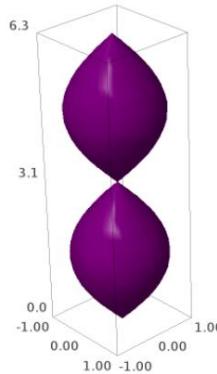
```
sage : var('u v')
sage : show(parametric_plot3d((sin(u)*cos(v),sin(u)*sin(v),u), \
(u,0,2*pi), (v,0,2*pi)), rapport hauteur/largeur=1)
```



Une méthode alternative utilise `plot3d()` avec une transformation cylindrique. La commande `Cylindrical()`, utilisée ci-dessous, affichera le rayon en fonction de l'azimut ( $\theta$ ) et de la hauteur ( $z$ ).

```
sage : S=Cylindrique('rayon', ['azimut', 'hauteur']) sage : var('theta, z')
sage : show(plot3d(sin(z),
(theta,0,2*pi), (z,0,2*pi), \
transformation=S, couleur='violet'), rapport_aspect=1)
```

Les deux méthodes produisent le même graphique.

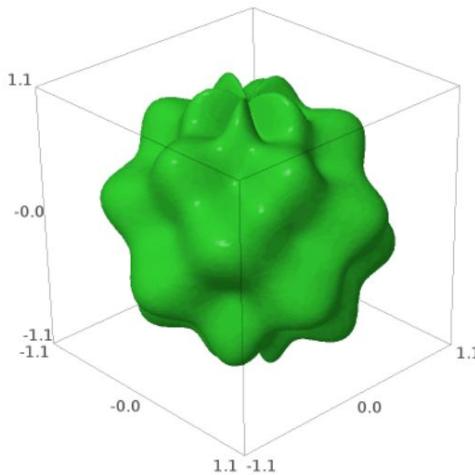


La conversion des coordonnées sphériques  $(\rho, \theta, \varphi)$  en coordonnées rectangulaires est donnée par  $x = \rho \sin \varphi \cos \theta$ ,  $y = \rho \sin \varphi \sin \theta$  et  $z = \rho \cos \varphi$ . La surface définie par  $\rho = 1 + \sin(5\theta)\sin(6\varphi)$  pour  $0 \leq \theta \leq 2\pi$  et pour  $0 \leq \varphi \leq \pi$  est appelée « sphère bosselée » et serait représentée graphiquement dans Sage par

```
sage: var('theta phi') sage:
rho = 1+(1/6)*sin(5*theta)*sin(6*phi) sage:
parametric_plot3d((rho*sin(phi)*cos(theta), \
rho*sin(phi)*sin(theta), rho*cos(phi)), (theta,0,2*pi),
(phi,0,pi), plot_points=[200,200], color='limegreen')
```

Nous pouvons également former le même graphique à l'aide de la commande `plot3d()` et d'une transformation appropriée. Cette transformation est donnée par la commande `Spherical()`, qui visualise le rayon en fonction de l'azimut ( $\theta$ ) et de l'inclinaison ( $\varphi$ ).

```
sage : T = Sphérique('rayon', ['azimut', 'inclinaison']) sage : var('phi theta')
sage :
plot3d(1+(1/6)*sin(5*theta)*sin(6*phi), (theta,0,2*pi), \
(phi,0,pi), transformation=T, plot_points=[200,200], color='limegreen')
```



### Outils avancés pour les tracés 3D

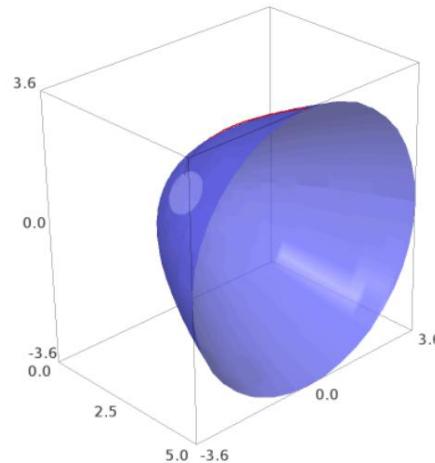
Sage propose un certain nombre d'outils qui aident à la production de types spéciaux de tracés 3D dans Coordonnées cartésiennes, cylindriques ou sphériques.

Surfaces de révolution. Une surface de révolution est un type particulier. Il est possible de les représenter graphiquement par paramétrisation, mais il est certainement utile que Sage dispose d'une procédure spécifique pour ces graphiques. Utilisez `revolution_plot3d()` pour les représenter graphiquement en se basant uniquement sur la fonction, l'axe de révolution et l'intervalle. Rappelons que si l'on fait tourner une fonction  $f(x)$  autour de l'axe des  $x$ , son aire est donnée par

$$S = \frac{2\pi f(x) \sqrt{1 + f'(x)^2} dx}{\text{un}}$$

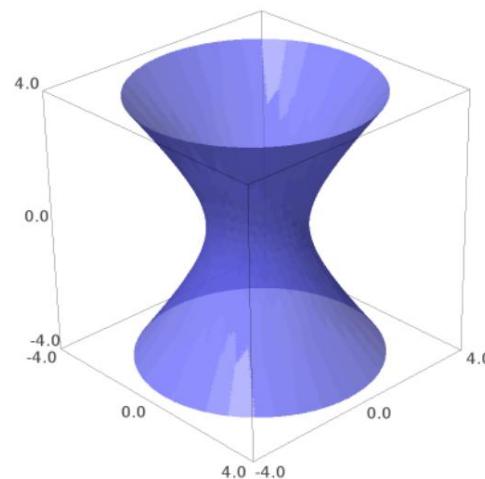
Si  $f(x) = 3x - 2$ , nous pouvons demander à Sage de calculer l'aire de la surface de révolution et de visualiser à la fois la courbe  $f(x)$  et la surface 3D.

```
sage : f(x) = sqrt(3*x-2) sage :
integrale(2*pi*f(x)*sqrt(1+diff(f,x)**2), x, 0, 5) 1/18*pi*(61*sqrt(61) - 1)
sage : show(revolution_plot3d(f,
(x,0,5), parallel_axis='x', show_curve=True, opacity=0.7), aspect_ratio=1)
```



Tracés implicites en 3D. Comme en 2D, la manière la plus pratique de décrire certaines surfaces est implicite. Cette fois, les équations peuvent impliquer une ou plusieurs variables  $x$ ,  $y$  et  $z$ . Avec la commande `implicit_plot3d()`, nous devons spécifier une équation, ainsi que les valeurs maximale et minimale de ces trois variables. Dans cet exemple, la commande `implicit_plot3d()` permet de générer un hyperbololoïde d'une seule feuille :

```
sage : var('xy z') sage :
implicit_plot3d(x^2/2+y^2/2-z^2/3==1, (x,-4,4), (y,-4,4), \ (z,-4,4), opacité=.7,
plot_points=200)
```



Après tout cela, n'êtes-vous pas prêt pour une boule de neige aux cerises sauvages ?<sup>1</sup>

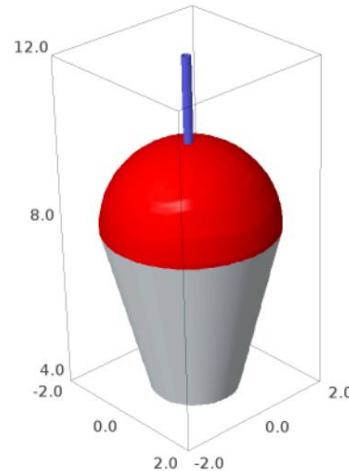
---

<sup>1</sup>Cerise sauvage car le gâteau de mariage n'apparaîtra pas.

```
sage: var('z theta r phi') sage: bh=4
# coordonnée z pour la base de la tasse sage: th=8 #
coordonnée z pour le haut de la tasse sage: br=bh/(th-
bh) # rayon de la base de la tasse sage: tr=th/(th-bh) # rayon
du haut de la tasse sage: a = .1 #rayon de la paille sage:
cupside=parametric_plot3d((z*cos(theta)/
(th-bh), \ z*sin(theta)/(th-bh),z), (theta,0,2*pi), \ (z,bh,th), color='aliceblue')

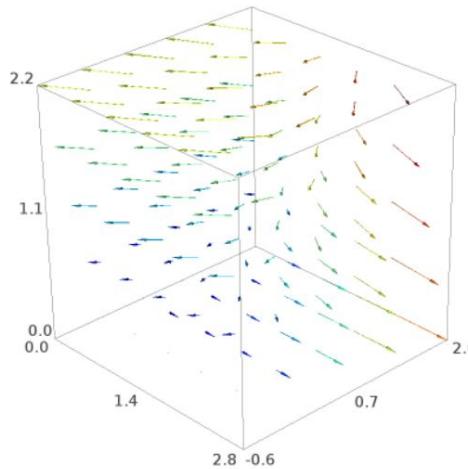
sage : cupbase=parametric_plot3d((r*cos(théta), r*sin(théta), bh), \
(theta,0,2*pi), (r,0,br),color='aliceblue') sage :
snow=parametric_plot3d((tr*sin(theta)*cos(phi), \ tr*cos(theta)*cos(phi),
tr*sin(phi)+th), \ ) (theta,0,2*pi'), (2'),color

sage : paille = paramétrique_plot3d((a*cos(theta),a*sin(theta),z), \ (theta,0,2*pi),
(z,bh+.01,th+2*tr)) sage : show(côté tasse+base
tasse+neige+paille,rapport_aspect=1)
```



Champs vectoriels. Un champ vectoriel attribue à chaque point du domaine ( 2 ou 3 ) un vecteur bidimensionnel ou tridimensionnel. Les champs vectoriels bidimensionnels peuvent être représentés graphiquement avec `plot_vector_field()`. Les champs vectoriels tridimensionnels peuvent être représentés graphiquement avec la commande `plot_vector_field3d()` . Dans l'exemple ci-dessous, nous représentons le champ vectoriel  $F(x, y, z) = -2xyi - 3zj + zk$ .

```
sage : plot_vector_field3d((2*x*y,-3*z,z),(x,0,2),(y,0,2),(z,0,2))
```



### Monter une colline

Quel est le chemin le plus rapide pour gravir une colline ? Commencez par regarder autour de vous, trouvez le chemin le plus raide possible et avancez un peu dans cette direction. Répétez ensuite ce processus en réévaluant votre trajectoire après avoir parcouru une courte distance. Continuez ainsi jusqu'à atteindre le sommet de la colline. Il vous suffit de regarder immédiatement autour de votre position actuelle à tout moment du parcours. Cette stratégie décrit une technique mathématique importante appelée « ascension de gradient ».2 Comme vous le savez, le sommet de la colline serait

appelé un maximum local. Les méthodes exactes peuvent permettre de trouver les coordonnées de ce maximum local. Lorsque les méthodes exactes échouent, la méthode d'ascension de gradient donne une excellente approximation. Mieux encore, elle permet également d'approcher le chemin de la montée la plus raide.

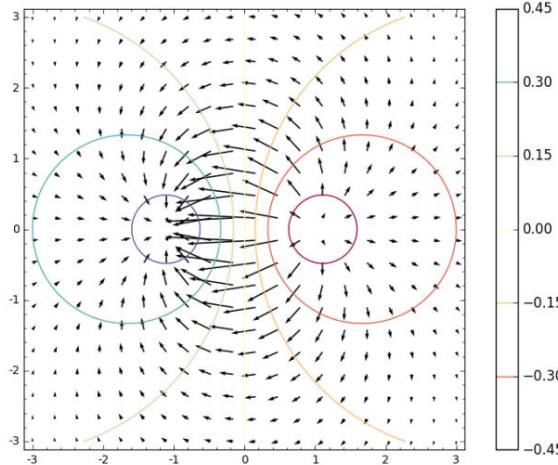
Comment savoir dans quelle direction procéder ? Le gradient d'une fonction  $f(x, y)$  est le fonction vectorielle  $f$  définie par

$$f(x, y) = \frac{\partial f}{\partial x} \vec{i} + \frac{\partial f}{\partial y} \vec{j}$$

(Ici,  $\vec{i}$  et  $\vec{j}$  représentent les vecteurs de base canoniques le long des axes  $x$  et  $y$ .) Lorsqu'il est évalué en un point  $(x_0, y_0)$ , le gradient donne un vecteur dans la direction du taux de variation maximal. Par exemple, si  $f(x, y) = -x/1+x^2+y^2$  comme précédemment, nous pouvons tracer son champ de vecteurs de gradient 2D. Chaque vecteur pointe dans la direction de la montée la plus raide à partir de son point de queue.

```
sage : var('y')
sage : f(x,y)=-x/(1+x^2+y^2) sage :
cp = contour_plot(f(x,y),(x,-3,3),(y,-3,3), \ cmap='Spectral',
                  fill=False,colorbar=True) sage : vf =
plot_vector_field(f.gradient(),(x,-3,3),(y,-3,3)) sage : show(cp+vf)
```

2Après avoir gravi la colline, vous voudrez peut-être redescendre — ou peut-être n'avez-vous jamais voulu la gravir la première fois Place ! Dans ce cas, vous pouvez utiliser la méthode de descente de gradient. Vous voulez deviner comment ça marche ?



Revenons à notre ascension de gradient : si nous calculons le gradient à notre point de départ, puis faisons un petit pas dans cette direction, nous pouvons répéter le processus à partir de ce point jusqu'au sommet de la colline. Les petits pas sont simplement des vecteurs de gradient réduits par une petite valeur de  $\alpha$ . À partir de n'importe quel point  $(x_n, y_n)$ , nous passons au point suivant  $x_{n+1}, y_{n+1}$  par la formule

$$x_{n+1}, y_{n+1} = (x_n, y_n) + \alpha F(x_n, y_n).$$

Puisque chaque point dépend uniquement du point précédent, une implémentation doit se souvenir uniquement de la position actuelle pour calculer le suivant. Plutôt que d'utiliser des indices comme ci-dessus, nous pouvons réécrire la formule plus simplement en utilisant  $q$  comme point actuel et  $p$  comme point précédent :

$$q = p + \alpha F(p)$$

Comment savoir si nous avons atteint le sommet ? Près du maximum local, la magnitude du vecteur gradient sera proche de zéro. (Cela signifie que le plan tangent sera presque horizontal.)

Lorsque ce vecteur de gradient court est réduit de  $\alpha$ , l'ajout du vecteur  $\alpha F(p)$  à  $p$  maintiendra  $q$  très proche de  $p$ . On peut donc dire que nous avons atteint le maximum local lorsque nos pas deviennent très petits, c'est-à-dire lorsque la distance entre les points  $p$  et  $q$  est inférieure à une certaine précision souhaitée.

Comme nous ne pouvons pas prédire le nombre d'étapes avant de commencer, nous utilisons une boucle `while` pour la répétition.

Nous pouvons tirer parti de la nature de la condition dans la boucle `while` en démarrant  $p$  et  $q$  de manière appropriée.

```

algorithme gradient_ascent
entrées
    •  $f(x, y)$ , une fonction différentiable donnée sous forme d'expression ou de fonction
        en  $x$  et  $y$  •
     $p$ , le point  $(x_0, y_0)$  •  $d$ ,
        le nombre de décimales de précision souhaité
    • , le facteur d'échelle d'une étape
sorties •
    une liste  $L$  des points le long du chemin approximatif de la montée la plus raide
faire
    convertir  $p$  en vecteur
    soit  $L = (p)$ 
    soit  $q = p$ 
    ajouter  $\rightarrow i$ 
    à  $p$  répéter tant que  $p$  et  $q$  sont plus éloignés que la précision souhaitée :
        soit  $p = q$ 
        soit  $q = p + \epsilon f(p)$ 
        ajouter  $q$  à la liste
    renvoyer la liste

```

Cela se traduit par le code Sage suivant :

```

sage : def gradient_ascent(f,p,eps,d) : p = vector(p).n()

L = [p]
q = pp
= p + vecteur([1,0]) grad =
f.gradient() tandis que
round(norm(pq), d) > 0 :
    p = qq
    = p + eps*grad(x=p[0], y=p[1])
    L.append(q)
retour L

```

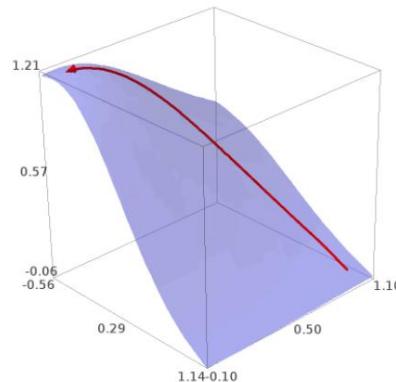
Ajoutons une visualisation de nos étapes le long du chemin. Si le chemin est une liste de points  $(x, y)$  issue de l'algorithme de gradient ascendant, nous pouvons représenter ces points sur le graphique 3D avec une flèche pointant vers le dernier point. Pour connaître les dimensions du graphique de  $f(x, y)$ , nous aurons besoin des valeurs minimales et maximales de  $x$  et  $y$  de tous les points du chemin. Une procédure d'aide pour trouver ces valeurs est fournie ici :

```
sage : def xyminmax(L) : xvals
= [pt[0] pour pt dans L] yvals = [pt[1]
pour pt dans L] xmin = min(xvals)
xmax = max(xvals) ymin =
= min(yvals) ymax =
max(yvals) renvoie
xmin, xmax, ymin, ymax
```

Dans notre visualisation, nous inclurons le tracé 3D de  $f(x, y)$  et les segments de droite fléchés. Nous étendons le graphique de  $f(x, y)$  pour inclure des informations complémentaires au-delà des valeurs minimales et maximales de  $x$  et  $y$ .

```
sage : définition de visual_gradient_ascent3d(f,p,eps,d) :
L = gradient_ascent(f,p,eps,d) xmin, xmax,
ymin, ymax = xyminmax(L) xrange = xmax - xmin
yrange = ymax - ymin g =
plot3d(f, (x, xmin - .1*xrange,
xmax+.1*xrange), \ (y, ymin - .1*yrange, ymax+.1*yrange), opacity=.5) g = g
+ line3d([(pt[0], pt[1], f(x=pt[0], y=pt[1])) \ pour pt dans L], couleur='rouge',
flèche=True, taille=30) show(g) q = L[-1] return q, f(x=q[0], y=q[1])
```

À partir de  $(1, 1)$  sur  $g(x, y) = 1+x^2+y^2$ ,  $\frac{1-x}{\sqrt{1+x^2+y^2}}$  L'algorithme constate que 25 étapes sont nécessaires pour une précision de 3 décimales avec 0,2. La valeur maximale est d'environ  $z = 1,2071$  à  $(-0,4141, 0,00069)$ , et le chemin a la forme suivante :



## Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. Les courbes paramétriques et les surfaces paramétriques peuvent être représentées graphiquement à l'aide de la même commande Sage.
  2. Chaque procédure graphique 3D dans Sage a une procédure 2D correspondante.
  3. En 3D, les graphiques sont combinés en les ajoutant.
  4. L'option `detect_poles=True` de Sage détectera l'asymptote dans  $f(x, y) = 3/x^2 + y^2 - 5$ . Si vous souhaitez représenter graphiquement  $\rho = \theta/\phi$  pour  $\pi/12 \leq \phi \leq \pi$  et  $0 \leq \theta \leq 6\pi$ , le moyen le plus simple est d'utiliser la commande `plot3d()`.
  6. La surface  $r = z \sin(2\theta)$  pour  $0 \leq z \leq 1$  et  $0 \leq \theta \leq 2\pi$  est la même que pour  $0 \leq \theta \leq 8\pi$ , donc le graphique de Sage de `plot3d(z*sin(2*theta), (theta,0,2*pi), (z,0,2), transformation=C)` est le même que celui de `plot3d(z*sin(2*theta), (theta,0,8*pi), (z,0,2), transformation=C)`, après avoir posé `C=Cylindrical('radius', ['azimuth', 'height'])`, 7. La meilleure façon de représenter graphiquement le paraboloïde  $y = x^2 + z^2$  consiste à isoler  $z$  en prenant la racine carrée, à tracer les branches positives et négatives séparément à l'aide de `plot3d()` et à les combiner à l'aide de l'addition.
  8. Si vous souhaitez voir les courbes de niveau de  $f(x, y)$ , utilisez `plot3d()` avec l' option `mesh=True` .
  9. Le domaine de  $f(x, y)$  est un sous-ensemble de  $\mathbb{R}^2$ .
- Si la fonction n'a pas de maximum local, l'algorithme de montée du gradient renvoie le point initial  $(x_0, y_0)$ .

Choix multiple.

1. Supposons que pts soit une liste de points 3D. Si vous souhaitez utiliser ces sommets pour tracer un polygone vide et délimité en 3D, que devez-vous faire ?
  - A. `polygone(pts, fill=False)`
  - B. `polygone(pts, couleur=Aucun, couleur des bords='vert')`
  - C. `ligne(pts+[pts[0]])`
  - D. demandez à votre instructeur
- instructeur 2. Combien de paramètres sont nécessaires pour définir paramétriquement des courbes tridimensionnelles ?
  - A. 1
  - B. 2
  - C. 3
  - D. 4
3. Combien de paramètres sont nécessaires pour définir paramétriquement des surfaces tridimensionnelles ?
  - A. 1
  - B. 2
  - C. 3
  - D. 4
4. Comparez un vecteur de gradient à la courbe de niveau dont il provient.
  - A. Le vecteur gradient est toujours tangent à la courbe de niveau.
  - B. Le vecteur gradient est toujours perpendiculaire à la courbe de niveau.
  - C. Le vecteur de gradient n'a aucun rapport avec la courbe de niveau.
  - D. Le vecteur de gradient a toujours la même couleur que la courbe de niveau.
5. L'erreur produite par `plot3d(sin(x*y)/(x*y), (x,5,-5), (y,-3,3))` est due à
  - A. la fonction étant indéfinie à l'origine
  - B. parenthèses mal placées
  - C. la région n'étant pas carrée

## D. l'inversion de xmin et xmax

6. Si vous tracez une surface 3D avec `plot3d()`, combien de paires de valeurs min et max font tu dois donner ?
- 1
  - 2
  - 3
  - 4
7. Si vous tracez un graphique d'une surface 3D avec `implicit_plot3d()`, combien de paires de valeurs min et max devez-vous fournir ?
- 1
  - 2
  - 3
  - 4
8. Pour modifier l'algorithme d'ascension de gradient pour faire une descente de gradient, nous devrions laisser
- $q = -p + f(p)$
  - $q = p + f(-p)$
  - $q = -p - f(p)$
  - $q = p - f(p)$
9. L'algorithme d'ascension du gradient pourrait être modifié pour trouver un maximum local de fonctions de combien de variables ?
- 1
  - 3
  - 4
  - Tous ces éléments
10. Laquelle des propositions suivantes décrit le mieux la signification du dégradé ?
- un nombre qui vous indique si vous êtes à un maximum local
  - un nombre qui vous indique s'il faut monter ou descendre
  - une fonction à valeur vectorielle qui peut être évaluée pour vous indiquer dans quelle direction aller pour y rester le même niveau
  - une fonction à valeur vectorielle qui peut être évaluée pour vous indiquer dans quelle direction aller pour monter le plus rapidement possible

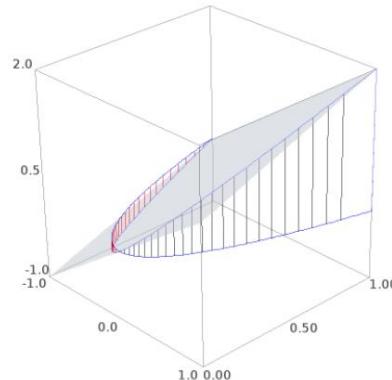
Réponse courte.

- Des couleurs différentes sont-elles nécessaires pour visualiser un champ vectoriel ou pourriez-vous toujours comprendre le champ vectoriel si tous les vecteurs étaient affichés dans la même couleur ?
  - Quels problèmes peuvent survenir avec l'algorithme de descente de gradient ? Explorez-les avec différentes fonctions et points initiaux. Comment pourriez-vous modifier le code pour éviter ces problèmes ? = 3 avec chacun
  - Tracez l'ellipsoïde  $2x^2 + y^2 + z^2 = 3$  avec les éléments suivants.  
`plot3d()` (b)  
`implicit_plot3d()` (c)  
`revolution_plot3d()` (d)  
`parametric_plot3d()`.
- Quelle méthode produit le meilleur graphique, même avec plus de points ? Quels problèmes rencontrez-vous avec les autres ?

Programmation.

1. Écrivez une procédure Sage qui représenterait graphiquement des segments de ligne avec des pointes de flèche sur tous les segments. Assurez-vous que cela fonctionne pour les collections de points 2D ou 3D.
2. Réécrivez l'algorithme de descente de gradient (sans tracé) de manière récursive.
3. Réécrire l'algorithme de descente de gradient graphique pour tracer sur la surface en utilisant `contour_plot()` plutôt que `plot3d()`. Inclure également le tracé du champ vectoriel de gradient dans la visualisation.
4. Réécrivez votre code du problème 3 pour afficher une animation sur le tracé de contour, où chaque image d'animation contient une étape de plus que l'image précédente.
5. Écrire une procédure Sage pour visualiser la région dont l'aire est calculée par une intégrale de droite par rapport à la longueur d'arc  $f(x, y)ds$ , dans le cas où la courbe  $C$  est paramétrée par  $x(t)$  et  $y(t)$  pour  $a \leq t \leq b$ . Le graphique doit être formé en combinant
  - la limite en bleu, y compris les lignes verticales du plan  $xy$  à  $f$  à  $x = a$  et  $x = b$  ainsi que la courbe  $C$ , tracée sur le plan  $xy$  et projetée sur la surface  $f(x, y)$ ,
  - l'intérieur de la région, non ombré, mais affiché à l'aide d'au moins 50 lignes verticales, montrant toute zone positive avec des lignes noires et toute zone négative avec des lignes rouges, et
  - le graphique de  $f(x, y)$  dans une couleur bleu alice semi-transparente .

Par exemple, avec 80 lignes verticales, `visualize_line_int(x+y, (t,t^2), -1, 1)` devrait produire un graphique comme celui ci-dessous.



## CHAPITRE 11

## Techniques avancées

Ce dernier chapitre présente deux techniques que l'utilisateur sérieux de Sage trouvera utiles, et probablement nécessaire, de temps en temps.

### Fabriquer ses propres objets

Sage inclut beaucoup de mathématiques. — Non, ce n'est pas vrai. Sage inclut

### Beaucoup de

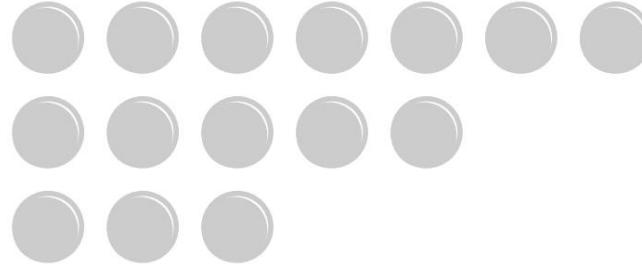
mathématiques. Vous n'aurez probablement jamais besoin d'utiliser quoi que ce soit qui ne soit pas dans Sage.

Cependant, certains mathématiciens travaillent parfois avec un groupe d'objets que Sage ne propose pas immédiatement. Si vous étudiez les jeux combinatoires, qui sont tout simplement géniaux<sup>1</sup>, Sage ne propose pas immédiatement de méthode pour effectuer ce que l'on appelle l'arithmétique Nimber<sup>2</sup>. Cette section présente l'arithmétique Nimber et montre comment créer votre propre type pour permettre à Sage d'effectuer l'arithmétique Nimber.

Contexte. L'idée de base de Nimbers est née d'un jeu nommé Nim, décrit pour la première fois dans Une revue mathématique du début du XXe siècle. Les règles sont simples :

- Le jeu se joue avec des cailloux disposés en rangées. • Chaque joueur peut prendre autant de cailloux qu'il le souhaite d'une rangée. • Le dernier joueur à retirer un caillou gagne.

Par exemple, supposons que David et Emmy décident de jouer à un jeu de Nim avec trois rangées de cailloux, où la première rangée a sept cailloux, la deuxième rangée en a cinq et la dernière rangée en a trois.

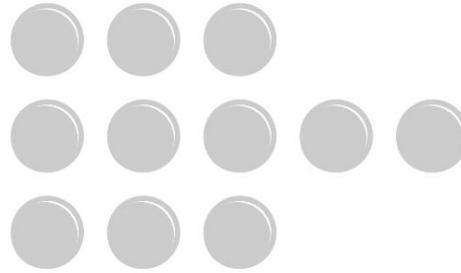


David commence ; supposons qu'il prenne quatre cailloux de la première rangée.

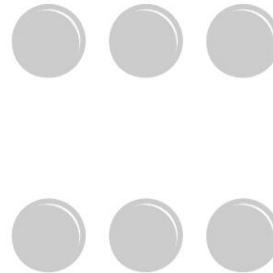
---

<sup>1</sup>Hackenbush, Nim, Chomp, Ideal Nim, Dots, Sprouts, ... voir [2] ou [1] pour plus d'informations que vous ne pouvez en trouver. 2

... selon la version utilisée par les auteurs au moment de la rédaction de ce document. L'inclusion de la [suite de jeux combinatoires](#) a été évoquée . ce qui résoudrait probablement ce problème en un clin d'œil et offrirait encore plus que des nombres : même des nombres surréalistes.



Ce n'était pas une décision très intelligente de la part de David, car Emmy peut prendre tous les cailloux de la deuxième rangée et quitter cette configuration.



David voit maintenant que quel que soit son mouvement sur une ligne, Emmy peut le reproduire sur l'autre ligne. Autrement dit, il a déjà perdu.

L'arithmétique de Número. L'idée de l'arithmétique de Número découle du constat suivant : une fois que le jeu se réduit à deux rangées visuellement symétriques, la partie est pratiquement terminée.

L'arithmétique fonctionne comme ceci :

- Le plus petit nombre est 0, suivi de 1, 2, .... • Étant donné un ensemble S de nombres, le plus petit nombre qui n'est pas dans S est appelé le minimum excluant, ou mex pour faire court.
- Pour additionner deux nombres, écrivez chacun comme une somme de puissances de 2, annulez les puissances identiques, puis simplifiez. • La soustraction est équivalente à l'addition. • Pour multiplier deux nombres m et n, trouvez  $\text{mex}\{\text{in} + \text{mj} + \text{ij} : i < m, j < n\}$ .

Ainsi, par exemple, l'addition fonctionne comme suit :

$$\begin{aligned}
 0 + x &= x \\
 1 + 1 &= 0 \\
 1 + 2 &= 3 \\
 2 + 2 &= 0 \\
 1 + 3 &= 1 + (1 + 2) = 2 \\
 2 + 3 &= 0 \\
 &= 2 + (1 + 2) = 1 \\
 3 + 3 &= 0
 \end{aligned}$$

tandis que la multiplication fonctionne comme ceci :

$$\begin{aligned}
 2 \times 3 &= \text{mex}\{0 \times 3 + 2 \times 0 + 0 \times 0, 1 \times 3 + 2 \times 0 + 1 \times 0, 0 \times 3 + 2 \times 1 + 0 \times 1, \\
 &\quad 1 \times 3 + 2 \times 1 + 1 \times 1, 0 \times 3 + 2 \times 2 + 0 \times 2, 1 \times 3 + 2 \times 2 + 1 \times 2\} = \text{mex}\{0, \\
 &\quad 3, 2\} \\
 &= 1.
 \end{aligned}$$

Notez que le deuxième mex ne laisse rien de côté : les sommes et les produits du premier mex sont des sommes et des produits de nombre, de sorte que (par exemple)

$$1 \times 3 + 2 \times 1 + 1 \times 1 = 3 + 2 + 1 = 0$$

Il est fastidieux de le faire à la main, c'est donc une bonne idée de l'automatiser dans Sage.

On pourrait bien sûr écrire une séquence de procédures exécutant directement les opérations, mais il est également possible de donner à Sage un moyen de traiter les nombres automatiquement et de les utiliser de manière naturelle. Par exemple, si nous créons les procédures `nim_add()` et `nim_mult()`, pour additionner et multiplier  $a \times b + c \times d$ , il faudrait taper

```
nim_add(nim_multi(a,b),nim_multi(c,d)) .
```

C'est difficile à comprendre. Il est beaucoup plus naturel de taper

$$a^*b + c^*d$$

et ensuite, tant que Sage reconnaît  $a$ ,  $b$ ,  $c$  et  $d$  comme des nombres, il effectue l'arithmétique automatiquement.

Utilisez une classe ! Pour ce faire, Sage utilise une classe. Une classe décrit le type d'un objet à l'ordinateur et lui indique les méthodes qui peuvent lui être envoyées.

Les classes nous permettent d'organiser les programmes autour des données et de leurs utilisations, et contribuent à centraliser les données. Dans Sage, les données associées à une classe, également appelées attributs, sont accessibles à l'intérieur de la classe grâce à la constante `self`, que nous devons écrire comme premier argument de chacune des méthodes de la classe. Vous le verrez dans un instant.

Pour créer une classe, utilisez le mot-clé `class` :

```
sage : classe ClassName :
#liste des procédures associées à cette classe
```

Comme le suggère le signe deux-points, les procédures associées à cette classe doivent être indentées. Lorsque l'indentation cesse, la classe cesse également. Chaque classe doit posséder au moins une méthode nommée `__init__()`, qui indique à Sage comment initialiser un élément de la classe. Cette méthode, appelée constructeur, est appelée silencieusement à chaque création d'un nouvel objet d'une classe. Ce nouvel objet est appelé instance de la classe.

---

<sup>3</sup>Nous n'avons pas réellement besoin d'utiliser le nom `self` ; nous pourrions utiliser `a` si nous le voulions, ou `this` ou `me`, mais `self` est la convention Sage hérite de Python.

<sup>4</sup>Il est également possible de l'appeler à nouveau après avoir créé un nouvel objet `a` en tapant `a.__init__(...)`, bien que cela doive être fait avec parcimonie, voire pas du tout.

Initialisation : Quelles informations sont propres à la classe ? Nous allons définir une classe nommée Nimer.

Un nombre est simplement un nombre dont l'arithmétique fonctionne différemment de la normale. Nous pouvons donc l'initialiser avec un nombre, qui devrait être un entier. Nous le testerons et lèverons une exception si ce n'est pas le cas. De plus, l'arithmétique des nombres est basée sur des puissances de 2 ; il serait donc judicieux de conserver une trace des représentations des nombres en termes de puissances de 2 ; autant l'intégrer au code d'initialisation. Notre méthode `__init__()` devrait donc initialiser deux éléments de données avec la classe Nimer :

- `self.value`, l'entier associé à cette valeur ; et
- `self.powers`, les puissances de 2 qui totalisent `self.value`.

Pour déterminer quelles puissances de 2 s'additionnent pour donner la valeur propre, nous allons vérifier si le nombre est divisible par 2, puis diviser par 2 pour soustraire les puissances, en ne conservant que le quotient. Par exemple :

- 11 n'est pas divisible par 2, donc 1 est une puissance de 2 dont la somme est 11. Si nous divisons  $11 = 1 + 2 + 8$  par 2, nous avons maintenant  $5 = 0 + 1 + 4$ . • 5 n'est pas divisible par 2, donc 1 est une puissance de 2 dont la somme est 5. Nous avions divisé par 2, donc  $1 \times 2 = 2$  est une puissance de 2 dont la somme est 11. Si nous divisons  $5 = 0+1+4$  par 2, nous avons maintenant  $2 = 0+0+2$ . • 2 est divisible par 2, donc 1 n'est pas une puissance de 2 dont la somme est 2. Nous avions divisé par 2 deux fois,  $= 4$  n'est pas donc  $1 \times 2^2$  une puissance de 2 dont la somme est 11. Si nous divisons  $0 = 0+0+2$  par 2, nous avons maintenant  $1 = 0 + 0 + 1$ .
- 1 n'est pas divisible par 2, donc 1 est une puissance de 2 dont la somme est 1. Nous avions divisé par 2 trois = 8 est une fois, donc  $1 \times 2$  ont<sup>3</sup> puissance de 2 dont la somme est 11. Si nous divisons  $1 = 0 + 0 + 1$  par 2, nous maintenant  $0 = 0 + 0 + 0$ .

Une fois que nous avons atteint 0, nous avons terminé, donc les puissances de 2 qui totalisent 11 correspondent aux divisions qui nous ont donné le reste 1 : c'est-à-dire 1, 2 et 8.

Cela suggère le code d'initialisation suivant.

```
sage : classe Numéro :
    def __init__(self, n):
        # vérifier le type valide si type(n) != Integer et type(n) != int ou n < 0 :
        generer ValueError( \
            « Les nombres doivent être des entiers non négatifs »
        )
        self.value = n
        self.powers = set() # trouver
        les puissances de 2 qui s'ajoutent à ni = 1

        tant que n != 0 :
            si n % 2 == 1 :
                self.powers.add(i)
            n = n // 2
            i = i * 2
```

Si vous travaillez avec une feuille de calcul, saisissez ceci dans une cellule. Nous ajouterons ultérieurement d'autres méthodes, que vous devrez saisir dans la même cellule. Si vous travaillez à partir de la commande

ligne, il est plus facile d'écrire ceci sous forme de script et de l'attacher ; au fur et à mesure que vous ajoutez d'autres méthodes à la classe et enregistrez le fichier modifié, Sage le rechargera automatiquement.

Une fois que nous avons tapé ce qui précède et l'avons exécuté ou attaché à Sage, comment créons-nous une instance Nember ? Nous utilisons le nom de la classe comme s'il s'agissait d'une procédure, dont les arguments sont appropriés pour être transmis à sa méthode `__init__()`.

```
(4) = 1/2 (5)
```

**ValueError : les nombres doivent être des entiers non négatifs**

Jusqu'ici tout va bien. Regardons ça de plus près.

```
sage : une
instance <__main__.Nember à 0x161c9e5a8>
```

Ce n'est pas un résultat très utile, n'est-ce pas ?

Représentations d'un objet. Pour un résultat plus exploitable, vous pouvez ajouter une méthode `__repr__()`.

La meilleure solution est probablement d'afficher `self.value`. Sage souhaite que `__repr__()` renvoie une valeur de chaîne, et vous pouvez convertir `self.value` en chaîne avec la commande `str()` :

```
sage : classe Numéro :
```

...

```
def __repr__(self) : renvoie
    str(self.value)
```

(N'oubliez pas que vous avez besoin de la définition `__init__()` dans l'espace indiqué par les points de suspension.)

Maintenant, lorsque nous initialisons un numéro, nous pouvons imprimer sa valeur :

```
sage : a = Nombre(4)
sage : un
4
```

D'un autre côté, il pourrait être utile de voir quelles puissances de 2 la classe a trouvées pour obtenir le nombre. Les méthodes `__init__()` et `__repr__()` sont appelées « noms de méthode spéciaux », mais il n'existe pas de nom de méthode spécifique approprié pour une représentation alternative et plus détaillée d'un objet. Pour cela, nous pouvons définir une méthode avec un nom de notre choix ; nous l'appellerons `.power_repr()`, et elle renverra simplement `self.powers`.

5

```
sage : classe Numéro :
```

...

```
def power_repr(self) : renvoie
    self.powers
```

---

Il n'est pas nécessaire d'entourer `power_repr` de traits de soulignement, car il ne s'agit pas d'une « méthode spéciale ». Le but de les soulignements servent à garantir que les gens ne redéfinissent pas accidentellement une méthode spéciale.

Nous pouvons maintenant voir comment cela fonctionne :

```
sage : a = Numéro(28)
sage : a.power_repr() {4, 8,
16}
```

Jusqu'ici, tout va bien.

Implémentation de l'arithmétique. Ce dont nous avons vraiment besoin, c'est d'un moyen d'automatiser l'addition et la multiplication des nombres, de préférence en utilisant des opérateurs arithmétiques ordinaires. Là encore, Sage propose des méthodes spécifiques pour cela ; nous utiliserons `__add__()` et `__mul__()`.

Mais comment allons-nous les mettre en œuvre ?

Pour l'addition, il suffit de choisir les puissances de 2 qui ne sont pas répliquées dans les puissances des deux nombres. Il existe une méthode simple : puisque nous avons conservé l'ensemble des puissances qui s'ajoutent au nombre, nous pouvons utiliser la méthode `.symmetric_difference()` sur un ensemble. Par exemple, les nombres  $14 = 2 + 4 + 8$  et  $20 = 4 + 16$  devraient s'ajouter à  $2 + 8 + 16 = 26$ . En appliquant la différence symétrique aux ensembles de puissances de 2, nous obtenons le résultat suivant :

```
sage : {2, 4, 8}.symmetric_difference({4,16}) {2, 8, 16}
```

L'addition est donc relativement simple à mettre en œuvre : renvoyer le nombre défini par la somme de la différence symétrique.

```
sage : classe Numéro :
...
déf __add__(self, b):
    renvoie Nombre(somme(
        self.powers.symmetric_difference( \ b.powers \
    )))
```

Testons cela sur notre exemple :

```
sage : Nombre(14) + Nombre(20)
26
```

Excellent!

La multiplication est un peu plus complexe, pour deux raisons. Premièrement, rappelons la définition de la multiplication numérique :

$$m \times n = \text{mex}\{\text{in} + \text{mj} + \text{ij} : i < m, j < n\}.$$

Cela nous oblige à calculer le mex d'un ensemble avec beaucoup de nombres que nous devons générer.

Construire l'ensemble n'est en fait pas si difficile : nous pouvons utiliser une compréhension de liste pour nous en occuper.

Cependant, Sage ne possède pas de procédure mex intégrée ; nous devons donc définir une procédure pour la calculer. Cette procédure ne devrait pas faire partie de la classe `Nimber`, car, bien que mex fonctionne sur les nombres, elle n'en est pas une propriété. Si vous le souhaitez, vous devriez en principe pouvoir trouver le mex d'un nombre.

ensemble d'entiers positifs sans modifier le code. Nous allons donc implémenter `mex()` comme une procédure distincte.

Comment implémenter cela ? Il faut trouver le plus petit nombre qui ne soit pas dans l'ensemble. Alors, pourquoi ne pas commencer avec  $i = 0$ , tester si  $i$  est dans l'ensemble, et augmenter et répéter si c'est le cas ? Cette approche fonctionnera :

```
sage : déf mex(S) : i = 0
tandis que Nimber(i) dans S :
    i = i + 1
renvoie le numéro (i)
```

Essayons un ou deux exemples rapides.

```
sage: mex([Nimber(0), Nimber(2), Nimber(4)]) sage: 1 sage: mex([Nimber(0),
Nimber(1),
Nimber(2)]) sage: 3
```

Les choses semblent bien se passer jusqu'à ce que vous regardiez d'un peu plus près :

```
0 = { 0, 1, 2}, 1, 2} 0 = { 0, 1, 2 , 2}
```

La réponse devrait être 3.

## PANIQUE!

...eh bien, non. Réfléchissons à ce qui pourrait en être la cause. La procédure `mex` doit déterminer si `Nimber(0)` est dans `S`. Vérifions `S` :

```
0 , 1, 1, 2,
2} sage : a , b =
Nimber(2), Nimber(2)
sage : a == a
Vrai
sage : a == b
FAUX
```

Que se passe-t-il ici ? Il y a deux problèmes : • Nous

rencontrons une réplication dans l'ensemble.

- Sage semble penser que `Nimber(2)` n'est pas égal à lui-même !

Ces deux problèmes proviennent du fait que nous n'avons pas indiqué à Sage comment déterminer si deux nombres sont égaux. Cela peut paraître évident, mais en réalité, ce n'est pas le cas ; l'égalité peut avoir deux significations :

- a et b sont le même objet (égalité), et • a et b sont des objets différents avec la même valeur (équivalence).

Le comportement par défaut de `==` est de vérifier le premier ; ainsi, `a == a` nous donne `True`, car nous comparons un objet à lui-même, tandis que `a == b` nous donne `False`, car nous comparons deux objets distincts.

De notre point de vue, bien sûr, il s'agit d'une distinction sans différence, comme le dit le dicton. Si nous voulons que Sage nous dise que deux nombres distincts sont effectivement égaux – autrement dit, si nous voulons que Sage nous indique quand deux nombres distincts sont équivalents – alors nous devons lui indiquer comment le déterminer.

Comparaison de Nimbres. Nous pouvons résoudre ce problème. Nous avons la technologie. Nous pouvons aussi résoudre le problème. Avec `mex`, en ajoutant des méthodes spéciales qui apprennent à Sage à comparer les nombres. Ce sont : `__eq__(self,other)`

vérifie si self et other sont identiques	<code>__ne__(self,other)</code>	vérifie si self et other sont
définitifs	<code>__lt__(self,other)</code>	vérifie si self est inférieur à other
self est inférieur ou égal à other	<code>__le__(self,other)</code>	vérifie si self est supérieur ou
égal à other	<code>__gt__(self,other)</code>	vérifie si self est supérieur à other

Nous devons définir les six, mais heureusement, ce n'est pas trop difficile, car nous pouvons simplement comparer les valeurs.

```
sage : classe Numéro :
...
déf __eq__(self, b):
    renvoie self.value == b.value
déf __ne__(soi, b) :
    renvoie self.value != b.value
déf __lt__(soi, b) :
    renvoie self.value < b.value
déf __le__(self, b):
    renvoie self.value <= b.value
déf __ge__(self, b):
    renvoie self.value >= b.value
déf __gt__(self, b):
    renvoie self.value > b.value
```

Cela nous permet au moins de faire ce qui suit :

```
sage : a, b, c = Número(2), Número(2), Número(3) sage : a == b
```

Vrai

```
sage : a == c
```

FAUX

Malheureusement, les ensembles ne fonctionneront toujours pas :

```
0 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
```

TypeError : instance impossible à hacher

Aïe. Qu'est-ce qu'une « instance hachable » ?

Un objet est hachable lorsqu'il est possible de l'associer à un entier (de type `int`, et non `Integer`) utilisé par Sage pour créer des ensembles et des dictionnaires. Cet entier est appelé hachage et ne doit jamais être modifié, car sa modification rendrait les ensembles et les dictionnaires étranges. Cela signifie que l'objet lui-même ne doit jamais être modifié. Ces deux critères s'appliquent aux `Nimbers`, car nous n'avons créé aucune méthode modifiant ses données, et la valeur d'un `Nimber` est un entier immuable. Pour créer ce hachage, il suffit de le configurer avec la méthode `__hash__()` :

```
sage : classe Numéro :
...
def __hash__(self):
    renvoie int(self.value)
```

... et maintenant les ensembles fonctionnent très bien :

```
0 , 1 , 2 }
```

Revenons à la multiplication. Maintenant que nous disposons d'ensembles de nombres fonctionnels, nous pouvons enfin multiplier des nombres. La définition de la multiplication de nombres est récursive ; il faut donc d'abord calculer les produits de paires plus petites. Notre récursivité nécessite des cas de base ; autant utiliser 0 et 1, car la définition de la multiplication de nombres montre que  $0 \times x = 0$  et  $1 \times x = x$  pour tout nombre  $x$  :

```
sage : classe Numéro :
...
déf __mul__(self, b):
    si self.value == 0 ou b.value == 0 :
        renvoie Number(0) si
    self.value == 1 :
        retour b
    elif b.value == 1:
        retour à soi-même
    autre:
        return mex({self*Nimber(j) + Nimber(i)*b \
                    + Nimber(i)*Nimber(j) \ pour i \
                    dans la plage(self.value) pour j \
                    dans la plage(b.value)})
```

Que diriez-vous d'une table de multiplication de `Nimber` ?

```
sage : pour i dans la plage(10) :
    pour j dans la plage(10) :
        print(Nombre(i) * Nombre(j)) print()
```

Vous n'aurez peut-être pas beaucoup de chance d'afficher plus de quelques lignes. Pourquoi ? Au-delà d'un certain point, la récursivité de la multiplication ralentit considérablement le processus. Sage possède un décorateur `@cached_method` qui devrait fonctionner comme le décorateur `@cached_function`, mais uniquement pour les méthodes de classe. Cependant, cela ne fonctionne pas dans ce cas, car le cache est lié à chaque objet et notre boucle crée constamment de nouveaux Numbers. Une meilleure approche serait de créer un cache global ; nous laissons cet exercice au lecteur.

Éléments omis. Nous avons omis plusieurs fonctions que les classes permettent de réaliser, notamment l'héritage et la surcharge.

L'idée de l'héritage est d'éviter la duplication de code en héritant automatiquement des méthodes d'un type qui s'appliquent également légitimement à un autre type. Par exemple, l'ensemble des nombres forme un champ ; ainsi, si un nombre héritait de la classe Field de Sage, il acquerrait automatiquement les méthodes prédefinies pour Field. L'inconvénient est qu'un champ est un objet plutôt abstrait. Ainsi, bien que Field fournisse de nombreuses méthodes, il n'en implémente pas la plupart, mais génère une erreur `NotImplementedError` lorsque vous tentez d'y accéder. Cela signifie que vous devez implémenter cela vous-même, ce qui n'est pas toujours simple. (Par exemple : comment trouver le multiplicatif inverse d'un nombre ?)

L'idée de la surcharge est de pouvoir implémenter une procédure avec différents types. Par exemple, vous pourriez vouloir multiplier un nombre par un entier standard, plutôt que de n'autoriser la multiplication que par des nombres. Dans ce cas, vous devrez modifier la procédure `__mul__()` pour tester ses entrées.

Dans d'autres cas, une méthode peut parfois prendre deux arguments, et d'autres fois trois arguments.

Ceci est accompli dans Sage en fournissant des valeurs par défaut pour certains arguments et en vérifiant les types d'arguments.

## Cython

Sage est basé sur Python, un langage interprété. Nous avons évoqué la différence entre le code interprété et les autres types de code à la page 13 ; interpréter du code implique essentiellement de traduire sans cesse des lignes.

De plus, Python s'appuie sur ce que l'on appelle le « typage dynamique ». Cela signifie que Python ne connaît pas le type d'une variable tant qu'il ne lui a pas attribué de valeur. Par exemple, dans la méthode `__mul__()` de Nimmer, Sage ne connaît pas le type de b tant qu'il n'exécute pas `__mul__()`. Cela signifie que Sage doit fréquemment vérifier le type d'une variable avant d'exécuter une opération ; après tout, il ne peut pas évaluer l'expression `b.value` sur un objet b dépourvu d'attribut nommé `value`.

L'autre approche majeure du typage utilisée en programmation est le « typage statique ». Dans cette approche, le programmeur doit déclarer le type de chaque variable avant l'exécution du programme. Cela peut être fastidieux pour le programmeur, mais cela évite à l'ordinateur de vérifier à chaque fois s'il doit réellement le faire. Cela peut entraîner des gains de vitesse considérables.

Cas test : la méthode de bisection. Pour voir l'impact que cela peut avoir, nous la représentons. notre procédure `method_of_bisection()` de la page 160 et discutons de la façon de la chronométrier.

```
sage : def méthode_de_bisection(a, b, n, f, x=x) : c, d = a, bf(x) = f pour i dans
la plage(n) :

# calculer le point médian, puis comparer e = (c + d)/
2 si (f(c) > 0 et f(e) >
0) ou (f(c) < 0 et f(e) < 0) :
    c = e
elif f(e) == 0:
    c = d = e
    casser
sinon :
    d = e
retour (c, d)
```

Rappelons que `a` et `b` sont des points finaux, `n` le nombre d'exécutions de la boucle, `f` une fonction et `x` une valeur indéterminée. Pour comprendre comment chronométrier le code, nous utiliserons une procédure spéciale appelée `%timeit`. Cette procédure répète plusieurs fois l'instruction qui la suit et renvoie le temps le plus rapide.

```
sage : %timeit method_of_bisection(1, 2, 40, x**2 - 2) 100 boucles, le meilleur des 3 :
17 ms par boucle
```

(Vous pourriez obtenir un nombre différent de 17 ms ; ce n'est pas grave, car cela dépend de nombreux facteurs autres que le programme : vitesse du processeur, vitesse du bus, etc.) Que s'est-il passé ? Sage a exécuté le programme en trois séries de 100 boucles, un peu comme ceci :

```
sage : pour i dans la plage (3) :
    pour j dans la plage (100) :
        méthode_de_bisection(1, 2, 40, x**2 - 2)
```

Pour chacun de ces trois ensembles, le temps nécessaire à l'exécution des 100 boucles a été enregistré, obtenant trois temps. Enfin, le meilleur de ces trois temps a été rapporté : 17 millisecondes. En résumé, une exécution de `method_of_bisection()` a duré environ 0,17 milliseconde, soit 170 microsecondes.

Pour voir l'effet de la compilation, saisissez ce qui suit, soit dans une cellule Sage, soit dans un script à joindre à la ligne de commande. (Si vous utilisez une feuille de calcul, saisissez `%cython` sur la première ligne. Si vous utilisez la ligne de commande, enregistrez le script avec le suffixe `.spyx` au lieu de `.sage`.) Ne vous souciez pas encore des modifications ; nous les expliquerons dans un instant.

```

de sage.symbolic.ring import SR de
sage.symbolic.callable import \
CallableSymbolicExpressionRing def
method_of_bisection_cython(a, b, n, f, x=SR.var('x')): c, d = a, b # f(x) = ff =
                                         
                                         
CallableSymbolicExpressionRing([x])(f) for i in range(n): #
calculer le point médian,
puis comparer e = (c + d)/2 si (f(c) > 0 et f(e)
> 0) ou \
(f(c) < 0 et f(e) < 0) :
c = e
elif f(e) == 0: c = d = e

casser
autre:
d = e
retour (c, d)

```

Lorsque vous exécutez cette cellule ou attachez ce script, Sage s'arrêtera un instant pour compiler le code.

Si vous avez fait une faute de frappe, vous verrez probablement une ou plusieurs erreurs répertoriées. Dans ce cas, examinez attentivement cette liste et essayez de corriger les erreurs. Une fois la compilation terminée (vous le saurez car Sage ne signale aucune erreur), vous pourrez la chronométrier comme précédemment.

sage : %timeit method\_of\_bisection\_cython(1, 2, 40, x\*\*2 - 2) **100 boucles, le meilleur**  
**des 3 : 16 ms par boucle**

Le code a légèrement été accéléré :  $16/17 \approx 94,1\%$  de la longueur, soit une accélération d'environ 5 %. On ne vous en voudra pas si cela ne vous impressionne pas. Il est possible de faire mieux dans certains cas ; le site web Cython en donne [un exemple](#).

Avant de décrire comment nous pouvons améliorer ce code, expliquons un peu les différences dans le code Cython à partir du code Sage original.

- Cython a

besoin que nous « importions » des objets que Sage fournit habituellement gratuitement.

Les instructions qui effectuent cette opération utilisent les mots-clés `from` et `import`. Nous avons dû importer l'anneau symbolique `SR` et un nouveau type curieux appelé `CallableSymbolicExpressionRing`, que nous aborderons plus loin.

- Cython a besoin que nous déclarions nos indéterminés — même `x` !  
... et pour déclarer nos indéterminés, nous avons besoin de la commande `var()`, qui est elle-même pas disponible immédiatement pour Cython ; nous devons y accéder en tant que méthode de `SR`.
- Cython a besoin que nous définissions les fonctions mathématiques d'une manière différente.

Lorsque Sage lit ce que vous saisissez dans une feuille de calcul, en ligne de commande ou dans un script Sage, il réécrit une partie de la forme pratique que nous utilisons vers une forme plus adaptée à ses bases Python. Prenons l'exemple de la déclaration d'une fonction ; comme vous pouvez le deviner d'après le code ci-dessus, la déclaration `f(x) = ...` est un raccourci pratique pour `f`.

= CallableSymbolicExpressionRing([x])(...). Un autre exemple est l'opérateur « carat »  $\wedge$  ; sa signification est totalement différente en Python, qui considère l'opérateur « double produit »  $^{**}$  comme le véritable opérateur d'exponentiation. C'est pourquoi nous avons toujours conseillé d'utiliser le double produit pour l'exponentiation. Cependant, si l'habitude nous prenait et que nous saisissions un carat, Sage l'accepterait silencieusement comme exponentiation, sauf s'il apparaît dans un script Cython.

Ces modifications sont donc nécessaires pour que le code soit compilé en Cython.

Nous pouvons résoudre ce problème. Nous avons la technologie. Rappelez-vous que nous avons mentionné que la vérification du type d'un objet perd beaucoup de temps. Et si nous assignions un type aux éléments les plus réutilisés ? Le code passe beaucoup de temps à calculer  $f(c)$  et  $f(e)$ . Il serait donc judicieux de les calculer une fois, puis de stocker leurs valeurs. Modifions le programme comme suit :

```
de sage.symbolic.ring import SR de
sage.symbolic.callable import \
CallableSymbolicExpressionRing def
method_of_bisection_cython(a, b, n, f, x=SR.var('x')): c, d = a, b # f(x) = ff =
    CallableSymbolicExpressionRing([x])(f) for i in range(n): #
        calculer le point médian,
        puis comparer e = (c + d)/2 fc, fe = f(c), f(e) si
        (fc > 0 et fe > 0)
        ou \ (fc < 0 et fe < 0):
            c = e
        elif fe == 0:
            c = d = e
            casser
        autre:
            d = e
        retour (c, d)
```

Cela nous donne une amélioration notable des performances.

```
sage : %timeit method_of_bisection_cython(1, 2, 40, x**2 - 2) 100 boucles, le meilleur
des 3 : 11,6 ms par boucle
```

C'est une amélioration d'un peu plus de 25 % par rapport à la version précédente, donc nous faisons mieux.

Mais ! — le lecteur attentif se plaindra — nous aurions pu précalculer ces valeurs sans Cython. Quelle différence cela fait-il ? C'est une différence positive, en effet. Nous ne montrerons pas le code ici, mais si vous modifiez `method_of_bisection()` pour précalculer ces valeurs, nous obtenons un timing du type :

```
sage : %timeit method_of_bisection(1, 2, 40, x**2 - 2) 100 boucles, le
meilleur des 3 : 12,7 ms par boucle
```

C'est toujours plus lent que la version Cython pré-calcul, mais c'est plus rapide que la première version Cython ! Encore une fois, cette légère amélioration pourrait vous paraître peu impressionnante.

C'est vrai, mais nous avons encore un tour dans notre sac.

Attribution de types statiques. Comme mentionné précédemment, Python est un langage typé dynamiquement.

Cython est plutôt un hybride. Comme nous l'avons vu plus haut, vous n'avez rien à saisir, mais vous pouvez saisir certaines informations. Vous allez voir que cela peut avoir un impact significatif.

En plus de calculer  $f(c)$  et  $f(e)$  une seule fois, essayons de leur assigner un type. Quel type devraient-elles avoir ? Voici quelques options courantes :

- int et long, qui correspondent aux types Python pour les entiers « plus petits » et « plus grands » ;
- float et double, qui correspondent aux types Python pour les nombres à virgule flottante « plus petits » et « plus grands » ;
- Integer et Rational, qui correspondent aux types Sage pour les entiers et les nombres rationnels.

Il existe de nombreux autres types, dont certains sont très utiles dans diverses circonstances. N'oubliez pas que vous pouvez généralement trouver le type d'un objet en demandant à sage : type(2/3), par exemple, nous donnera <type 'sage.rings.rational.Rational'>. Notez-le, nous y reviendrons dans un instant.

Si vous revenez à la présentation de la méthode de bisection, nous avons mentionné qu'il s'agit d'une sorte d'« approximation exacte », car elle spécifie un intervalle (donc approximation) sur lequel la racine se trouve certainement (donc exacte). Vous vous souviendrez que la méthode de bisection nous donne des fractions, ce qui exclut plus ou moins les nombres entiers (int), longs (long) et entiers (Integer). Devrions-nous alors utiliser float (flottant), double (double) ou rationnels (rationnels) ? N'importe lequel d'entre eux fonctionnera, et les deux premiers sont bien plus rapides que le troisième, mais les auteurs ont un faible pour les nombres exacts dans toute leur splendeur, longue et complexe ; nous opterons donc pour celui-ci.

Rappelons que la commande type() nous a indiqué qu'un Rational est sage.rings.rational.Rational. Parmi ces quatre mots, les trois premiers nous indiquent d'où nous importons, et le dernier nous indique ce que nous importons.

Nous allons ajouter une ligne à notre liste d'importation qui s'occupe de cela :

```
de sage.rings.rational cimport Rational
```

Si vous regardez attentivement, vous remarquerez que nous avons utilisé le mot-clé cimport au lieu du mot-clé import comme pour les deux autres. Ceci est dû au fait que nous importons un type implémenté comme classe dans un autre script Cython. Il n'est pas possible d'importer des types implémentés comme classes dans des scripts Sage ou Python.

Une fois cela fait, nous ajoutons une ligne au début du code définissant fc et fe comme rationnels. Nous devons ensuite forcer les calculs  $f(c)$  et  $f(e)$  à devenir rationnels, car ils sont par défaut de simples expressions. Voici toutes les modifications à apporter ; nous pouvons donc lister le code modifié ci-dessous :

```

de sage.symbolic.ring import SR de
sage.rings.rational cimport Rational de sage.symbolic.callable
import \ CallableSymbolicExpressionRing def
method_of_bisection_cython(a, b, n, f,
x=SR.var('x')): cdef Rational fc, fe c, d = a, bf = CallableSymbolicExpressionRing([x])
(f) for i in range(n): # calcule
le point médian,
puis compare e = (c + d)/2 fc, fe = Rational(f(c)), Rational(f(e))
si (fc > 0 et fe > 0) ou (fc
< 0 et fe < 0) :

    c = e
elif fe == 0:
    c = d = e
    casser
autre:
    d = e
retour (c, d)

```

Le timing de ce code est remarquablement plus rapide :

```

sage : %timeit method_of_bisection_cython(1, 2, 40, x**2 - 2) 100 boucles, le meilleur
des 3 : 6,27 ms par boucle

```

C'est presque la moitié du temps qu'il fallait auparavant, et certainement deux fois moins que le temps Python le plus rapide que nous ayons jamais vu. Au total, le code prend désormais environ un tiers du temps initial.

Et si on ajoutait ces informations de type au script Python ? — Oh, attendez. On ne peut pas.

Attention : ajouter des informations de type pour les variables restantes n'améliore pas significativement les performances et, dans certains cas, ralentit même le programme. (Nous avons vérifié.) Si vous jugez nécessaire d'écrire des scripts Cython, puis de les optimiser en ajoutant des informations de type, il est conseillé de profiler votre code pour identifier les tâches les plus longues. Nous ne donnons ici que quelques brèves indications.

Sage propose un outil de profilage appelé `prun()`. Il compte le nombre d'appels de chaque procédure, mesure le temps passé par Sage sur chaque procédure et détermine ensuite les procédures les plus pertinentes à calculer. Cet outil fonctionne aussi bien avec les procédures Sage/Python classiques qu'avec les procédures Cython.

Lors de la rédaction de ce chapitre, nous avons essayé `prun()` sur la méthode Cythonisée originale `method_of_bisection()` comme suit :

```

sage : prun(méthode_de_bisection_cython(1, 2, 30, f))

```

Sage a exécuté le programme et a immédiatement fourni une sortie sous la forme d'un tableau, chaque colonne indiquant •

- le nombre d'appels (ncalls) ; •
- le temps total passé dans la procédure (totime) ; •
- le temps moyen passé dans chaque appel (premier par appel) ; • le temps total cumulé passé dans cette procédure et toutes les procédures qu'elle appelle (cumtime) ; • le temps cumulé moyen passé dans chaque appel (deuxième par appel) ; et • l'emplacement de l'appel.

Lorsque nous l'avons exécuté pour la première fois, trois des quatre premières lignes indiquaient

- que Sage avait effectué • 106 appels à la méthode de substitution pour les objets sage.symbolic.expression.Expression ;
- 107 appels à callable.py:343(\_element\_constructor\_) ; et • 106 appels à callable.py:448(\_call\_element\_).

Sans avoir vu ces lignes auparavant, il n'était pas difficile de deviner que Sage consacrait beaucoup de temps au calcul de la valeur d'une fonction en un point. Cela nous a incités à essayer d'éviter de recalculer  $f(c)$  et  $f(e)$ . Une fois cela fait, le nombre d'appels à ces trois procédures a été réduit de plus d'un tiers, atteignant respectivement 60, 61 et 60.

Ceci étant dit, l'indice utile pour attribuer un type à quelque chose est qu'il y avait 209 appels à `{isinstance}`. Ceci, ainsi que de mystérieuses invocations de procédures dans le fichier `complex_interval_field.py`, indiquait qu'une importante vérification de type était en cours. Or, la version finale du programme ne rapporte qu'une seule innovation de `{isinstance}`, et aucun appel depuis `complex_interval_field.py`.

Déterminer précisément quelles variables nécessitent des informations de type relève de l'art, et les compétences s'acquièrent par l'expérience et les essais-erreurs. C'est pourquoi nous classons cette compétence, ainsi que la création de cours, dans la catégorie « Techniques avancées ». Si vous approfondissez vos connaissances dans le monde merveilleux des mathématiques, vous pourriez en avoir besoin un jour.

## Exercices

Vrai/Faux. Si l'affirmation est fausse, remplacez-la par une affirmation vraie.

1. La plupart des personnes qui travaillent avec Sage auront, à un moment donné, besoin de créer leurs propres calculs mathématiques.  
objets.
2. Les jeux combinatoires sont tout simplement géniaux.
3. L'addition de nombres s'annule d'elle-même ; c'est-à-dire que  $x + x = 0$  pour tout nombre  $x$ .
4. La multiplication de nombres utilise l'élément maximal d'un ensemble de nombres.
5. Une des raisons d'encapsuler Nimbers dans une classe est d'écrire les opérations arithmétiques de manière plus format pratique et naturel.
6. Les attributs d'une classe sont des variables qui incluent des données propres à la classe.
7. La modification des comportements de base d'une classe est soumise à des méthodes spéciales : `__init__()`, `__repr__()`, `__eq__()`, et ainsi de suite.
8. Dans Sage, un ensemble détermine automatiquement quand deux objets d'une classe sont équivalents.
9. Un hachage est un terme technique pour un objet dont la classe est trop compliquée pour être ajoutée à un ensemble.
10. Le dragon en colère de la récursivité vole dans ce chapitre et met le feu à au moins une méthode de classe.
11. Cython nous permet d'accélérer les programmes Sage en compilant et en attribuant des types statiques.
12. Les langages dynamiques connaissent le type de chaque objet avant l'exécution du programme ; les langages statiques restent agnostiques jusqu'à ce que la valeur soit attribuée.

13. Nous ne pouvons utiliser Cython qu'à partir de la feuille de calcul, en plaçant l' instruction %cython avant un bloc de code.
14. Nous pouvons « Cythoniser » le code Sage sans avoir à nous soucier de le modifier.
15. Nous ne pouvons pas importer de types implémentés en tant que classes dans les scripts Cython, uniquement des types natifs de Sage comme int et Rational.

Choix multiple.

1. Les méthodes utilisées pour initialiser les données propres à une classe sont appelées : A.  
attributs B.  
constructeurs C.  
initialiseurs D. créateurs
2. Le terme correct pour désigner une variable dont le type est de classe C est :
  - A. un attribut de C.
  - B. un constructeur pour C.
  - C. un descendant de C.
  - D. une instance de C.
3. La valeur de mex{1, 2, 4, 5, 7, 8, 9} est :
  - A. 0
  - B. 3
  - C. 6
  - D. 9
4. En arithmétique de Ninteger, la valeur de 3 + 10 est :
  - A. 0
  - B. 2
  - C. 9
  - D. 13
5. Pourquoi la multiplication de nombres est-elle difficile à faire à la main ?
  - A. cela nécessite une récursivité
  - B. c'est nouveau et différent C.  
l'auto-annulation de l'addition D. ce n'est pas difficile à faire à la main
6. La bonne façon de créer une nouvelle instance d'une classe nommée C, dont le constructeur prend un argument entier que nous voulons être 2, est de taper : A. a = C(2)
  - B. a = nouveau C(2)
  - C. a = nouveau(C, 2)
  - D. a = C.\_\_init\_\_(2)
7. Le terme pour les méthodes spécialement nommées que Sage attend pour certains comportements « naturels » d'un Les classes sont : A. les méthodes d'attribut B. les méthodes magiques C. les méthodes spéciales D. les méthodes de points
8. Un objet est hachable lorsque nous pouvons l'associer à une valeur de quel type ?
  - A. tout type bien ordonné B.  
hachage

C. Entier D.

int

9. Quel mot-clé nous permet d'attribuer un type à une variable en Cython ?

A. cdef B.

def C.

type D.

aucun mot-clé ; placez simplement le type avant le nom de la variable 10.

Quel(s) mot(s)-clé(s) utilisons-nous pour accéder aux objets et types Sage définis dans d'autres fichiers Cython ?

A. accès... depuis...

B. de... importer...

C. #include D.

**PANIQUE !**

Réponse courte.

1. Résumez pourquoi il peut être utile de créer une classe pour un nouvel objet mathématique, plutôt que d'implémenter des opérations sous forme de procédures avec les noms `my_object_add()`, `my_object_mul()`, etc.
2. Résumez les similitudes et les différences entre les attributs et les variables d'instance.
3. Bien que Cython compile les instructions Python en véritable code machine, le résultat ne peut être exécuté indépendamment d'un interpréteur Python (comme Sage). Pensez-vous que cela fasse de Cython un véritable langage compilé ? Pourquoi ?
4. La méthode de bisection peut être exécutée avec des types `float` plutôt que `rationnels`. Pensez-vous que cela la rendrait plus rapide ou plus lente ? Essayez et comparez le résultat à votre estimation.

Programmation.

1. Une autre méthode spécifique à une classe est `__nonzero__()`, qui permet à Sage de déterminer si une instance d'une classe est, disons, « non nulle ». Implémentez cette méthode pour `Nimber`, en renvoyant `True` si et seulement si `self.value==0`.
2. Il est un peu plus correct en Python que `__repr__()` renvoie une chaîne avec laquelle vous pourriez recréer l'objet ; c'est-à-dire que la séquence de commandes suivante doit réellement avoir la sortie donnée, pas seulement 2.

```
sage : a = Nombre(2)
sage : un
Numéro(2)
```

- (a) Modifiez la méthode `__repr__()` de `Nimber` pour qu'elle renvoie la chaîne « correcte ». Astuce : Vous pouvez joindre des chaînes par « addition ». (b) Python recommande d'utiliser la méthode `__str__()` pour renvoyer une représentation « informelle », « plus pratique » ou « concise » de l'objet. Cette valeur est renvoyée à chaque utilisation de la procédure `str()`. Ajoutez une méthode `__str__()` à la classe `Nimber` qui renvoie la valeur initialement écrite pour `__repr__()`. Une fois ce problème résolu, la classe devrait fonctionner comme suit :

```
sage : a = Nombre(2)
sage : un
2) sage : str(a)
2
```

3. Nous avons vu que la multiplication Número pouvait être améliorée en utilisant une variable de cache. Modifiez votre classe Número pour introduire une variable globale nommée `cached_multiplications`, initialisée comme un dictionnaire vide. Modifiez ensuite la méthode `__mul__()` de Número pour obtenir les résultats suivants :

- Soit `c=self.value` et `d=b.value` . • Si  $c < d$ , échangez les deux. • Essayez d'affecter à `result` la valeur associée à la clé  $(c,d)$  dans `cached_multiplications`. • Si cela génère une erreur, interceptez-la, effectuez la multiplication de la manière habituelle, affectez sa valeur à `result` et enregistrez-la dans le dictionnaire. • Enfin, renvoyez `result`.

Essayez maintenant de générer la table de multiplication qui a pris trop de temps dans le texte.

4. Il n'est pas déraisonnable d'imaginer que quelqu'un souhaite instancier un Número en utilisant un ensemble de puissances de 2, ce qui nous éviterait de devoir le faire calculer par l'ordinateur. Modifiez la procédure `__iter__()` afin qu'elle teste le type de `n`. Si `n` n'est pas un tuple, une liste ou un ensemble, elle procède comme précédemment ; sinon, elle convertit `n` en ensemble, attribue cette valeur à `self.powers`, puis calcule `self.value`. L'implémentation correcte devrait produire les résultats suivants (en utilisant notre implémentation originale de `__repr__()`, et non celle attribuée dans un exercice précédent) :

```
a = Nombre ({1, 4})
sage : un 5

sage : a.power_repr() {1, 4}
```

Bonus : ajoutez une vérification selon laquelle la collection ne contient que des puissances de 2, de sorte que l'entrée `{1, 3}` génère une `ValueError`.

## CHAPITRE 12

### LATEX utile

De nombreux passages de ce texte utilisent LATEX, et de nombreux exercices et laboratoires vous demandent de l'utiliser. Ce chapitre résume toutes les commandes nécessaires à l'exécution de ce texte, ainsi que quelques commandes supplémentaires pour expérimenter. Il ne s'agit pas d'une introduction générale à LATEX ; il existe de nombreux ouvrages bien plus pertinents, dont l'objectif est de présenter ce formidable outil.

#### Commandes de base

La figure 1 présente une liste pratique de commandes LATEX. Pour les utiliser dans une chaîne de texte, vous devez les entourer de signes dollar, par exemple `$x \in \mathbb{R}$`. Vous pouvez également utiliser `\(` et `\)` comme délimiteurs ; par exemple, `\(x \in \mathbb{R}\)`.

#### Délimiteurs

Si vous avez une expression complexe, vous voudrez peut-être des délimiteurs (parenthèses, crochets, etc.) de grandir avec elle. On le voit dans la différence entre

$$(x^{xxx}) \quad \text{et} \quad \begin{matrix} xxx \\ x \end{matrix}$$

Pour ce faire, placez la commande `\left` ou la commande `\right` immédiatement avant le délimiteur. Chaque `\left` doit correspondre à un `\right`, mais si vous n'en voulez qu'un, vous pouvez placer un point après l'autre pour indiquer que vous ne voulez rien.

La deuxième expression ci-dessus provient de la saisie :

```
\left( x^{xxx} \right) \right)
```

Vous pourriez obtenir l'intervalle  $[2^{\ln 5}, \infty)$  en tapant ce qui suit :

```
\left[ 2^{\ln 5}, \infty \right)
```

Cela aura l'air mieux que `[2^{\ln 5}, \infty)` (qui donne  $[2^{\ln 5}, \infty)$ ), car dans le premier les délimiteurs étirés pour correspondre à la hauteur des objets impliqués.

#### Matrices

Il est préférable de définir les expressions matricielles sur des lignes séparées ; nous pouvons le faire en utilisant les délimiteurs `\[` (« début de l'affichage mathématique ») et `\]` (« fin de l'affichage mathématique »).

Notation LATEX	concept représenté exemple	dans le regroupement	résultat
{... }	LATEX voir ci-dessous	racine	voir ci-dessous
$x^2$	carrée en	$x^2$	$x^2$
\sqrt	exposant	$\sqrt{x^2+1}$	$\sqrt{x^2+1}$
-	élément en	$x_{\text{le suivant}}$	$x_{\text{next}}$
\in \	indice de	S	$x \in S$
{... \}	un ensemble contenant ...	{1,5,7}	{1, 5, 7}
\frac{a}{b}	fraction de a sur b	$\frac{a}{b}$	$\frac{2}{5}$
\alpha, \beta, etc. \infty	Lettres grecques 2\pi infini	2\pi	2\pi
(-\infty,\infty)	fonctions correctement	(-\infty,\infty)	(-\infty,\infty)
\sin, \cos, etc.	formatées	$\sin(\frac{\pi}{6})$	$\sin(\frac{\pi}{6})$
\rightarrow, \leftarrow, etc.	flèches	$\lim_{x \rightarrow 2}$	$\lim_{x \rightarrow 2}$
\sum, \int, \prod	somme,	$\int_a^b f(x) dx$	$\int_a^b f(x) dx$
	integral, produit	$\sum_{i=1}^n f(x_i) \Delta x$	$\sum_{i=1}^n f(x_i) \Delta x$
\leq, \geq	$\leq, \geq$	$a \leq b$	$a \leq b$
\notin, \neq	/, =	$b \notin S$	$a / S$
\subset, \not\subset	,	$S \setminus T$	$S \setminus T$
\dots, \cdots	..., ...	$\mathbb{N} = \{1, 2, \dots\}$	$= \{1, 2, \dots\}$
\cap, \cup	intersection, union	$S \cap T \cup U$	$S \cap (T \cup U)$
\mathrm{...}	mettre en italique...	$\mathrm{...}$	suivant
\mathbb{...}	écrire... en « gras tableau »	$\mathbb{R}$	
\mathbf{...}	écrivez ... en gras	$\mathbf{F}$	de
\mathcal{...}	écrire ... en police calligraphique	$\mathcal{S}$	

FIGURE 1. Balisage LATEX utile pour Sage

Nous commençons une matrice en utilisant la commande `\begin{pmatrix}` et la terminons en utilisant la commande `\end{pmatrix}.` (Si nous ne voulons pas de parenthèses autour de la matrice, nous utilisons la paire de commandes `\begin{matrix}` et `\end{matrix}.)`

Nous spécifions les entrées de la matrice ligne par ligne. Les colonnes sont séparées par l'espacement (&), tandis que les lignes sont séparées par une double barre oblique inverse (\|).

Par exemple, nous pouvons obtenir la matrice

$$\begin{pmatrix} 2 \cos x & -1 & x \\ e^{2x} & & \\ x+1 & & \sin x^2 - 1 \\ 1-x & & e^{-2x} \\ & x & \end{pmatrix}$$

en tapant ce qui suit :

```
\[ \begin{pmatrix} 2 \cos x & -1 & x \\ e^{2x} & & \\ x+1 & & \sin x^2 - 1 \\ 1-x & & e^{-2x} \\ & x & \end{pmatrix} \]
```

## Partie 2

Laboratoires

## Prérequis pour chaque laboratoire

Les travaux pratiques de cette partie sont organisés selon le domaine général des mathématiques auquel ils s'appliquent le plus. Nous indiquons ici les chapitres du manuel nécessaires à la réalisation de ce travail pratique selon l'approche retenue. Si l'enseignant a une approche différente, les prérequis pourraient être moins stricts.

**Différents types de tracés** : Ce laboratoire nécessite le chapitre 2 sur les « Calculs de base » et le chapitre 3 sur « De jolies (et moins jolies) photos ».

**Cauchy, le cercle et la parabole croisée** : Ce TP nécessite le chapitre 2 « Calculs de base » et le chapitre 3 « Belles (et moins belles) images ». Il peut être un peu plus facile une fois que l'élève a terminé le chapitre 6 « Résolution d'équations ».

**Un quotient de différence important** : ce laboratoire ne nécessite que quelques calculs de base du chapitre 2.

**Illustrer le calcul** : Ce laboratoire nécessite le calcul, qui apparaît dans le chapitre 2 sur les « Calculs de base », et le traçage, qui apparaît dans le chapitre 3 sur les « Jolies (et moins jolies) images ».

**Règle de Simpson** : Ce laboratoire nécessite du calcul, qui apparaît dans le chapitre 2 sur « Calculs de base », des feuilles de travail interactives, qui apparaissent à la fin du chapitre 4 sur « Écrire vos propres procédures », et des boucles for, qui apparaissent dans le chapitre 5 sur « Se répéter avec des collections ».

**La méthode Runge-Kutta** : Ce laboratoire nécessite le calcul, qui apparaît dans le chapitre 2 sur « Calculs de base », et les boucles les plus élémentaires, qui apparaissent dans le chapitre 5 sur « Se répéter définitivement avec des collections ».

**Coefficients de Maclaurin** : Ce laboratoire nécessite le calcul, qui apparaît dans le chapitre 2 sur « Calculs de base », et les boucles for, qui apparaissent dans le chapitre 5 sur « Se répéter avec des collections ». Série

p : Ce laboratoire nécessite le traçage, qui apparaît dans le chapitre 3 sur « Jolies (et moins jolies) images », et les boucles définies, qui apparaissent dans le chapitre 5 sur « Se répéter avec des collections ».

**Passons à autre chose** : ce laboratoire nécessite le chapitre 2 sur « Calculs de base », le chapitre 3 sur « Jolies (et moins jolies) images » et le chapitre 6 sur « Résolution d'équations ».

**Maxima et minima en 3D** : Ce laboratoire nécessite la résolution d'équations, qui apparaissent au chapitre 6, la création de tracés 3D à partir du chapitre 10 et des dictionnaires, qui apparaissent au chapitre 5 sur « Se répéter avec des collections ».

**Propriétés algébriques et géométriques des systèmes linéaires** : Ce laboratoire nécessite l'algèbre linéaire, qui apparaît dans le chapitre 6 sur « Résolution d'équations ».

**Matrices de transformation** : Ce TP nécessite l'algèbre linéaire, abordée au chapitre 6 « Résoudre des équations », et le tracé de vecteurs, abordé au chapitre 3 « Des images belles (et moins belles) ». Il serait utile de connaître les boucles for, abordées au chapitre 5 « Se répéter avec des collections ».

**Visualisation des vecteurs propres et des valeurs propres** : Ce laboratoire nécessite l'algèbre linéaire, qui apparaît dans le chapitre 6 sur « Résoudre des équations », le tracé de vecteurs, qui apparaît dans le chapitre 3 sur « De jolies (et moins jolies) images », et les boucles for, qui apparaissent dans le chapitre 5 sur « Se répéter avec des collections ».

Moyenne des moindres carrés : ce laboratoire nécessite de l'algèbre linéaire, qui apparaît au chapitre 6 sur « Résoudre des équations » et le chapitre 3 sur « De belles (et moins belles) images ».

Méthode de Bareiss : Ce laboratoire nécessite l'algèbre linéaire, qui apparaît dans le chapitre 6 sur « Résoudre des équations », et les boucles for, qui apparaissent dans le chapitre 5 sur « Se répéter avec des collections ».

Une façon de répondre plus efficacement à la dernière partie serait d'utiliser les expressions try/except, qui apparaissent au chapitre 7 sur la « Prise de décision ».

Méthode de Dodgson : Ce laboratoire nécessite l'algèbre linéaire, qui apparaît dans le chapitre 6 sur « Résoudre des équations », et les boucles for, qui apparaissent dans le chapitre 5 sur « Se répéter avec des collections ».

Fonctions biunivoques : Ce laboratoire nécessite des dictionnaires, qui apparaissent dans le chapitre 5 sur « Re-se répéter définitivement avec des collections ».

Le jeu des ensembles : Ce laboratoire nécessite des collections, qui apparaissent dans le chapitre 5 sur « Répéter votre-moi-même définitivement avec des collections. »

Le nombre de façons de sélectionner m éléments dans un ensemble de n : Ce projet nécessite le chapitre 9 sur « Se répéter de manière inductive ».

Propriétés des anneaux finis : Ce laboratoire ne nécessite que quelques calculs de base du chapitre 2.

Ce serait plus simple avec les boucles for, qui apparaissent dans le chapitre 5 sur « Se répéter définitivement avec des collections », mais les nombres sont suffisamment petits pour que ce ne soit pas strictement nécessaire (attention cependant, vos élèves pourraient vous détester de l'avoir assigné avant de couvrir les boucles).

L'horloge chinoise du reste : ce laboratoire nécessite l'arithmétique modulaire, du chapitre 2 sur les « Calculs de base », et divers objets de tracé, du chapitre 3 sur les « Jolies (et moins jolies) images ».

Géométrie des racines radicales : Ce TP ne nécessite que quelques calculs de base du chapitre 2. Il serait plus simple avec les boucles for, présentées au chapitre 5, intitulé « Se répéter avec précision avec des collections », mais les nombres sont suffisamment petits pour que ce ne soit pas strictement nécessaire. (Contrairement à un autre TP, on ne devrait pas vous en vouloir si vous lui assignez ce TP avant d'aborder les boucles.)

Séquences de Lucas : Ce laboratoire nécessite la section « Les valeurs propres et les vecteurs propres résolvent un lapin » « Dilemme » du chapitre 9 sur « Se répéter de manière inductive ».

Introduction à la théorie des groupes : Ce laboratoire nécessite le chapitre 5 sur « Se répéter soi-même défini-« niely with collections » et le chapitre 7 sur « La prise de décision ».

Théorie du codage et cryptographie : la majeure partie de ce laboratoire peut être réalisée après le chapitre 9 sur « Se répéter de manière inductive », mais le bonus nécessite la section sur Cython du chapitre 11 sur « Techniques avancées ».

Fractions continues : Ce laboratoire nécessite le chapitre 5 sur « Se répéter indéfiniment » et le chapitre 7 sur « La prise de décision ».

## Mathématiques générales

### Polynômes irréductibles

Supposons qu'un polynôme possède des coefficients entiers. Il est parfois possible de le factoriser en deux ou plusieurs polynômes qui ont également des coefficients entiers :

$$2x^2 - x - 6 = (x - 3)(x + 2).$$

Parfois, cependant, il est possible de factoriser uniquement lorsque nous autorisons des coefficients irrationnels ou même complexes (vérifiez cela en appliquant la commande `expand()` de Sage au côté droit de l'équation) :

$$2x^2 - x + 1 = x - \frac{1 + i\sqrt{3}}{2} - x - \frac{1 - i\sqrt{3}}{2}.$$

Nous appelons ce deuxième type de polynôme irréductible.

Ce devoir étudie l'irréductibilité d'une famille simple, mais importante de polynômes. Soit

$$pn = x^n - 1,$$

où  $n$  est un entier positif. Autrement dit,

$$p1 = x - 1, p2 = x^2 - 1, p3 = x^3 - 1, \dots$$

Ces polynômes sont-ils irréductibles ? Non, car  $pn(1) = 0$ , donc, d'après le théorème des facteurs,  $x - 1$  est un facteur de  $pn$ .

Comment se factorise-t-on ? Rappelons la formule d'une série géométrique :

$$(1) \quad 2a + ar + ar^2 + \dots + ar^{n-1} = a \times \frac{r^n - 1}{r - 1}.$$

(Une série est « géométrique » si le rapport entre un terme et celui qui le suit reste le même, quel que soit le terme choisi. Dans la série ci-dessus, ce rapport est  $r$ .) Notez la partie droite le côté  $a r^{n-1} - 1$ ; c'est la même chose que  $pn(r)$ . Soit  $a = 1$  et  $r = x$ , de sorte que l'équation 1 devient

$$1 + x + x^2 + \dots + x^{n-1} = \frac{x^n - 1}{x - 1}.$$

Multipliez les deux côtés par  $x - 1$ , puis réorganisez un peu, et nous trouvons que

$$x^n - 1 = (x - 1)x^{n-1} + \dots + x^2 + x + 1.$$

Nous avons factorisé  $pn$  !

Ainsi,  $pn$  n'est pas irréductible lorsque  $n > 1$ . Nous tournons notre attention vers ses facteurs :  $x - 1$  est irréductible, mais qu'en est-il

$$qn = x^{n-1} + x^{n-2} + \dots + x + 1 ?$$

Les trois premiers exemples sont

$$q2 = x + 1 \quad (\text{irréductible})$$

$$2q3 = x^2 + x + 1 \quad (\text{irréductible : essayez le})$$

formule quadratique pour voir pourquoi)

$$q4 = x^3 + x^2 + x + 1 = (x + 1)x^2 + 1 \quad (\text{facteurs!})$$

Une fois que nous arrivons à  $q5 = x^4 + x^3 + x^2 + x + 1$ , nous commençons à penser qu'il serait peut-être bien que quelqu'un d'autre Faites le travail. C'est là qu'intervient Sage.

- Créez une nouvelle feuille de calcul. Intituez-la « Polynômes irréductibles ». Assurez-vous que votre nom apparaisse dans la feuille. une cellule de texte en haut de la feuille de calcul.

2. Revoyez les commandes `factor()` et `expand()` du chapitre 2 (pages 31-32). Utilisez-les pour vérifier les factorisations trouvées pour  $q_2$ ,  $q_3$  et  $q_4$ .
3. Utilisez la commande `factor()` de Sage pour déterminer quels  $q_n$  sont irréductibles.  
Astuce : pour des valeurs de  $n$  plus grandes, il peut être très difficile de taper  $x^{20} + x^{19} + \dots$  et ainsi de suite.  
Il est plus simple de factoriser  $x^{21} - 1$ .
4. Examinez vos résultats et formulez une conjecture quant aux valeurs de  $n$  qui rendent  $q_n$  irréductible.
5. Testez votre supposition :
  - Choisissez un nombre  $n > 100$  pour lequel vous pensez que  $q_n$  est irréductible et utilisez Sage pour vérifier si c'est.
  - Choisissez un certain nombre de  $n > 100$  pour lesquels vous pensez que  $q_n$  est un facteur et utilisez Sage pour vérifier si c'est le cas.
6. Rédigez un bref résumé de votre travail dans une cellule de texte à la fin de la feuille de travail. Utilisez LATEX.  
lorsque vous tapez des polynômes, afin qu'ils soient faciles à lire.

## Différents types de tracés 1.

Créez une nouvelle feuille de travail. Donnez-lui le titre « Labo : Différents types de tracés ». Ajoutez d'autres informations pour vous identifier, si nécessaire.

## Partie 1 : Intrigues implicites.

2. Sélectionnez un problème selon le schéma suivant.

Si votre identifiant se termine par... ... utilisez cette équation.	
0,1,2 + x = y	$2^2 - 3,4^2 + 5 + y \cdot 6^2 - 7^2 - 8,9^2$
	$2x = \frac{25}{4}xy^2$
	$4x^2 - x^4 + y^4 = 1$

3. Tout d'abord, utilisez au moins 300 points pour créer et afficher un tracé implicite de l'équation sur la région  $[-2,25, 2,25] \times [-2,25, 2,25]$ . La courbe peut être de la couleur de votre choix, à condition qu'elle soit noire.<sup>2</sup> 4.

Choisissez une valeur x dans l'intervalle  $[-2, 2]$ . En utilisant vos compétences en mathématiques (pas celles de Sage, ni celles de quelqu'un d'autre - vous pouvez me demander de l'aide), trouvez l'équation d'une droite tangente à la courbe en ce point. Écrivez l'équation de cette droite dans une cellule HTML. Utilisez les commandes LATEX de base pour un rendu impeccable. Remarque : il peut y avoir plusieurs valeurs y pour cette valeur x, donc plusieurs tangentes à la courbe pour cette valeur x, mais vous n'en tracerez qu'une seule. Ne choisissez pas un point où la tangente est horizontale ou verticale ; ce serait de la triche.

5. Vérifiez votre affirmation en combinant le tracé de la courbe avec celui de la tangente. La droite doit être rouge. Placez un point rouge à l'intersection de la droite et de la courbe. La taille du point doit être suffisamment grande pour qu'il soit bien visible ; la taille par défaut est insuffisante.

## Partie 2 : Graphiques paramétriques.

6. Une courbe de Bézier est une courbe paramétrique qui implique quatre points de contrôle  $(x_0, y_0)$ ,  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$  et les équations

$$\begin{aligned} x(t) &= x_0(1-t)^3 + 3x_1t(1-t)^2 + 3x_2t^2(1-t) + x_3t^3 \\ y(t) &= y_0(1-t)^3 + 3y_1t(1-t)^2 + 3y_2t^2(1-t) + y_3t^3 \end{aligned}$$

où  $t \in [0, 1]$ . Tracez une courbe de Bézier noire. Vous pouvez choisir les quatre points de contrôle au hasard, mais n'en utilisez aucun que vous avez déjà vu. N'hésitez pas à créer une boucle, si vous le pouvez. Ajoutez des lignes pointillées rouges qui relient les 1er et 2e points de contrôle  $(x_0, y_0)$  et  $(x_1, y_1)$ , et les 3e et 4e points de contrôle  $(x_2, y_2)$  et  $(x_3, y_3)$ , permettant au spectateur de voir comment les points de contrôle se rapportent aux lignes tangentes.

7. Utilisez votre graphique pour rédiger une brève description de la relation entre les points de contrôle et les lignes.

## Partie 3 : Animez !

8. Animer la courbe de Bézier :

- Modifiez les valeurs des 2e et 3e points 12 fois. (Ne les modifiez pas beaucoup, seulement un peu, pour que la courbe ne saute pas trop brusquement.)
- Pour chaque modification, créez un nouveau graphique. Chaque graphique doit être identique à celui créé à l'étape 6, sauf que les 2e et 3e points de contrôle sont différents. • Animez tous les graphiques à l'aide de la commande `animate()` et `show()`.

<sup>2</sup>Pas d'excuses à Henry Ford.

Partie 4 : Faites en sorte que cela soit beau !

9. Ajoutez des cellules HTML tout au long du document pour délimiter clairement les différentes parties. Le début et la fin de chaque partie doivent être clairement indiqués. Dans la partie 1, une cellule HTML doit indiquer précisément la valeur de  $x$  choisie ; dans la partie 2, les points de contrôle utilisés ; dans la partie 3, la séquence d'ajustements apportés aux deuxième et troisième points de contrôle. Vous devez également ajouter un texte explicatif avant chaque tracé : « Ceci est le premier tracé, avec des points... », « Ceci est le deuxième tracé, avec des points... », ... et enfin « Ceci est l'animation de toutes les images ». Utilisez LATEX pour les expressions et instructions mathématiques ; n'oubliez pas la référence aux commandes LATEX utiles à la page [251](#).

## Cauchy, le cercle et la parabole croisée

Laisser

$$f(x) = 2x^2 + 1. \text{ Créez un tracé de}$$

1

 $f(x)$  sur le domaine  $[-1, 1]$  et appelez-le p1.

2. Créez une image du demi-cercle dans les quadrants I et II qui est centré à l'origine, a un rayon 1, et est coloré en vert. Appelez- le p2.

3. Trouvez les coordonnées exactes des points d'intersection entre le demi-cercle et le graphique de  $f(x)$ . Il existe trois points de ce type :

- un sur l'axe des y, que nous appelons P0 ;
- un dans le premier quadrant, que nous appelons P+, avec les coordonnées  $x+, y+$  ; et
- un dans le deuxième quadrant, que nous appelons P-, avec les coordonnées  $x-, y-$ .

Astuce : Utilisez l'équation  $x = 1$  pour  $y \geq 0$ . Vous pouvez trouver cette équation avec ~~un crayon et du papier~~. Vous pouvez également la trouver avec Sage, mais vous devez afficher les commandes utilisées. Gardez à l'esprit qu'il peut être plus facile de résoudre ce problème avec un crayon et du papier.

4. Créez deux points rouges de taille 30, l'un à P+ et l'autre à P-. Combinez-les en un seul. image appelée p3.

5. Tracez la fonction  $y = x^2$  sur le domaine  $[-1, 1]$  et appelons-le p4.

Combinez p1, p2, p3 et p4 en un seul graphique. Assurez-vous que les points rouges se trouvent au-dessus des autres graphiques, et non en dessous.

7. Expliquez pourquoi les trois graphiques se rencontrent à P- et P+ .

Indice : élévez  $x+$  au carré et rationalisez toutes les fractions.

8. Considérez le rectangle dont le coin inférieur gauche se trouve sur l'origine et dont le coin supérieur droit se trouve à P+. Trouvez le rapport entre sa hauteur et sa largeur.

9. Le nombre d'or est le nombre

$$\varphi = \frac{\sqrt{5} - 1}{2}.$$

Exprimez votre réponse à la question n°8 en termes de  $\varphi$ .

## Calcul et équations différentielles

## Un quotient de différence important

Dans ce laboratoire, nous montrons comment vous pouvez utiliser Sage pour travailler sur la preuve que

$$\frac{d}{dx}(x^n) = nx^{n-1} \cdot dx$$

Créez une nouvelle feuille de calcul. Définissez le titre sur « Labo : Un quotient de différence important ».

2. Créez une cellule HTML. Inscrivez votre nom et le semestre en cours. Choisissez une couleur. Vous pouvez choisir la couleur de votre choix, à condition qu'elle ne soit ni noire ni blanche. Le blanc serait également mauvais.
3. Dans la première cellule de calcul, utilisez la commande var() pour définir les indéterminées x et h.
4. Dans les cellules de calcul suivantes, définissez  $f(x) = x^n$  et demandez à Sage de développer le produit pour  $(x + h)^n$  plusieurs valeurs concrètes de n. La commande expand() a été présentée à la page 32 . vous devez choisir plusieurs valeurs séquentielles de n : par exemple, n = 1, n = 2, et ainsi de suite.
5. Dans une cellule HTML suivant ces cellules de calcul, formulez une hypothèse sur la valeur des deux derniers termes de  $(x + h)$  et sur le facteur commun des termes restants. Autrement dit, nous avons

$$tg(x, h) + u + v \text{ où}$$

u et v sont les deux derniers termes rapportés par Sage, et t est le facteur commun des termes restants.

6. En calcul, on vous dit que f(x) est

- la définition de  $\frac{d}{dx}$

$$h \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h},$$

- et  $n \in \mathbb{N}$   $\frac{d}{dx} = nx^{n-1}$ .

Dans une dernière cellule HTML, expliquez pourquoi vos réponses aux étapes 4 et 5 démontrent ce fait. Astuce : utilisez ces informations pour résoudre le problème algébriquement. Vous n'avez pas besoin de connaître les détails de g(x, h) ; sur le papier, vous pouvez simplement remplacer f(x + h) par tg(x, h) + u + v, puis effectuer la soustraction, puis la division, et voir ce qui en ressort.

## Illustrer le calcul

- Créez une nouvelle feuille de travail. Intitulez-la « Laboratoire : Illustrer le calcul ». Ajoutez d'autres informations permettant de vous identifier, si nécessaire.
- Sélectionnez un problème selon le schéma suivant.

Si votre identifiant se termine par... ... utilisez cette fonction... ... sur cet intervalle.		
0,1,2	$f(x) = \sin x$	$-\frac{\pi}{3}, \frac{2\pi}{3}$
3,4,5	$(x) = \cos x$	$-\frac{\pi}{3}, \frac{3\pi}{3}$
6,7,8,9	$= \tan x$	$-\frac{\pi}{6}, \frac{\pi}{3}$

Partie 1 : Produits dérivés.

- Trouvez l'équation de la droite tangente à  $f$  en  $x = \pi/4$ . Tout calcul réalisable avec Sage devrait être évident dans votre feuille de calcul !
- Combinez les tracés de  $f$  et de la tangente à celle-ci sur l'intervalle donné. La courbe de  $f$  doit être noire et avoir une largeur de 2. La droite doit être bleue et avoir une largeur de 2.
- Créez une animation d'au moins 8 images montrant l'approche de la sécante vers la tangente lorsque  $x \rightarrow \pi/4$  en partant de la gauche. Réutilisez les tracés de  $f$  et de la tangente ci-dessus.  
Les lignes sécantes doivent être rouges et avoir une largeur de 1. Vous pouvez choisir les points de votre choix pour la sécante, à condition que  $x \rightarrow \pi/4$  en partant de la gauche. Une fois terminée, votre animation doit ressembler à celle du programme du cours : par exemple, la ligne sécante doit se dérouler en va-et-vient, et non dans une seule direction.

Partie 2 : Intégrales exactes.

- Calculez l'aire nette entre  $f$  et  $g(x) = 1 - x$  sur l'intervalle donné.<sup>2</sup>
- Combinez les courbes de  $f$  et  $g$  sur l'intervalle donné. Remplissez la zone entre  $f$  et  $g$ .  
Les courbes pour  $f$  et  $g$  doivent être noires, avec une largeur de 2. Le remplissage peut être de la couleur de votre choix, mais il doit être semi-transparent. Ajoutez un texte à l'intérieur du remplissage, qui contient la zone. Utilisez LATEX pour un texte élégant.

Partie 3 : Intégrales approximatives.

- Reprenez votre cours de calcul et révisez le calcul de la longueur d'arc avec des intégrales. Écrivez la formule et, dans une cellule HTML, expliquez brièvement quel outil de géométrie du lycée est utilisé pour la calculer.
- Utilisez Sage pour approximer la longueur de l'arc de l'ellipse  $x^2/4 + y^2/9 = 1$ . Limitez l'approximation à 5 points d'échantillonnage et arrondissez votre réponse à 5 décimales.
- Reprenez le problème 9, en limitant cette fois l'approximation à 10 points d'échantillonnage. Quelle partie de la réponse indique que vous avez une réponse plus précise ? Comparez-la à la valeur obtenue avec la fonction numerical\_integral() avec sa valeur par défaut (actuellement 87 points d'échantillonnage).

Partie 4 : BONUS ! (pour ceux qui disposent d'un temps et/ou d'une motivation exceptionnels). Animez des approximations de la longueur de l'arc, où chaque image montre

- pas d'axes ;
- l'ellipse, en noir, avec une largeur de courbe de 2 ;
- six cadres de 5 segments de ligne en pointillés, puis 6 segments de ligne en pointillés, ..., et enfin 10 segments de ligne en pointillés, en rouge, de largeur 1 ;

- une étiquette de texte dans chaque cadre avec l'approximation correspondante de la longueur de l'arc, au centre de l'ellipse, en noir.

Ce bonus vaut autant que le devoir entier. Si vous le souhaitez, vous pouvez effectuer la partie 4 au lieu des parties 1 à 3. Assurez-vous de bien maîtriser votre travail ; cela peut prendre un certain temps.

### La règle de Simpson

En calcul II, vous devriez avoir appris la règle de Simpson pour approximer la valeur d'une intégrale sur un intervalle.

1. Dans une cellule HTML,
  - (a) énoncer la formule de la règle de Simpson ;
  - (b) résumez brièvement l'idée qui donne naissance à la formule.
2. Écrivez une procédure Sage qui accepte comme entrées une fonction intégrable  $f$ , les extrémités  $a$  et  $b$  d'un intervalle et d'un entier positif  $n$ , puis utilise la règle de Simpson pour estimer  $\int_a^b f(x) dx$ .
3. Écrivez une procédure Sage qui accepte comme entrées une fonction intégrable  $f$ , les extrémités  $a$  et  $b$  d'un intervalle et d'un entier positif  $n$ , puis illustre la règle de Simpson avec un tracé des deux  $f$  et les figures dont les aires servent à estimer la valeur de l'intégrale. La procédure invoquer la solution à la question précédente et imprimer la zone sous forme d'étiquette LATEX quelque part sur la courbe.
4. Créez une procédure interactive pour résoudre la question précédente. En plus d'accepter  $f$ ,  $a$ ,  $b$  et  $n$  comme entrées, la procédure interactive doit permettre à l'utilisateur de sélectionner la couleur. pour les chiffres utilisés pour estimer la valeur de l'intégrale.

### La méthode Runge-Kutta

La méthode de Runge-Kutta pour approximer la solution d'une équation différentielle est similaire à la méthode d'Euler. Le pseudo-code suivant décrit Runge-Kutta :

```
algorithme Runge_Kutta
entrées
    • df , la dérivée d'une fonction • (x0 ,
      y0 ), valeurs initiales de x et y • Δx,
      taille du pas • n,
      nombre de pas à effectuer

    soit a = x0 , b = y0
    répéter n fois
        soit k1 = df (a, b)
        soit k2 = df (a + Δx/2, b + Δx/2 · k1 )
        soit k3 = df (a + Δx/2, b + Δx/2 · k2 )
        soit k4 = df (a + Δx, b + Δx · k3 )
        ajouter Δx/6 (k1 + 2k2 + 2k3 + k4 ) à
        b ajouter Δx
    à a retourner (a, b)
```

Implémentez Runge-Kutta dans le code Sage. Testez-le sur la même équation différentielle et initiale. condition que nous avons essayée dans la méthode d'Euler, et comparez le résultat.

## Coefficients de Maclaurin

En calcul III, vous devriez avoir étudié les développements de Maclaurin. Les coefficients de Maclaurin sont les coefficients numériques des termes du développement de Maclaurin.

1. Écrivez une procédure `maclaurin_coeffs()` qui accepte en entrée une fonction  $f$  et un entier positif  $d$ . Elle renvoie la liste des coefficients de Maclaurin de  $f(x)$  jusqu'au degré  $d$  inclus. Conseil : vous devriez probablement utiliser les procédures Sage `taylor()` et `coefficient()`.
2. Testez votre procédure avec des fonctions élémentaires connues telles que  $e^x$ ,  $\sin x$  et  $\cos x$ . Vérifiez que vos réponses correspondent à celles qui apparaissent dans un manuel de calcul.
3. Utilisez votre procédure pour trouver les coefficients de Maclaurin de  $f(x) = \arcsin 2(x)$  jusqu'au degré 20.
4. Observez les coefficients de Maclaurin non nuls de  $f(x)$  pour le degré 6 et supérieur. Quelle est la valeur commune ? Quel facteur voyez-vous dans les numérateurs ?
5. En utilisant votre réponse au problème précédent, modifiez votre invocation pour  
`maclaurin_coeffs(arcsin(???)**2, 20)` pour que tous les numérateurs soient 1. Enregistrez cette liste sous  $L1$ . Pourquoi est-il logique de procéder ainsi ?
6. À partir de  $L1$ , utilisez une compréhension de liste pour créer  $L2$ , une liste des réciproques des coefficients non nuls.
7. Quel est le plus petit facteur commun non trivial des éléments de  $L2$  ? Écrivez une autre liste de compréhension pour créer  $L3$ , qui contient les éléments de  $L2$ , mais divisés par ce facteur commun.
8. Consultez cette nouvelle liste dans l'[Encyclopédie en ligne des séquences d'entiers](#). Quelle séquence de nombres trouvez-vous ?
9. Revenez en arrière sur vos dernières étapes pour répondre au problème initial :

$$\arcsin 2 x = \sum_{k=1}^{\infty} ??? .$$

(Nous n'avons pas prouvé qu'il s'agit de la bonne série ; nous l'avons seulement conjecturée sur la base de quelques coefficients.)

## série p

Dans ce qui suit,  $x$  désigne le plafond de  $x$  et  $x$  désigne le plancher de  $x$ .

1. Écrivez le code pour calculer la somme

$$\sum_{k=1}^n \frac{1}{k^p},$$

et utilisez des boucles pour générer des listes des 50 premières sommes, pour  $p = 1, 2, 3$ .

2. Générez des tracés de ces sommes pour  $p = 1, 2, 3$  en traçant les fonctions en escalier

$$s_p(x) = \sum_{k=1}^x \frac{1}{k^p}$$

sur l'intervalle  $[1, 50]$ . À l'aide de vos tracés, devinez si la série p

$$\sum_{k=1}^{\infty} \frac{1}{k^p}$$

converge, pour  $p = 1, 2, 3$ .

3. Générez des tracés des séquences pour  $p = 1, 2, 3$  en traçant les fonctions

$$f_p(x) = \frac{1}{x^p}, g(x) = x \frac{1}{p}, h_p(x) = \frac{1}{x^p}$$

ensemble (en utilisant des couleurs différentes) sur l'intervalle  $[1, 10]$ .

4. Que suggèrent ces intrigues à propos de

$$\sum_{k=2}^n \frac{1}{k^p}, \quad \sum_{k=1}^n \frac{1}{k^p} \text{ et } x^p, \quad \sum_{k=1}^{n-1} \frac{1}{k^p},$$

et que suggèrent-ils à ce sujet

$$\sum_{pk=1}^{\infty} \frac{1}{p^k} \text{ et } \sum_{x=1}^{\infty} \frac{1}{x^p} dx ?$$

5. En utilisant vos réponses au problème 4, déterminez si oui ou non

$$\sum_{k=1}^{\infty} \frac{1}{k^p}$$

converge, pour  $p = 1, 2, 3$ .

## Sur une tangente

Laisser

$$f(x) = 2 + x - \frac{1}{x}$$

1. Tracez la fonction  $f(x)$  sur  $[-3, 3]$ .
2. Définissez dans Sage la fonction mathématique  $L(a, x)$  comme étant la droite tangente à  $f$  en  $x = a$ .  
Remarque : il doit s'agir d'une fonction à deux variables.
3. Tracez  $f(x)$  et  $L(a, x)$  ensemble pour chaque  $a \in \{1, 1.5, 2\}$ .
4. Notez que chaque droite coupe  $f$  deux fois. Notons  $Q_a(x)$  la deuxième intersection. Utilisez Sage pour trouver les coordonnées de  $Q_a$  en fonction de  $a$ .  
Indice : Résolvez l'équation  $L(a, x) = f(x)$  pour  $x$ . Une solution doit être  $x = a$ . L'abscisse de  $Q_a$  sera l'autre solution de cette équation. Appelez-la  $h(a)$ . Alors  $Q_a = (h(a), f(h(a)))$ .
5. Soit  $F(a)$  l'aire entre le graphique de  $f(x)$  et  $L(a, x)$ .
6. Utilisez Sage pour trouver une formule explicite pour  $F(a)$  en fonction de  $a$ . Simplifiez-la autant que possible.
7. Soit maintenant  $a_0 = 3$  et faisons un tracé qui inclut les points  $P = (a_0, f(a_0))$  et  $Q_{a_0}$ , ainsi que les fonctions  $f(x)$  et  $L(a_0, x)$  sur le domaine  $[-3, 3]$ .
8. Calculez la valeur exacte de  $F(a_0)$  avec Sage. Simplifiez le résultat autant que possible.  
Remarque : recherchez la valeur exacte, pas une approximation décimale.

## Croisement à un angle Dans

« Écrire vos propres procédures », nous avons appris à définir une procédure Sage qui calcule la tangente à  $f(x)$  à  $x = a$ . Ensuite, à la page 104, il vous a été demandé d'écrire le code d'une procédure qui renvoie la droite normale à  $f(x)$  à  $x = a$ .

On peut considérer la tangente comme la droite qui fait un angle  $\theta = 0$  avec la courbe de  $f(x)$  en  $x = a$ , et la normale comme la droite qui fait un angle  $\theta = \pi/2$ . Dans ce laboratoire, nous généraliserons la construction et définirons la droite qui fait un angle  $\theta$  arbitraire avec la courbe de  $f(x)$ .

Supposons que  $f(x)$  soit différentiable en  $x = a$ . Soit  $L\theta(a, x)$  la droite passant par  $(a, f(a))$  qui fait un angle  $\theta$  avec la droite tangente à  $f(x)$  en  $x = a$ . Ainsi,  $L0(a, x)$  est la droite tangente, et  $L\pi/2(a, x)$  est la ligne normale.

1. Trouvez une formule explicite pour la pente de  $L\theta(a, x)$ .

Indice : Dessinez une image. Interprétez la dérivée  $f'(a)$  comme la valeur d'une fonction trigonométrique, puis utilisez une identité trigonométrique.

2. Écrivez une procédure Sage qui renvoie  $L\theta(a, x)$ .

Remarque : Ce problème n'est pas aussi simple que de remplacer  $m = f'(a)$  dans le code de la ligne tangente de la page 91 par la réponse que vous avez trouvée dans l'élément précédent, car votre procédure doit également fonctionner lorsque  $\theta = \pi/2$  ou lorsque  $\tan(\theta)f'(a) = 1$ . 3. Maintenant, soit  $f(x) = x$  than  $(a, f(a))$ . Trouver toutes les valeurs de  $a$  pour lesquelles  $L\theta(a, x)$  intersectera  $f(x)$  en un point autre

Indice : Laissez Sage faire le travail. Résolvez l'équation  $L\theta(a, x) = f(x)$  et voyez quelles valeurs de  $a$  forment sens.

4. Pour chaque  $a$  trouvé dans l'élément précédent, soit  $F(a)$  l'aire entre  $f(x)$  et  $L\theta(a, x)$ . Utilisez Sage pour trouver une expression explicite pour  $F(a)$ . Votre expression doit être une expression en  $\theta$  et  $a$ , et aussi simplifiée que possible.

Astuce : utilisez `factor(expand(...))`.

5. Trouvez la valeur exacte de la valeur minimale possible de  $F(a)$  (pas d'approximations décimales).

6. Trouvez maintenant la valeur minimale de  $F(a)$  lorsque  $\theta = 0, \theta = \pi/6, \theta = 4$  et  $\theta = \pi/2$ .

## Maxima et Minima en 3D

1. Écrivez une procédure nommée `critical_points()` qui prend en entrée une fonction  $f$  deux fois différentiable en deux variables  $x$  et  $y$ , et renvoie une liste de points  $(x_i, y_i)$  qui sont des points critiques de  $f$ . Assurez-vous de ne trouver que des points critiques à valeur réelle ; vous pouvez utiliser la procédure `imag_part()` pour tester chaque point.
2. Écrivez une deuxième procédure, `second_derivs_test()`, qui accepte en entrée une fonction  $f$  deux fois différentiable dans deux variables  $x$  et  $y$ , ainsi qu'un point critique  $(x, y)$ , et renvoie l'un des éléments suivants : • 'max' si le point est un maximum ; • 'min' si le point est un minimum ; • 'saddle' si le point est un point selle ; et • 'inconclusive' si le test de dérivée seconde n'est pas concluant.
3. Écrivez une troisième procédure, `plot_critical_points()`, qui accepte en entrée une fonction  $f$  deux fois différentiable dans deux variables  $x$  et  $y$ , et renvoie un tracé tridimensionnel des éléments suivants :
  - $f(x, y)$  dans la couleur par défaut ;
  - maxima locaux en rouge ;
  - minima locaux en jaune ;
  - points de selle en vert ;
  - autres points critiques en noir.Utilisez un dictionnaire pour trouver la couleur appropriée pour les points. Utilisez les procédures `max()` et `min()` de Sage pour déterminer les valeurs  $x$  et  $y$  minimales et maximales du graphique afin de vous assurer que tous les points critiques apparaissent.

## Algèbre linéaire

## Propriétés algébriques et géométriques des systèmes linéaires

Un système d'équations.

- Sage dispose d'une procédure randint() qui choisit des nombres apparemment aléatoires. Son fonctionnement est le suivant : randint(m,n) donne un entier compris entre m et n. Utilisez cette commande pour générer six entiers aléatoires distincts a, b, c, d, e, f compris entre -20 et 20, puis substituez-les dans le système d'équations suivant. (Rappel : « distinct » signifie qu'il n'existe pas deux entiers identiques.)

$$\begin{aligned} ax + par &= ecx \\ + dy &= f \end{aligned}$$

- Tracez chaque équation sur le plan xy. Une droite est bleue et l'autre est rouge. (Peu importe laquelle.) Les deux droites ne seront vraisemblablement pas parallèles, mais se couperont en un seul point. Ajustez les axes x et y pour que cette intersection soit visible. Si les droites sont parallèles ou coïncidentes, modifiez l'une des valeurs a, b, ..., f pour qu'elles se coupent en un seul point.
- Utilisez Sage pour trouver la solution exacte du système d'équations. Il doit s'agir d'un point (x0 , y0 ). Créez un nouveau tracé avec les deux lignes (toujours de couleurs différentes) et un gros point jaune au-dessus de leur intersection. (Pas trop gros, mais suffisamment gros pour être vu.) Assurez-vous que le point se trouve au-dessus des lignes.

Un invariant.

- Définir les indéterminées a, b, ..., f. Les utiliser pour définir le système d'équations suivant, sans aucun nombre :

$$\begin{aligned} ax + par &= ecx \\ + dy &= f \end{aligned}$$

- Résolvez le système ci-dessus pour x et y.
- Vous avez peut-être remarqué que la solution a un dénominateur commun. (Si vous ne l'aviez pas remarqué, c'est le moment de le remarquer.) Qu'y a-t-il d'étonnant, mais pas tant que ça, dans ce dénominateur ?

Indice : Réfléchissez à quelques opérations matricielles de base sur la matrice correspondant aux membres gauches des équations originales. C'est quelque chose que vous auriez dû calculer en algèbre au lycée, et que vous auriez certainement calculé en algèbre linéaire.

- Supposons que a et b aient des valeurs concrètes connues. Utilisez votre réponse à la question n° 6 pour expliquer pourquoi l'existence d'une solution dépend entièrement de c et d — et n'a rien à voir avec e et f !

Indice : Je m'interroge sur l'existence d'une solution, et non sur sa valeur spécifique une fois qu'elle existe. La valeur spécifique dépend certes des valeurs e et f, mais son existence ne dépend que des valeurs c et d. La question vous demande donc d'utiliser la réponse précédente pour expliquer cette question d'existence, et non de valeur.

- Soit g le dénominateur de la solution trouvée dans l'équation n° 6. Remplacez les valeurs de a et b de l'équation n° 1 dans l'équation g = 0. Vous avez maintenant une équation linéaire à deux variables, c et d. Changez c en x et d en y.
- Tracez la droite déterminée (sans jeu de mots) par cette équation, dont l'ordonnée à l'origine est 0. Tracez également l'équation d'origine : ax + by = c. Quel est le lien entre ces deux droites ? (Ceci peut ne pas être clair, sauf si vous utilisez l'option de tracé aspect\_ratio=1 pour le clarifier.)

10. Cette relation entre les deux droites serait-elle valable quelles que soient les valeurs de a et b ? Autrement dit, si les seules valeurs modifiées étaient a et b, à la fois sur la droite et dans la fonction  $g(x)$ , la droite  $ax + by = c$  aurait-elle toujours la même relation avec la valeur correspondante de  $g(x)$  ?

Un point à l'infini.

11. Revenons à votre système d'équations initial. Définissez les listes D et E de telle sorte que les valeurs de D se déplacent en 10 pas de d presque à a, et que les valeurs de E se déplacent en 10 pas de e presque à b. (La différence entre les valeurs finales et a ou b doit être minime.) Par exemple, si votre système initial est

$$1x + 2y = 3$$

$$4x + 5y = 6$$

alors vous pourriez avoir quelque chose comme  $D = (4, 3, 2, 1, 5, 1, 1, 1, 01, 1, 001, 1, 0001, 1, 00001, 1, 000001)$  et  $E = (5, 4, 3, 2, 5, 2, 1, 2, 01, 2, 001, 2, 0001, 2, 00001, 2, 000001)$ .

12. Parcourez les listes D et E pour créer un tracé pour chaque système  $ax +$

$$by = c \text{ di } x +$$

$$ei y = f \text{ (ici, di)}$$

et ei font référence aux ième éléments de D et E, respectivement.) Combinez les tracés dans une animation séquentielle. Animez-la.

13. Décrivez la relation éventuelle entre les lignes, en particulier si vous laissez  $di \rightarrow a$  et  $ei \rightarrow b$ .

14. La géométrie projective introduit un nouveau point tel que toutes les droites, même parallèles, se coupent au moins une fois. Utilisez l'animation pour expliquer pourquoi il est approprié d'appeler ce point « point à l'infini ».

Astuce : vous souhaiterez peut-être ajuster les valeurs x et y minimales et maximales de votre animation pour voir cela plus clairement.

## Matrices de transformation

Définissons deux variables symboliques a et b.

$$A = \begin{matrix} \text{Soit cos } a \\ \text{sina} \end{matrix} \quad \text{et } B = \begin{matrix} \cos b \sin b \\ -\sin b \cos b \end{matrix}$$

1. Calculez AB dans Sage. Utilisez vos connaissances en trigonométrie pour spécifier une forme plus simple que celle proposée par Sage. Utilisez LATEX pour écrire cette forme simplifiée dans une zone de texte sous le calcul de AB.
2. Extrayez et donnez un nom à l'entrée de la première ligne et de la première colonne de AB.

celui-ci ni                  pas celui-ci  
celui-ci certainement pas celui-ci Essayez de

lui donner un nom significatif, pas seulement x, ce qui est une mauvaise idée, de toute façon.

3. Dans une nouvelle cellule de calcul, saisissez le nom que vous venez de créer pour cette entrée. Ensuite, saisissez un point. Appuyez ensuite sur Tab. Recherchez parmi les noms qui s'affichent une commande permettant de simplifier l'expression trigonométrique. Utilisez cette commande pour confirmer votre résultat dans la partie (b).
4. Définissez une nouvelle matrice, C, obtenue en substituant la valeur  $a = \pi/3$  dans A.
5. Soit v le vecteur défini par  $(5, 2)$ .
6. Calculez les vecteurs Cv, C 2v, C 3v, C 4v, C 5v.
7. Tracez les vecteurs v, Cv, C 2v, C 3v, C 4v, C 5v en différentes couleurs (au choix). Représentez-les sous forme de flèches ou de points, mais pas comme des fonctions en escalier. Combinez-les en un seul tracé, plutôt que six tracés différents.
8. Quel est l'effet géométrique de la multiplication répétée de v par la matrice C ? À quoi ressemblerait, selon vous, C 60v ? Et C 1042v ?

Visualisation des valeurs et vecteurs propres. Sage dispose

d'une procédure `randint()` qui choisit des nombres apparemment aléatoires. Son fonctionnement est le suivant : `randint(m,n)` donne un entier compris entre  $m$  et  $n$ . Utilisez cette commande pour générer quatre entiers aléatoires distincts compris entre 0 et 10, puis utilisez-les pour définir une matrice  $M$   $2 \times 2$ . (N'oubliez pas que « distinct » signifie qu'il n'y a pas deux entiers identiques.)

1. Utilisez Sage pour trouver les valeurs propres et les vecteurs propres de  $M$ . Il devrait y avoir deux vecteurs propres distincts, mais il est possible que vous n'en obteniez qu'un seul, avec une multiplicité de 2. Dans ce cas, modifiez très légèrement les entrées de votre matrice afin qu'elle produise deux vecteurs propres distincts.

2. Extrayez les vecteurs propres et nommez-les  $v_1$  et  $v_2$ .

Astuce : pour un crédit complet, extrayez-les à l'aide de l'opérateur parenthèse ; ne définissez pas de nouveaux vecteurs.

3. Soit  $K$  la liste des vecteurs obtenus à partir des 4 produits  $M_i v_1$  pour  $i = 1, 2, \dots, 4$ . Utilisez une boucle `for` pour définir cette liste ; vous perdrez la grande majorité des points sur cette partie si vous les faites un par un.

Astuce : soyez prudent. L'expression `range(10)` commence à 0 ; vous souhaitez commencer à 1.

4. Soit  $L$  la liste des vecteurs obtenus à partir des 4 produits  $M_i v_2$  pour  $i = 1, 2, \dots, 4$ . Utilisez une boucle `for` pour définir cette liste ; vous perdrez la grande majorité des points sur cette partie si vous les faites un par un.

5. Définissez  $p=Graphics()$  et  $q=Graphics()$ . (Cela définit un tracé « vide ».)

6. Utilisez une boucle `for` pour ajouter un tracé de chaque vecteur de  $K$  à  $p$ . Le premier vecteur doit être rouge ; les vecteurs suivants doivent se décaler progressivement vers le bleu, jusqu'à ce que le dernier soit complètement bleu. Voici comment je suggère de passer du rouge au vert :

```
for i in range(len(L)): p += plot(L[i],color=((10-i)/10,i/10,0),zorder=-i)
```

7. Utilisez une boucle `for` pour ajouter un tracé de chaque vecteur de  $L$  à  $q$ . Les vecteurs doivent passer du violet (rouge et bleu) au jaune (rouge et jaune).

8. Affichez  $p$  dans une cellule ; affichez  $q$  dans une autre.

9. Décrivez comment le résultat est cohérent avec la discussion sur les valeurs propres et les vecteurs propres à la p. 204. Si ce que nous voulons dire n'est pas clair, essayez d'ajuster les valeurs  $xmin$ ,  $xmax$ ,  $ymin$ ,  $ymax$  pour voir une partie plus petite du graphique.

## Moyenne des moindres carrés

Un système de deux équations linéaires à deux variables trouvera une solution exacte pour des données exactes. Malheureusement, le monde réel manque souvent de données exactes. Les scientifiques et les ingénieurs compensent donc en collectant des données supplémentaires et en faisant la moyenne des résultats. Par exemple, dans le cas de deux inconnues  $x$  et  $y$ , nous pourrions prendre  $m$  mesures correspondant au

$$\begin{array}{l} \text{système } x + \\ a_1 y = b_1 x + a_2 y = b_2 \\ \vdots \\ x + a_m y = b_m . \end{array}$$

La meilleure solution  $(x, y)$  minimise l'erreur ; c'est-à-dire qu'elle minimise

$$\begin{array}{ll} b_1 & x + a_1 y \\ \vdots & - \quad \vdots \\ b_m & x + a_m y \end{array}^2 = [b_1 - (x + a_1 y)]^2 + \dots + [b_m - (x + a_m y)]^2 .$$

Nous pouvons minimiser cela de la manière suivante : soit

$$A = \begin{array}{ccc|c} 1 & a_1 & & b_1 \\ \vdots & \vdots & & \vdots \\ & & & b_m \end{array}, \quad X = \begin{array}{c} x \\ \vdots \\ y \end{array}, \quad B = \begin{array}{c} b_1 \\ \vdots \\ b_m \end{array}.$$

et  
1 heure du matin

Le système d'équations ci-dessus se traduit par  $AX$

$$= B.$$

Si nous multiplions les deux côtés de gauche par  $A^T$ , nous avons  
 $A^T A X = A B$ .

abscisses       $T$  et  $A$  sont une matrice carrée. Si au moins deux points  $(a_i, b_i)$  ont des distinctes, le produit de  $A$  est inversible et peut être résolu en multipliant les deux côtés :

$$X = A^{-1} A^T B.$$

On peut montrer que ce  $X$  minimise l'erreur (bien que nous ne le montrerons pas).

1. Écrivez un pseudo-code pour une procédure qui accepte deux listes de mesures correspondantes  $a_1, \dots, a_m$  et  $b_1, \dots, b_m$  et résout pour  $x_1, \dots, x_m$ .
2. Implémentez le pseudo-code en tant que procédure Sage nommée `least_squares_2d()`.
3. Écrivez une procédure Sage qui accepte deux listes de mesures correspondantes  $a_1, \dots, a_m$  et  $b_1, \dots, b_m$ , trace chaque point  $(a_i, b_i)$  et trace la ligne qui se trouve le long du vecteur solution  $(x_0, y_0)$ . La procédure peut éventuellement accepter deux arguments supplémentaires : `color_points` et `color_line`, qui indiquent respectivement la couleur des points et celle de la ligne. Veillez à utiliser les commandes `min()` et `max()` de Sage pour définir correctement  $y_{\min}$ ,  $y_{\max}$ ,  $x_{\min}$  et  $x_{\max}$ .

## Méthode Bareiss

La méthode Bareiss pour calculer un déterminant peut être décrite en pseudo-code comme suit :

```

algorithme Bareiss_déterminant
entrées
    • une matrice n × n M
sorties •
    detM

soit D une copie de M
soit a =
    1 pour k    {1,..., n - 1}
        pour i    {k + 1,..., n}
            pour j    k + 1,..., n
                dk,j soit  $\frac{di,j dk,k-di,k}{un}$ 
                di,j = soit a
            = dk,k return Dn,n

```

1. Implémentez ce pseudo-code comme un programme Sage et testez-le sur les matrices suivantes. Lorsqu'il produit un résultat, celui-ci est-il correct ? S'il ne produit pas de résultat, que fait-il exactement ? Dans ce dernier cas, étudiez les entrées de la matrice pour identifier l'erreur.

$(a) M = \begin{matrix} 3 & 2 & 1 & 5 \\ 4 & 3 & 6 & 3 \\ 4 & \end{matrix}$	$(b) M = \begin{matrix} 1 & -4 & 1 & 2 & -1 & 4 \\ 4 & 1 & 3 & 3 & 3 & 4 & 2 & 5 & 2 \\ -1 & \end{matrix}$	$(c) M = \begin{matrix} 1 & 2 & 3 & 4 & 0 \\ 1 & 1 & 1 & 1 & 2 \\ 1 & 2 & 1 & 1 & 3 & 3 \end{matrix}$
---	--	---

2. Comme avec la méthode gaussienne, nous pouvons récupérer un calcul de l'algorithme de Bareiss en cas d'échec, et plus ou moins de la même manière. Modifiez votre implémentation de Bareiss pour en tenir compte, puis testez-la à nouveau sur les trois matrices pour vous assurer qu'elle fonctionne correctement.

### La méthode de

Dodgson est une méthode « intuitivement simple » pour calculer un déterminant.<sup>3</sup> Le pseudo-code suivant la décrit précisément. (Ici, la numérotation des lignes et des colonnes commence à 1, plutôt qu'à 0 comme dans Sage.)

```

algorithme Dodgsons_method
entrées
    • M, une matrice n × n
sorties •
    detM

soit L une matrice (n - 1) × (n - 1) de 1 pour i
{1,..., n - 1} soit m
le nombre de lignes de M soit N une
matrice (m - 1) × (m - 1) pour j {1,...,
m - 1}
pour k {1,..., m - 1}
    Lj ,k soit = (Mj,k×Mj+1,k+1-Mj+1,k×Mj ,k+1)/
Nj ,k soit L la sous-matrice (m - 2)×(m - 2) de M commençant à la ligne 2, colonne 2
remplacez M par N
renvoyez M

```

1. Implémentez ce pseudo-code comme un programme Sage et testez-le sur les matrices suivantes. (Un problème risque de se produire avec l'une d'elles.)

$$\begin{array}{lll}
 \text{(a) } M = \begin{matrix} 3 & 2 & 1 & 5 \\ 4 & 3 \\ 6 & 3 & 4 \end{matrix} & \text{(b) } M = \begin{matrix} 1 & -4 & 1 & 2 & -1 & 4 \\ 4 & 1 & 3 & 3 & 3 & 4 & 2 & 5 & 2 \\ -1 \end{matrix} & \text{(c) } M = \begin{matrix} 1 & 2 & 3 & 4 & 0 \\ 1 & 1 & 1 & 1 & 2 \\ 1 & 2 & 1 & 1 & 3 & 3 \end{matrix}
 \end{array}$$

2. Lorsqu'un résultat est produit, ce résultat est-il réellement correct ?
3. Si cela ne produit pas de résultat, que fait-il exactement ?
4. Dans ce dernier cas, essayez d'effectuer le calcul manuellement. Expliquez le problème.

---

3. Nommé d'après son inventeur, Charles Lutwidge Dodgson, diacre de l'Église d'Angleterre et titulaire d'une chaire de mathématiques. Il est surtout connu pour deux livres pour enfants qu'il a écrits sous le pseudonyme de « Lewis Carroll ».

Alice au pays des merveilles et Alice de l'autre côté du miroir.

Ne cédez pas à la tentation de le réparer, à moins que vous ne souhaitiez un projet de recherche stimulant.

## Mathématiques discrètes

## Fonctions un à un

Dans ce laboratoire, nous considérons un dictionnaire comme une application ou une fonction reliant un ensemble A, le domaine (l'ensemble des clés du dictionnaire), à un autre ensemble B, l'intervalle (ou codomaine). Cet intervalle peut contenir plus d'éléments que ceux qui apparaissent dans les valeurs du dictionnaire.

1. Écrivez une procédure Sage qui accepte un dictionnaire D en entrée et renvoie True si D est un dictionnaire à un seul élément fonction to-one et False sinon.
2. Écrivez une procédure Sage qui accepte un dictionnaire D en entrée et renvoie True si D est une fonction bijective ; sinon, elle renvoie False et un contre-exemple.
3. Écrivez une procédure Sage acceptant un dictionnaire D et un ensemble B en entrée et renvoyant Vrai si D est sur B, et Faux sinon. Dans ce dernier cas, elle doit également renvoyer tous les éléments de B qui ne sont pas des valeurs de D.
4. Écrivez une procédure Sage acceptant un dictionnaire D et un ensemble B en entrée et renvoyant Vrai si D est bijectif et sur B, et Faux dans le cas contraire. Dans ce dernier cas, elle affiche (et non renvoie !) la propriété qui échoue et le contre-exemple correspondant.

## Le jeu des ensembles

Le jeu se compose de 81 cartes, chacune ayant quatre propriétés : • Le nombre d'objets sur la carte est 1, 2 ou 3. • La couleur du ou des objets est identique : rouge, vert ou violet. • L'ombrage du ou des objets est identique : plein, rayé ou vide. • La forme du ou des objets est identique : losanges, ovales ou gribouillis. 4 cartes.

Notez que chaque propriété a 3 possibilités, ce qui explique pourquoi il y a  $3^4 = 81$

Le but du jeu est d'identifier autant de « sets » que possible dans un groupe de cartes. Un « set » est toute collection de cartes dans laquelle chaque propriété est identique ou différente.

1. Écrivez une procédure Sage make\_cards() qui crée et renvoie une liste de toutes les cartes. Chaque carte doit être un tuple de quatre entiers, chaque entier indiquant sa propriété. Ainsi, (1, 1, 3, 2) représenterait une carte avec un ovale rouge vide.
2. Écrivez une procédure Sage is\_set() qui accepte un argument, L, qui est une liste de 3 cartes. Elle renvoie True si L forme un ensemble, et False sinon. Vous pouvez utiliser cette procédure pour appeler d'autres procédures testant si une propriété est identique ou différente pour toutes les cartes.
3. Dans une nouvelle cellule, saisissez « import itertools ». Écrivez une procédure Sage has\_set() qui accepte un argument, B, une liste d'au moins trois cartes, et renvoie « Vrai » si B possède un ensemble, et « Faux » sinon. Vous pouvez utiliser la commande BI = itertools.combinations(B,3) pour créer un itérateur qui choisira toutes les combinaisons possibles de 3 cartes de B. Vous pouvez ensuite parcourir BI en utilisant une boucle for .
4. Écrivez une procédure Sage prob\_set\_in(N) qui prend un entier d'au moins 3 et renvoie la probabilité qu'un plateau de taille N sélectionné aléatoirement contienne un ensemble. Il est déconseillé d'essayer cette procédure sur une valeur N élevée, car son exécution serait très longue. Testez-la uniquement avec N = 3 (quelques secondes, maximum) et N = 4 (quelques minutes, maximum). Vous pouvez utiliser la commande CN = itertools.combinations(CL,N) pour créer un itérateur qui sélectionnera toutes les combinaisons possibles de N cartes dans une liste CL (dans ce cas, une liste de cartes).

## Le nombre de façons de sélectionner m éléments dans un ensemble de n

Supposons que vous ayez un sac contenant n objets distincts et que vous souhaitez en sélectionner m. Un tel choix de boules est appelé une combinaison, par opposition à une permutation où l'ordre est important. Il existe de nombreuses applications où l'on souhaite compter le nombre de façons de sélectionner m boules dans un sac de n objets distincts. On écrit cela sous la forme  $\binom{n}{m}$  ou la formule pour cette valeur est

$$\frac{n!}{m!(n-m)!}.$$

1. Écrivez une procédure Sage qui accepte n et m en entrée et calcule  $\binom{n}{m}$ . Utilisez votre procédure pour calculer  $\binom{n}{m}$  pour n = 2, 3, ..., 8 et m = 0, ..., n.
3. Remarquez-vous une tendance dans les nombres ? Indice : Revenez au triangle de Pascal.
4. Décrivez comment on pourrait utiliser des combinaisons pour implémenter une procédure qui calcule la fonction de Pascal. Triangle sans utiliser la récursivité.

---

<sup>5</sup>Par exemple, la loterie PowerBall attribue des prix en fonction du nombre de boules sélectionnées, parmi 69, que dans une émission de télévision, et pas nécessairement dans le même ordre. (Il existe également un autre critère, mais il faut au moins connaître le nombre de façons de sélectionner 5 boules sur 69.)

## Algèbre et théorie des nombres

## Propriétés des anneaux finis

1. Créez une nouvelle feuille de travail. Intitulez-la : « Laboratoire : Propriétés des anneaux finis ». Ajoutez d'autres informations pour vous identifier, si nécessaire.

Relisez la section sur les « Structures mathématiques dans Sage » dans le chapitre intitulé « Calculs de base ».

2. Créez une section intitulée « Arithmétique modulaire : démonstration », puis :

- (a) Définissez un anneau  $R_{10}$ , l'anneau fini de 10 éléments.

comme étant (Indice : les notes révisées montrent une manière plus simple de procéder que la démonstration en classe.)

- (b) Définissez  $m$  comme étant la valeur de 2 dans  $R$ , et calculez  $1 \times m, 2 \times m, 3 \times m, \dots, 10 \times m$ .

(Indice : si votre réponse au dernier produit est 20, vous vous y prenez mal. Vous devez convertir 2 en une valeur de l'anneau  $R$ . Les notes montrent comment procéder.)

- (c) Définissez  $n$  comme étant la valeur de 3 dans  $R$  et calculez  $1 \times n, 2 \times n, 3 \times n, \dots, 10 \times n$ .

- (d) Définissez  $r$  comme étant la valeur de 5 dans  $R$ , et calculez  $1 \times r, 2 \times r, 3 \times r, \dots, 10 \times r$ .

- (e) Définissez  $s$  comme étant la valeur de 7 dans  $R$  et calculez  $1 \times s, 2 \times s, 3 \times s, \dots, 10 \times s$ .

- (f) Définissez  $t$  comme étant la valeur de 9 dans  $R$  et calculez  $1 \times t, 2 \times t, 3 \times t, \dots, 10 \times t$ .

3. Pour ce qui suit, écrivez votre réponse dans un champ de texte à la fin de la feuille de travail. Le haut de

la zone de texte devrait avoir pour titre « Arithmétique modulaire : analyse ».

- (a) Remarquez que  $10 \times 2 = 10 \times 3 = \dots = 10 \times 9$  dans cet anneau. Pourquoi cela est-il logique, étant donné le module ?

- (b) Lequel des nombres  $m, n, r, s, t$  répertorie tous les nombres de 0 à 9 ?

- (c) Quelle propriété partagent les nombres que vous avez énumérés en (b) que les autres nombres n'ont pas ?

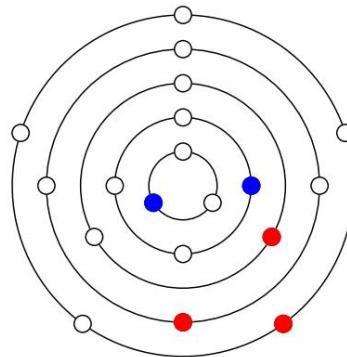
## L'horloge chinoise du reste

Dans le College Mathematics Journal de mars 2017, Antonella Perucca a décrit une « horloge à reste chinois » qui schématise le temps en utilisant les restes de la division.

- (1) Pour l'heure  $h$ , soit  $a, b$  les restes de la division de  $h$  par 3 et 4, respectivement.
- (2) Pour la minute  $m$ , soit  $c, d, e$  les restes de la division de  $m$  par 3, 4 et 5, respectivement.
- (3) Dessinez cinq anneaux concentriques noirs, en marquant : • le premier et le troisième (à partir du centre) avec trois cercles vides également espacés, • le deuxième et le quatrième (à partir du centre) avec quatre cercles vides également espacés, et • le plus à l'extérieur avec cinq cercles vides également espacés.
- (4) Enfin, placez • un cercle bleu sur le quatrième cercle dans l'anneau le plus intérieur (en comptant dans le sens des aiguilles d'une montre à partir du haut),  
 • un cercle bleu sur le  $b$ -ième cercle du deuxième anneau le plus intérieur,  
 un cercle rouge sur le  $c$ -ième cercle du troisième anneau le plus intérieur,  
 • un cercle rouge sur le  $d$ -ième cercle du quatrième anneau le plus intérieur, et • un cercle rouge sur le  $e$ -ième cercle de l'anneau le plus extérieur.

Un théorème appelé théorème du reste chinois nous dit que ce diagramme donne une représentation unique de l'heure actuelle.

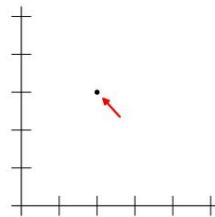
Par exemple, supposons que l'heure actuelle soit 5h22. Alors  $a = 2$ ,  $b = 1$ ,  $c = 1$ ,  $d = 2$  et  $e = 2$ , résultant dans le diagramme



Écrivez un programme Sage qui, étant donné une heure  $h$  et une minute  $m$ , produit l'horloge restante chinoise correspondante.

## La géométrie des racines radicales

REMARQUE. Dans ce devoir, nous considérons toutes les solutions comme des nombres complexes  $a + bi$ , où  $i = \sqrt{-1}$ . Lorsqu'on vous demande de représenter  $a + bi$  sur le plan complexe, représentez-le au point  $(a, b)$ . Ainsi, par exemple, représenter le nombre complexe  $2 + 3i$  vous donnerait le point  $(2, 3)$ :



- Créez une nouvelle feuille de travail. Intitulez-la : « Laboratoire : Géométrie des racines radicales ». Ajoutez d'autres informations permettant de vous identifier, si nécessaire.
- Utiliser Sage pour résoudre l'équation  $x^2 - 1 = 0$ . Tracez toutes les solutions sur un plan complexe. Ce n'est pas très intéressant, n'est-ce pas ?
- Utilisez Sage pour résoudre l'équation  $x^3 - 1 = 0$ . Tracez toutes les solutions sur un plan complexe, et non sur le comme précédemment. C'est un peu plus intéressant.
- Utiliser Sage pour résoudre l'équation  $x^4 - 1 = 0$ . Tracez toutes les solutions sur un troisième plan complexe. Ceci être un peu ennuyeux, encore une fois.
- Utilisez Sage pour résoudre l'équation  $x^5 - 1 = 0$ . Tracez toutes les solutions sur un quatrième plan complexe. (Vous aurez probablement besoin d'utiliser `real_part()` et `imag_part()` ici.) Cette figure devrait être saisissante, surtout si vous reliez les points, mais ne le faites pas ; tracez simplement les points, notez le résultat et passez à autre chose.
- Pouvez-vous deviner ? Oui, utilisez Sage pour résoudre l'équation  $x^6 - 1 = 0$ . Représentez toutes les solutions sur un autre plan complexe. Cela devient probablement ennuyeux, ne serait-ce que parce que vous pouvez probablement deviner le résultat non seulement de celui-ci, mais aussi de...
- Utilisez Sage pour résoudre l'équation  $x^7 - 1 = 0$ . Tracez toutes les solutions sur un autre plan complexe. Si vous comprenez le modèle géométrique, passez à autre chose ; sinon, protestez. Ou, pour gagner du temps, vous devinerez probablement que je vous demanderai de tracer les solutions pour quelques exemples supplémentaires de  $x^n - 1 = 0$ , mais avec des valeurs de  $n$  plus grandes. Tôt ou tard, vous devriez percevoir un modèle géométrique.
- Dans une cellule HTML, expliquez pourquoi le motif géométrique des images justifie l'affirmation selon laquelle  $-1$  ont tous les solutions à  $x^n$  la forme  $2\pi k + i \sin$

parce que  $\frac{2\pi}{n} + i \sin\left(\frac{2\pi}{n}\right)$ .

Assurez-vous d'expliquer le rapport entre  $k$  et  $n$ . Si cela peut vous aider, examinez d'abord le tracé paramétrique de  $\cos(2\pi t) + i \sin(2\pi t)$  pour  $t \in [0, 1]$ , en vous rappelant, comme précédemment, que vous tracez  $a + bi$  comme  $(a, b)$ .

Avant de lever la main, paniqué, prenez un instant, puis respirez profondément et repensez à la signification du sinus et du cosinus. Consultez un manuel si nécessaire. Vous pouvez y arriver, c'est sûr !

De même, ne me demandez pas comment créer un graphique paramétrique. Nous l'avons déjà vu, donc si vous ne vous en souvenez pas, consultez les notes.

- Définir  $\omega = \cos \frac{2p}{6} + i \sin \frac{2p}{6}$ . En utilisant une boucle for dans Sage, calculez  $\omega, \omega^2, \dots, \omega^6$ . Tracez chaque nombre sur le plan complexe, et décrivez comment le résultat est cohérent avec vos réponses aux questions n° 6 et n° 8.

10. Supposons maintenant que  $\omega = \cos \frac{2p}{n} + i \sin \frac{2p}{n}$ . À la main, élaborez une explication pour laquelle

$$(2) \quad k\omega = \cos \frac{2\pi k}{n} + i \sin \frac{2\pi k}{n}.$$

Saisissez cette explication dans votre feuille de calcul Sage, en utilisant LATEX pour la rendre plus claire. Si vous connaissez la preuve par induction, cela fonctionnera. Sinon, adoptez l'approche suivante :

- Expliquez pourquoi l'équation (2) est vraie pour  $k = 1$ .

Développez  $\omega^2$  et utilisez des identités trigonométriques que vous êtes censé vous rappeler de simplifier à la valeur de l'équation (2) pour  $k = 2$ . • Faites de même pour  $\omega^3$ .

Enfin, indiquez un modèle

dans les deux étapes précédentes que vous pouvez répéter à l'infini, de sorte que si l'équation (2) est vraie pour une certaine valeur de  $k$ , elle est également vraie pour la valeur suivante de  $k$ . – a pour quelques belles valeurs de

11. Il est temps d'améliorer notre jeu. Expérimitez avec  $x^n = a$  et une valeur suffisamment grande de  $n$ . Que voyez-vous ? Formulez une conjecture quant à la forme géométrique de ces solutions.

(Le terme « belles valeurs de  $a$  » peut signifier deux choses. Premièrement, il peut signifier des nombres de la forme  $a = b^n$ ; par exemple, si  $n = 6$ , alors vous pourriez laisser  $b = 3$ , et vous auriez  $-729$ , ce qui n'est pas aussi effrayant qu'il y a  $= 3^6 = 729$ . Vous travaillerez alors avec l'équation  $x^6 = 729$  paraît.)

Franchement ! Deuxièmement, tu pourrais prendre le Bob Ross (Approche, dans laquelle tout nombre est une valeur intéressante, car, en réalité, tous les nombres sont des nombres intéressants et heureux. C'est plus difficile, cependant.)

12. Comme auparavant, faites en sorte que tout soit joli, avec des sections, des commentaires dans les zones de texte, au moins un peu LATEX, etc.

## Séquences de Lucas

La suite de Fibonacci est définie comme =

$$f_1, f_2 = 1, f_{n+2} = f_n + f_{n+1}.$$

Les nombres de Fibonacci sont un exemple de ce que les mathématiciens appellent désormais une [suite de Lucas](#).

(Plus d'informations sur le lien.) Nous définissons généralement les séquences de Lucas de manière récursive, mais vous pouvez trouver une « formule fermée » d'une manière similaire à ce que nous avons fait en classe pour la séquence de Fibonacci.

Soient a, b, c et d les deux premiers chiffres de votre numéro d'étudiant. Si deux chiffres sont identiques, modifiez-les pour qu'ils soient tous les quatre différents. La séquence = cn +

$$= a, 1 \quad 2 = b, \quad n+2 \text{ dn + 1 est une}$$

séquence de Lucas que nous appellerons « séquence [insérer votre nom de famille ici] ».

1. Dans une cellule HTML Sage, indiquez la définition de la séquence [insérer votre nom de famille ici]. Utilisez LATEX!
2. Dans la même cellule, indiquez les cinq premiers nombres de la séquence [insérez votre nom ici].
3. Définissez une matrice L et un vecteur v qui génèrent la suite. Par exemple, si les quatre premiers chiffres de votre pièce d'identité sont 1, 2, 8 et 9, alors

$$L = \begin{pmatrix} 9 & 8 \\ 1 & 0 \end{pmatrix} \quad \text{et } v = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

Calculez Lv, L 2v, L 3v, L 4v et L 5v dans Sage, puis comparez les résultats à ceux de l'étape 2. S'ils diffèrent, l'une des étapes 2 ou 3 est erronée. Ou il y a une faute de frappe. Posez la question et/ou corrigez avant de continuer.

4. Calculez les « données propres » de L. Extrayez les vecteurs et valeurs propres et demandez à Sage de les convertir sous forme radicale. (Les nombres ne doivent plus se terminer par un point d'interrogation.)
5. Construisez les matrices Q et Λ telles que  $L = Q\Lambda Q^{-1}$ . Utilisez Sage pour vérifier que  $L = Q\Lambda Q^{-1}$ .
6. Construire la matrice M =  $Q\Lambda Q^{-1}$ .

Astuce : les notes de cours en parlent ; cela nécessite quelques connaissances en algèbre linéaire.

7. Utilisez le produit de M et v pour trouver la forme fermée de la séquence [insérez votre nom ici].

Astuce : encore une fois, les notes devraient vous être utiles ici si vous avez besoin d'aide.

8. Utilisez la forme fermée pour calculer les cinq premiers nombres de la suite [insérer votre nom ici] et comparez vos résultats à ceux des questions 2 et 3. S'ils diffèrent, vous avez un problème ou une faute de frappe ; contactez-moi et/ou corrigez-le !

## Introduction à la théorie des groupes

Arrière-plan.

DÉFINITION. Si un ensemble  $S$  et une opération  $\circ$  satisfont aux propriétés de clôture, d'associativité, d'identité et de réciproque, alors  $S$  est un groupe sous  $\circ$ . Ces propriétés sont définies de la manière suivante :

- clôture :  $x \circ y \in S$  pour tout  $x, y \in S$  ;
- associative :  $(x \circ y) \circ z = x \circ (y \circ z)$  pour tout  $x, y, z \in S$  ;
- identité : on peut trouver  $i \in S$  tel que  $x \circ i = x$  et  $i \circ x = x$  pour tout  $x \in S$  ;
- inverse : pour tout  $x \in S$ , on peut trouver  $y \in S$  tel que  $x \circ y = y \circ x = i$ .

EXEMPLE. Les entiers forment un groupe lors de l'addition, car

- l'addition de deux entiers quelconques donne un entier ( $x + y \in \mathbb{Z}$  pour tout  $x, y \in \mathbb{Z}$ ) ;
- l'addition d'entiers est associative ; il existe une identité additive ( $x + 0 = x$  et  $0 + x = x$  pour tout  $x \in \mathbb{Z}$ ) ; et
- tout entier  $x$  a un inverse additif qui est également un entier ( $x + (-x) = (-x) + x = 0$ ).

EXEMPLE. Les entiers ne forment pas un groupe lors de la multiplication, pour deux raisons : • 0

n'a pas d'inverse multiplicatif  $0^{-1}$  ; et • les autres entiers  $a$  ont des inverses multiplicatifs  $1/a$ , mais la plupart ne sont pas des entiers. Un groupe ne satisfait la propriété d'inverse que s'il contient les inverses de chaque élément.

Dans ce laboratoire, vous utiliserez du pseudo-code pour écrire du code permettant de tester si un ensemble fini est un groupe sous multiplication. Vous le testerez ensuite sur trois ensembles : deux réussissent et un échouent. Une complication de ce projet est que la procédure doit dépendre de l'opération ; il est donc impossible d'écrire une procédure pour une seule opération.

Pseudo-code.

Clôture. Nous devons vérifier chaque paire  $x, y \in S$ . Nous pouvons vérifier si cela est vrai pour « chaque » élément d'un ensemble fini utilisant des boucles définies.

```

algorithme is_closed
    entrées
        S, un ensemble
    fini,
    produit vrai si S est fermé par multiplication ; faux sinon faire pour
    s
        S pour
            t ∈ S si
                st ∈ S
                    print("échec de fermeture pour", s, t)
                    renvoie faux
    renvoie vrai

```

Associatif. Il faut vérifier chaque triplet  $x, y, z \in S$ , ce qui nécessite des boucles définies. Le pseudo-code est un exercice.

Identité. Nous pouvons tester si nous pouvons trouver une identité à l'aide d'une variable spéciale appelée indicateur à valeur booléenne (parfois appelée signal). Nous ajustons la valeur de l'indicateur selon qu'un candidat continue de satisfaire une propriété connue. À la fin de la boucle, l'indicateur indique si nous avons terminé (c'est-à-dire si nous avons trouvé une identité). La structure des quantificateurs (« nous pouvons trouver... pour tout... ») impose au pseudo-code de présumer l'existence d'une identité jusqu'à preuve du contraire.

```

algorithme find_identity
entrées
    S, un ensemble
fini produit
    une identité, s'il peut la trouver ; sinon, faire

pour s    S
    soit maybe_identity = true
    pour t
        S si st = t ou ts = t
            soit maybe_identity = false si
            maybe_identity = true
            retour s
    print("pas d'identité")
    retour

```

Inverse. Nous recherchons un inverse pour chaque élément. Ici encore, nous utilisons un indicateur, car la logique exige de trouver un inverse. Contrairement au pseudo-code précédent, nous supposons qu'un inverse n'existe pas jusqu'à preuve du contraire ; en effet, l'ordre des quantificateurs est inversé (« pour tout... on peut trouver... » au lieu de « on peut trouver... pour tout... »). Ce pseudo-code nécessite également d'identifier l'identité de l'ensemble dans l'entrée.

```

l'algorithme a des entrées
inverses
    S, un ensemble
fini i, une identité de S sous multiplication
produit
    vrai si chaque élément de S a un inverse multiplicatif ; faux sinon faire pour s    S
    soit
        found_inverse = faux pour t
        S si st = i
            et ts = i soit
                found_inverse = vrai si
                found_inverse = faux
                print("pas d'inverse pour", s)
                renvoie faux
        renvoie vrai

```

En les assemblant. Ce pseudo-code teste si un ensemble est un groupe sous une opération en en invoquant les quatre algorithmes définis ci-dessus.

```

algorithme is_a_group
entrées
    S, un ensemble
fini produit
    vrai si S est un groupe sous multiplication ; faux sinon faire si

    is_closed(S) et is_associative(S) let i =
        find_identity(S) si i = et
        has_inverses(S, i)
            renvoie vrai
        renvoie faux

```

Vos tâches. Utilisez LATEX dans vos feuilles de travail Sage lorsque cela est approprié. Deux des ensembles des exercices 3 à 5 sont des groupes ; un autre n'en est pas un.

1. Étudiez le pseudo-code de la fermeture et écrivez le pseudo-code d'un algorithme nommé `is_associative` qui teste si un ensemble  $S$  est associatif par multiplication. Modifiez le pseudo-code de `is_closed` avec une troisième boucle et modifiez la condition du if en conséquence.
2. Écrivez du code Sage pour chacun des cinq algorithmes définis ci-dessus en pseudo-code. Vous les testerez sur les ensembles suivants.
3. Définissez un anneau  $R$  comme étant l'anneau fini de 101 éléments. (Reprenez le TP n° 2 si vous avez oublié comment faire.) Soit  $S = \{1, 2, \dots, 100\} R$  ; autrement dit,  $S$  doit inclure tous les éléments de  $R$  sauf 0. Assurez-vous de définir  $S$  en utilisant des éléments de  $R$ , et non des entiers simples. (Reprenez le TP n° 2 si vous avez oublié comment faire.) Testez votre code Sage sur  $S$  :  $S$  est-il un groupe sous multiplication ? Si non, quelle propriété échoue ?
4. Redéfinir l'anneau  $R$  comme un anneau fini de 102 éléments. Soit  $S = \{1, 2, \dots, 101\} R$  ; autrement dit,  $S$  doit inclure tous les éléments de  $R$  sauf 0. Assurez-vous de définir  $S$  en utilisant des éléments de  $R$ , et non des entiers simples. Testez votre code Sage sur  $S$  :  $S$  est-il un groupe sous multiplication ? Si ce n'est pas le cas, quelle propriété échoue ?
5. Définir les matrices

$$\begin{array}{llll} \text{je}_2 = & \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix} & \text{je} = & \begin{matrix} \text{je} & 0 & 0 \\ -\text{je} & & \end{matrix} \\ & & j = & \begin{matrix} 0 & 1 \\ -1 & 0 \end{matrix} & k = & \begin{matrix} 0 & ii & 0 \end{matrix} \end{array}$$

et l'ensemble

$$Q = \{\text{je}_2, -\text{je}_2, i, -i, j, -j, k, -k\}.$$

Testez votre code Sage sur  $Q$  ;  $Q$  est-il un groupe sous multiplication ? Si ce n'est pas le cas, quelle propriété échoue ?

REMARQUE. Cet ensemble est parfois appelé l'ensemble des quaternions.

6. En utilisant les matrices du problème n°4, définissez

$$S = \{I, \text{je}_2, i, j, -i, -j, k, -k\}.$$

- (a) Vous avez probablement remarqué que  $S \subseteq Q$ .  $S$  est-il aussi un groupe ? Si oui, on appelle  $S$  un sous-groupe de  $Q$ .  
Si ce n'est pas le cas, quelle propriété échoue ? (b) L'ensemble  $S$  est en fait constitué de matrices de la forme A du laboratoire n° 6, problème n° 1.  
Indiquez dans une cellule HTML la valeur correcte de  $a$  pour chaque matrice.

## Théorie du codage et cryptographie

Deux applications de l'algèbre et de la théorie des nombres sont la théorie du codage et la cryptographie :

- L'objectif de la théorie du codage est de transmettre des informations de manière fiable, en détectant les erreurs et, lorsque cela est possible, en les corrigeant.
- L'objectif de la cryptographie est de transmettre des informations de manière sécurisée, de sorte qu'un espion ne puisse ni comprendre ni comprendre le sens de la transmission.

Pour appliquer ces concepts mathématiques, une étape préliminaire de transformation du texte en nombres, et inversement, est nécessaire. Vous écrirez deux procédures ou plus pour aider quelqu'un ; l'une d'elles sera interactive.

Pour transformer du texte en nombres, il est nécessaire de le regrouper en blocs de taille adaptée. Par exemple, la phrase

SORTEZ DE L'ESQUIVETTE

On peut les regrouper de plusieurs manières. L'une d'elles consiste à les regrouper en blocs de deux :

GE À UT DE DO DG EX

et une autre consiste à le regrouper en blocs de quatre :

OBTENEZ-LE DE DODG XXXX

Dans les deux cas, la longueur du message ne divise pas la longueur du groupe de manière égale, nous remplissons donc le message avec des X.

Comment faire ? Python possède une commande appelée `ord()` qui permet de convertir chaque caractère. acteur en nombre. Cette commande convertirait notre phrase en nombres.

71, 69, 84, 79, 85, 84, 79, 70, 68, 79, 68, 71, 69 .

Vous remarquerez que j'ai supprimé les espaces et que je n'ai pas encore créé les groupes. Comment les regrouper ? D'abord, transformez-les en nombres de 0 à 25 en soustrayant le nombre correspondant à A. Cela nous donne

6, 4, 19, 14, 20, 19, 14, 5, 3, 14, 3, 6, 4 .

Pour les regrouper, nous raisonnons de la manière suivante :

- Nous n'avons pas besoin de lettres minuscules ni de ponctuation pour faire passer notre message.<sup>6</sup>
- Ainsi, chaque valeur à encoder se situe entre 1 et 26. • Nous pouvons utiliser un système numérique en base 26 pour encoder n'importe quel groupe de lettres.

Ainsi, pour encoder un groupe de quatre lettres avec des valeurs numériques a, b, c, d, nous pouvons calculer

$$m = a + 26b + 262c + 263d .$$

En général, pour encoder un groupe de n lettres avec des valeurs numériques a<sub>1</sub>, a<sub>2</sub>, ..., a<sub>n</sub>, il faut calculer

$$m = a_1 + 26a_2 + 26^2a_3 + \dots + 26^{n-1}a_n .$$

Pour décoder un groupe codé m de n caractères, effectuez les opérations suivantes n fois : •

- déterminez le reste de la division de m par 26 et appelez-le a ; • reconvertissez a en caractère ; et enfin, • remplacez m par m-a/26.

Vous devez écrire au moins deux procédures.

---

<sup>6</sup> Anecdotes connexes : Les langues écrites anciennes utilisaient généralement toutes les lettres majuscules sans ponctuation ni même espace.

1. La première procédure, `encode()`, prend deux entrées : un entier  $n$  et une liste  $L$  de caractères. Elle convertit chaque caractère en un nombre, organise les nombres en groupes de  $n$  caractères, puis convertit chaque groupe en un seul nombre à l'aide de la formule ci-dessus. Elle renvoie une liste  $M$  de nombres, un pour chaque groupe. On peut supposer que  $n$  divise la longueur de  $L$ . (Cela devient un peu plus difficile s'il faut déterminer le remplissage.)
2. La deuxième procédure est interactive. Elle propose à l'utilisateur une zone de texte et un curseur. Dans la zone de texte, l'utilisateur saisit un message. Dans le curseur, il sélectionne entre 2 et 6 caractères. La procédure prend ces valeurs et les envoie à `encode()`, puis affiche la liste de numéros renvoyée par `encode()`.

Pour obtenir des crédits supplémentaires,

vous pouvez également : 3. Implémenter une troisième procédure, `decode()`, qui prend deux entrées : un entier  $n$  et une liste  $M$  d'entiers.

Il convertit chaque entier en un groupe de  $n$  entiers, puis convertit chaque entier en un caractère.

Pour cela, vous aurez peut-être besoin de la commande Python `chr()`.

4. Cythonize soit `encode()` , soit `decode()`. Au moins un type de variable doit être déclaré.

## Fractions continues

Une représentation fractionnaire continue d'un nombre a la forme suivante :

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \dots}}}$$

Les fractions continues possèdent un certain nombre de propriétés fascinantes, dont certaines seront explorées dans ce laboratoire. Le pseudo-code suivant calcule une fraction continue jusqu'à m chiffres :

```

algorithme fraction_continue entrées
• b, un
    nombre réel • m, le
    nombre d'emplacements pour développer la fraction continue sorties • une
    liste (a0 ,
        a1 ,..., am ) listant les entrées de la fraction continue

    soit L une liste de m + 1 zéros soit i
    = 0 tant
    que i ≤ m et b = 0 pose Li au
        plancher de b soit d = b -
        Li si d = 0
        remplace
            d par 1/d soit b = d
            ajoute 1 à
            i renvoie L

```

1. Implémentez ce pseudo-code comme une procédure Sage. Indice : la procédure `floor()` de Sage pourrait s'avérer utile ici.
2. Utilisez votre code pour calculer les approximations de fractions continues à 20 positions maximum pour 12/17, 17/12, 4/15, 15/4, 729/1001 et 1001/729.
3. Notez au moins deux propriétés que vous remarquez à propos des fractions continues ci-dessus. Expliquez pourquoi ces propriétés ont du sens.
4. Utilisez votre code pour calculer les approximations de fractions continues à 20 positions maximum pour  $\sqrt{41}$ ,  $\sqrt[3]{5/2}$ ,  $\sqrt[4]{1+5/2}$ ,  $\sqrt[5]{1+7/3}$  et  $\sqrt[7]{1+7}$ .
5. Notez au moins deux propriétés que vous remarquez concernant ces fractions continues. L'une d'elles peut être identique à celle des nombres rationnels, mais l'autre ne devrait certainement pas l'être.
6. Utilisez votre code pour calculer les approximations des fractions continues à 20 chiffres maximum pour  $e$ ,  $e^2$  et  $e^3$ . Notez les tendances que vous observez concernant ces fractions continues. (Il est peu probable que vous en trouviez une commune aux nombres précédents, mais il serait intéressant d'en trouver une.)

## Bibliographie

- [1] Michael H. Albert, Richard J. Nowakowski et David Wolfe. *Leçons de jeu*. AK Peters, Ltd., Wellesley, Massachusetts, 2007.
- [2] Elwyn Berlekamp, John Conway et Richard Guy. *Des méthodes gagnantes pour vos jeux mathématiques*. AK Peters / CRC Press, deuxième édition, 2001.
- [3] Anna M. Bigatti. Calcul des séries de Hilbert-Poincaré. *Journal d'algèbre pure et appliquée*, 119(3):237–253, 2013-2014. 1997.
- [4] Anna M. Bigatti, Massimo Caboara et Lorenzo Robbiano. Calcul des bases de Gröbner inhomogènes. *Journal of Symbolic Computation*, 46(5) : 498–510, 2010.
- [5] Alberto Damiano, Graziano Gentili et Daniele Struppa. Calculs dans l'anneau des polynômes quaternioniques miaux. *Journal of Symbolic Computation*, 45:38–45, janvier 2010.
- [6] Vladimir Gerdt et Yuri Blinkov. Bases involutives des idéaux polynomiaux. *Mathématiques et ordinateurs en simulation*, 45(5–6):419–541, mars 1998.
- [7] Maxima. Maxima, un système de calcul formel. Version 5.38.1. <http://maxima.sourceforge.net/>, 2016.
- [8] Marin Mersenne. Pensées physico-mathématiques. 1644.
- [9] Diophante d'Alexandrie. Arithmétique. 1670. Avec des notes marginales de Pierre de Fermat.
- [10] Euclide d'Alexandrie. Éléments. c. 300 av. J.-C.
- [11] Jamie Uys. Les Dieux doivent être fous, 1980. Film distribué par 20th Century Fox.

## Indice

AA, 203  
 valeur absolue,  
`26 .add()`,  
 115 nombres algébriques, voir aussi algorithme  
 AA , 91 alpha,  
`52, 55 animate()`,  
`68 .append()`,  
 114 arc(),  
 58 argument, 79, 82  
     valeur par défaut,  
     84 obligatoire,  
 82  
     arguments  
 facultatifs, 84  
`arrow()`, 52,  
 60 assume(), 39 forget(), 39  
`.base_ring()`, 140  
  
 mise en  
     , cache, 198 voir nombres complexes  
 CC, 42,  
  
 138 `.change_ring()`,  
 140 circle(),  
 57 classe,  
 233 attribut, 233  
     instance, 233  
     méthode, 21  
     méthodes spéciales  
     arithmétique, 236  
     comparaison,  
 238 `.__hash__()`,  
 239 `.__init__()`,  
 233 `.__repr__()`, 235 `.clear()`, 116 `coefficient()`, 268 collection, 51, 109  
     indexation, 109, 110  
     appartenance à, voir dans mutable,  
 109  
 couleur, 52 nombre complexe  
  
 parties réelles et imaginaires, voir `real_part()` et `imag_part()`  
 nombres complexes, voir aussi CC corps complexe,  
 42 plan complexe, 135  
`complex_point()`, 135  
 norme, 44  
 compréhensions, 123  
 constante, 28  
 constructeur, 233  
`copy()`, 142  
 cosinus,  
`59 .count()`, 113  
`Cylindrical()`, 219  
  
 def, 79  
 boucle définie,  
`106 .degree()`, 131,  
 133 DeprecationWarning  
     Substitution utilisant la syntaxe d'appel de fonction, 34, 83  
     Plages sans nom pour plusieurs variables, 73 dérivée, 35  
`diagonal_matrix()`,  
 141 dict(), 110 dictionnaire, 110, 114  
 commandes partagées  
     avec toutes les collections, 112 méthodes, 116  
     diff,  
  
 35 `.difference()`,  
 115 `.difference_update()`, 115  
 division,  
     ratio, quotient et reste, 26 domaine, 118  
  
 e, 27  
 valeur propre,  
 148 vecteur propre,  
 147, 203  
`ellipse()`, 58  
     équation, 30 côté gauche ou droit, voir `.rhs()`, `.lhs()`  
     nécessite deux signes égaux, 31, 137  
 système, 139

Méthode d'Euler, 105  
 sauf, 160  
 expand(), 31  
 exponentiation, 26  
 expression  
     distincte de l'équation, 30 .extend(),  
     114  
  
 factor(), 31  
 factoriel, 130  
 Faux, 26  
 Séquence de Fibonacci, 194–196  
 corps, 42  
 find\_root(eq,a,b), 137 virgule  
 flottante float, 46  
  
     RealLiteral, 46  
     Nombre réel, 46  
     RR,  
     46 floor(), 297  
     forget(), voir assume()  
     ensemble gelé, voir  
     ensemble frozenset(), 110  
  
 Entiers gaussiens, 181  
     division, 181–185  
 variable globale, 89, 198  
 nombre d'or, 204, 261  
  
 .has\_key(), 116  
 hachage, 239  
  
 Moi,  
 27 identifier,  
 29 identity\_matrix(), 141  
 imag\_part(), 44  
 implicit\_plot(), 71  
 implicit\_plot3d(), 222 in,  
 112 ind,  
 36  
 boucle infinie, 106  
 Erreur d'indentation  
     on attendait un bloc indenté, 80 non  
     indenté ne correspond à aucun bloc extérieur  
         niveau d'indentation, 95  
 indéterminé, 28  
     réinitialisation d'un nom,  
 29 .index(), 113  
 IndexError  
     index de liste hors limites, 117, 180  
 indexation, 110  
     numérotation, 111  
 résolution  
     d'inégalité,  
 136 boucle infinie, voir boucle, infinie  
  
 Infini  
     unsigned\_infinity, 37 infinity  
 +Infinity,  
     27 -Infinity, 27  
     input, 91 .insert(),  
 114  
 instance, 233  
 entiers, voir  
 aussi ZZ, voir aussi entiers gaussiens  
     int, 46  
 Entier, 45 anneau  
     d'entiers, 42, 144 modulo n,  
     45 intégrale, 35  
 intégrer, 35  
 intégration  
     approximative,  
         41 exacte, 38, 39  
         @interact, voir les  
 feuilles de travail interactives feuilles de travail  
 interactives, 96–100 objets d'interface  
     disponibles, 97 .intersection(),  
     115 .intersection\_update(), 115  
 IOError  
  
     n'a pas trouvé de fichier ... à joindre,  
 95 .is\_immutable(),  
 142 .is Mutable(),  
 142 .isdisjoint(),  
 115 .issubset(),  
 115 .issuperset(), 115  
 itération, 106  
  
 KeyError, 115, 117  
 mot-clé, 30, 79  
     break, 158  
     cimport, 244  
     def, 79  
     elif, 155  
     else, 155  
     except, 160  
     for, 107, 108, 117  
     from, 242  
     global, 89, 199 if,  
     155  
     import, 242  
     print, 80  
     raise, 162  
     return, 89  
     try, 160  
     while, 175  
  
 langage  
     interprété v. compilé v. bytecode, 12 LATEX,  
     61–63, 250

.leading\_coefficient(), 131 len(), 112 .lhs(), 133, 137 lim, 35 limite, 35 saut de ligne, 25 ligne(), 52 indépendance linéaire, 148 style de ligne, 52, 57, 60 liste, 109, 111, 112 commandes partagées avec toutes les collections, 112 commandes partagées avec le tuple, 113 méthodes, 112, 114 liste(), 109 variable locale, voir variable, globale vs. boucle locale, 106 définie, 106 indéfinie, 106 infinie, voir boucle infinie variable de boucle, 118 imbrication, 120–121 pseudocode, 106 matrices, 140–149 modification de l'anneau de base, 140 construction, 140, 141 mutabilité, 142 matrice(), 140 max(), 112 message message d'erreur, voir la méthode d'erreur particulière, voir la méthode de classe, 233 méthode de bisection, 152 min(), 112 arithmétique modulaire, voir les entiers modularité, 78, 88 multiplication, voir aussi expand(), 31 Erreur de nom nom global ... n'est pas défini, 89, 200 nom ... n'est pas défini, 31, 74 nombres naturels, 44 propriété de bon ordre, 185, 192 imbrication, 120–121, 161 méthode de Newton, 174 Nim, 231 arithmétique de Nimber, 232–233 voir les nombres naturels , norme(), 44 intégrale\_numérique(), 41 -oo, voir l'infini oo, voir l'infini sortie, 91 PANIQUE !, 20, 36, 47, 87, 111, 144, 167, 237, 248 parametric\_plot(), 66 parametric\_plot3d(), 218 triangle de Pascal, 192–194 π, 27 pi, 27 plot(), 65 plot3d(), 214 plot\_vector\_field3d(), 223 point(), 52 polar\_plot(), 67 polygon(), 55 polynôme trouver le degré, 131 trouver le coefficient dominant, 131 .pop(), 114–116 .popitem(), 116 puissance, 26 imprimer, 80 procédure, 33, 79 pseudocode, 91 autres exemples, 92 notre norme, 91 , voir les nombres rationnels QQ, 42, 144 .quo(), 45 quotient, 26 randint(), 274, 277 nombres aléatoires, 274, 277 range(), 116 nombres rationnels, voir aussi QQ corps rationnel, 42, 144 nombres réels, voir aussi RR corps réel, 42 real\_part(), 44 récursivité, 192–196, 239 alternatives, 196 alternatives, 206 reste, 26 .remove(), 114, 115 répéter, 106 mot réservé, 79 réinitialiser(), 29 retourner, 89 .reverse(), 114 revolution\_plot3d(), 221 .rhs(), 133, 136, 137 anneau, 42, 138 changer la base d'une matrice anneau, 140 racine, 132 racines(), 135, 138 round(), 31

, voir les nombres réels  
 RR, 42, 138  
 Erreur d'exécution  
 f semble ne pas avoir de zéro sur l'intervalle,  
 138  
 profondeur de récursivité maximale dépassée lors de  
 l'appel d'un objet Python, 196, 200  
 aucune racine explicite trouvée, 138

séquence  
 forme fermée, 202  
 set, 109, 114  
 commandes partagées avec toutes les collections, 112  
 éléments doivent être immuables, 110, 142 gelé,  
 110, 114 méthodes,  
 115 opérations  
 sur, 115 set(),  
 109 .set\_immutable(), 142  
 Référence éhontée à L'Homme qui valait trois milliards que les  
 étudiants d'aujourd'hui ne comprendront jamais, 146, 238,  
 243 show()

animation, 68  
 graphics object, 64  
 simplify(), 31 sinus,  
 59

résoudre des solutions approximatives, 137–  
 139 solutions exactes, 132–  
 137 systèmes d'équations, 139  
 solve(), 132  
 solvabilité par radicaux, 137 .sort(),  
 114 sorted, 112  
 sorting, 112,  
 114 Spherical(), 220  
 square root, 26 str(),  
 235, 248  
 substitution, 33–35

dictionnaire, 33, 110  
 somme(),  
 124 différence\_symétrique(), 115  
 mise à jour de la différence\_symétrique(), 115  
 différence\_symétrique.différence\_symétrique(), 236 SyntaxError  
 impossible  
 d'assigner à l'opérateur, 31 syntaxe  
 non valide, 25, 31, 80, 155 le mot-clé ne  
 peut pas être une expression, 137

complétion par tabulation, 21, 113, 131, 142,  
 204 taylor(), 268  
 text(), 60  
 théorèmes

Théorème de division pour les entiers gaussiens, 181  
 Théorème de division pour les entiers, 181  
 Théorème de décomposition propre, 203  
 Théorème des valeurs intermédiaires, 152  
 Vrai, 26  
 essayer,  
 160 tuple, 109, 111, 112  
 commandes partagées avec toutes les collections, 112  
 commandes partagées avec la liste, 113  
 méthodes, 112  
 tuple(), 109 type

dynamique vs statique, 240  
 type(), 43  
 TypeError, 81  
 l'objet 'set' ne prend pas en charge l'indexation, 117 l'objet  
 'tuple' ne prend pas en charge l'affectation  
 d'éléments, 117  
 Tentative de coercition ..., 44  
 ne peut concaténer qu'une liste (pas un « tuple ») à une liste,  
 117 ne peut  
 concaténer qu'un tuple (pas une « liste ») à un tuple, 117

Impossible d'évaluer l'expression symbolique en une  
 valeur numérique., 138 La  
 multiplication d'une ligne par un élément de champ rationnel  
 ne peut pas être effectuée sur un anneau entier.  
 Utilisez change\_ring ..., 144  
 ou les matrices mutables ne sont pas hachables.,  
 142 Aucune coercition canonique de l'anneau symbolique vers  
 le champ rationnel., 205 ...  
 l'objet n'est pas appellable., 32 ... prend  
 au plus ... argument (... donné), 117 ... prend  
 exactement ...  
 argument (... donné), 74, 82 instance non hachable.,  
 238 110 type non  
 hachable : type(s) d'opérande  
 pris en charge., 32 ..., non

et,  
 36 .union(), 115  
 unsigned\_infinity, 37 .update(),  
 115, 116

Erreur de valeur  
 L'hypothèse est incohérente, 39  
 Le calcul a échoué car Maxima a demandé des contraintes  
 supplémentaires, 38, 39  
 L'intégrale est divergente., 38, 39 Les  
 nombres doivent être des entiers non négatifs, 235 Le  
 nom ... n'est pas un identifiant Python valide, 26

le nombre d'arguments, 34

variable, 28  
global vs. local, 89  
boucle variable, 118  
réinitialisation d'un nom,  
29 vecteur, 140,  
145 vecteur propre, 203  
propriété bien ordonnée, 185, 192  
`zero_matrix()`, 141  
ZeroDivisionError  
Division rationnelle par zéro, 166  
Division symbolique par zéro, `ordre`  
162 , 52, 53  
, voir entiers, anneau entier  
`ZZ`, 42, 144

