Calcul formel

Gaetan Bisson

https://gaati.org/bisson/

Introduction

Tout mathématicien a l'habitude de poser des petits calculs : pour $n = 3$, on sait facilement évaluer le produit de deux nombres à n chiffres ou encore inverser une matrice de taille $n \times n$. Lorsque n grandit, mener de tels calculs à bout devient impossible sans méthode rigoureuse. Décrite pas à pas, cette méthode peut être implantée sur ordinateur de sorte à pouvoir réaliser des calculs de tailles telles que leur exécution manuelle prendrait des milliards d'années.

Le calcul formel est l'étude de telles méthodes permettant de manipuler des objets mathématiques symboliques, c'est-à-dire de manière exacte, par opposition au calcul approché. Il est omniprésent dans la société numérique actuelle du fait des besoins importants en calculs de toute sorte ; la cryptographie en est notamment une importante application.

L'objectif de ce cours est comprendre les notions et techniques fondamentales du calcul formel. Ceci nous permettra d'illustrer diverses notions mathématiques vues en licence et de les revisiter sous un nouvel angle. Nous utiliserons le logiciel Sage pour mettre en pratique les concepts théoriques vus en cours.

Table des matières

1 Le logiciel Sage **4**
1.1 Opérations élémentaires 4
1.2 Obtenir de l'aide 5
1.3 Boucles et indentation 6
1.4 Fonctions et objets 8
1.5 Listes-tableaux 9
1.6 Exercices 10

2 Algorithmes et complexité **11**
2.1 Calcul matériel et symbolique 11
2.2 Calcul logiciel et complexité 12
2.3 Algorithmes de tri 14
2.4 Algorithmes sur les matrices 16

3 Arithmétique des entiers **17**
3.1 Représentation 17
3.2 Multiplication 18
3.3 Exponentiation 19
3.4 PGCD et Bézout 20
3.5 Primalité 21
3.6 Factorisation 22

4 Arithmétique des polynômes **27**
4.1 Représentation 27
4.2 Facteurs carrés 29
4.3 Racines réelles 30
4.4 Factorisation sur les corps finis 31

5 Calcul numérique **34**
5.1 Notions de fonctions 34
5.2 Limites 35
5.3 Zéros 36
5.4 Intégration 39
5.5 Équations différentielles 40
5.6 Tracés et animations 41

6 Applications en cryptographie 44
6.1 Chiffrement 44
6.2 Signature 44
6.3 Partage de clefs 44
6.4 Attaques 44

Bibliographie 46

Chapitre 1

Le logiciel Sage

Les logiciels de calcul formel commerciaux (dont MATLAB, Maple et Mathematica) présentent tous l'inconvénient majeur de n'offrir aucun accès à leur code source. Leurs utilisateurs sont ainsi dans l'impossibilité d'en étudier ou modifier le fonctionnement interne et, en particulier, de corriger, améliorer, ou de rajouter des fonctionnalités.

C'est pourquoi, de par le monde, des mathématiciens ont écrit des programmes offrant les fonctionnalités de calcul nécessaires à leurs travaux de recherche et ont choisi de les distribuer librement, c'est-à-dire sous une licence garantissant aux utilisateurs le droit d'étudier, de modifier et de redistribuer leur code source. C'est notamment le cas de PARI/GP [10] en théorie des nombres, de R [11] en statistiques, ou encore de Singular [12] en algèbre commutative. Le logiciel Sage [15] combine ces programmes et bien d'autres pour fournir une interface unifiée à un vaste spectre de fonctionnalités mathématiques ; elle se présente comme extension du langage de programmation Python.

Note. Ce chapitre est librement inspiré du tutoriel officiel [14].

1.1 Opérations élémentaires

Sage utilise le symbole = pour affecter une valeur à une variable. On peut donc taper :

```
sage: a = 2
sage: a
2
```

Il ne faut pas confondre l'affectation avec l'égalité, qui se note ==. Continuant le code ci-dessus, on a donc :

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

Outre les comparaisons, on peut naturellement effectuer les opérations arithmétiques évidentes :

```
sage: 1 + 2 + 3
6
sage: 1 + 2 * 3
7
sage: 1 + 2 ^ 3    # puissance
9
sage: (1 + 2) % 3  # reste
0
sage: (1 + 2) // 3 # quotient
1
```

Ces opérations ne sont pas limitées aux nombres entiers :

```
sage: 1 + 2 / 3
5/3
sage: 4 ** (3 / 2)
8
sage: 2 / 2 ** (1 / 2)
sqrt(2)
```

Remarquer que Sage ne fait aucune approximation : il manipule des nombres exacts comme $\sqrt{2}$ et non des valeurs approchées comme 1.414213562. On peut toutefois lui demander d'en calculer, soit en lui fournissant dès le départ une valeur inexacte, soit avec la fonction `n` :

```
sage: sqrt(2.0)
1.41421356237310
sage: N(sqrt(2))
1.41421356237310
sage: N(sqrt(2), digits=42)
1.41421356237309504880168872420969807856967
```

Sage s'efforce de rendre possible l'utilisation des notions ci-dessus avec toute structure algébrique pour laquelle elles ont du sens. Regardons notamment le cas des matrices :

```
sage: m = matrix([[1, 2], [2, 1]])
sage: m ** (-1)
[-1/3  2/3]
[ 2/3 -1/3]
sage: m.exp()
[1/2*(e^4 + 1)*e^(-1) 1/2*(e^4 - 1)*e^(-1)]
[1/2*(e^4 - 1)*e^(-1) 1/2*(e^4 + 1)*e^(-1)]
sage: m.charpoly()
x^2 - 2*x - 3
sage: m.charpoly().roots()
[(3, 1), (-1, 1)]
```

1.2 Obtenir de l'aide

La documentation de Sage est la source d'information la plus complète sur ce logiciel. Seuls les éléments du langage de programmation sous-jacent ne sont que brièvement présentés car ils relèvent du domaine de Python. Leurs documentations sont librement accessibles aux adresses :

http://www.sagemath.org/doc/

http://www.python.org/doc/

L'interface de Sage vient de surcroit avec une aide intégrée qui est probablement le moyen le plus commode d'obtenir des informations précises sur ses fonctionnalités : pour accéder au descriptif d'une fonction, il suffit de lui ajouter le suffixe « ? ».

Exemple.
```
Signature:      sqrt(x, *args, **kwds)
Docstring:
   INPUT:

   * "x" -- a number

   * "prec" -- integer (default: "None"): if "None", returns an exact
     square root; otherwise returns a numerical square root if
     necessary, to the given bits of precision.

   * "extend" -- bool (default: "True"); this is a placeholder, and is
     always ignored or passed to the "sqrt" method of "x", since in
     the symbolic ring everything has a square root.

   * "all" -- bool (default: "False"); if "True", return all square
     roots of "self", instead of just one.

   EXAMPLES:

      sage: sqrt(4)
      2
      sage: sqrt(4, all=True)
      [2, -2]

      sage: sqrt(-1)
      I
      sage: sqrt(2)
      sqrt(2)
      sage: sqrt(2)^2
      2
      sage: sqrt(x^2)
      sqrt(x^2)
```

1.3 Boucles et indentation

En Python, donc en Sage, c'est l'indentation du code qui délimite les blocs d'instructions. Cette délimitation est cruciale pour les structures de contrôles. Cette section décrit ce que cela signifie concrètement.

La structure de contrôle la plus simple est le branchement conditionnel, noté `if` ; il exécute un bloc de code si et seulement si une condition donnée est vérifiée. Par exemple :

```
sage: if 1 == 2:
....:     print(1)
....:     print(2)
....:
```

Comme la condition `1==2` n'est pas vérifiée, le bloc de code qui s'ensuit, c'est-à-dire l'ensemble des instructions indentées par rapport au `if`, n'est pas exécuté ; Sage n'affiche donc rien. En revanche, on a :

```
sage: if 1 == 2:
....:     print(1)
....: print(2)
....:
sage: 2
```

Dans ce cas, comme « `print(2)` » n'est pas indentée par rapport au `if`, cette instruction n'est pas conditionnée ; elle se contente donc d'être la suivante après `if` dans la liste des commandes à exécuter.

La boucle est une structure de contrôle qui exécute un même bloc de code pour plusieurs valeurs d'une variable donnée. Sa forme la plus simple est « `for x in L:` » suivie du bloc de code à exécuter, où `L` désigne la liste des valeurs à donner successivement à la variable `x`. Par exemple, pour afficher les carrés des nombres de 1 à 9, on peut faire :

```
sage: list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: for i in range(1, 10):
....:     print(i ** 2)
....:
1
4
9
16
25
36
49
64
81
```

Et pour calculer la factorielle de 42, on peut faire :

```
sage: f = 1
sage: for i in range(1, 43):
....:     f = f * i
....:
sage: f
1405006117752879898543142606244511569936384000000000
```

Le mot clef `while` permet quant à lui d'exécuter un bloc de code tant qu'une condition est satisfaite. Par exemple, pour calculer la partie entière du logarithme en base deux de 1234 on pourrait faire :

```
sage: r = 1234
sage: n = 0
sage: while r > 1:
....:     r = r // 2
....:     n = n + 1
....:
sage: n
10
```

Exercice. *Calculer la somme des carrés des entiers entre* 1 *et* 100.

Exercice. *Calculer une approximation de la quantité* $\sum_{k=1}^{1000} \frac{(-1)^k}{k}$.

Exercice. *Déterminer les entiers* $n \in \{1, \ldots, 80\}$ *pour lesquels le nombre* $2^n - 1$ *est premier.*

1.4 Fonctions et objets

On définit une fonction grâce à la commande `def` comme il suit :

```
sage: def impair(n):
....:     return n % 2 == 1
....:
sage: impair(3)
True
```

On peut aussi combiner les mots clefs que nous avons vu, par exemple, pour définir la factorielle :

```
sage: def factorielle(n):
....:     r = n
....:     for i in range(2, n):
....:         r = r * i
....:     return r
....:
sage: factorielle(42)
1405006117752879898543142606244511569936384000000000
```

Noter que Sage accepte aussi les définitions récursives (même si leur exécution est plus coûteuse à gérer pour le logiciel que les versions déroulées comme ci-dessus) :

```
sage: def factorielle(n):
....:     if n < 2:
....:         return 1
....:     return n * factorielle(n - 1)
```

Lorsqu'un utilisateur comme nous définit une fonction, il ne précise pas quel type d'arguments elle attend ni quel type de résultat elle renvoie. Pour de petites fonctions dont on contrôle les appels, cela ne pose aucun problème. En revanche, pour du code gros et complexe, spécifier le type des arguments et du résultat permet d'éviter de nombreux problèmes. En tant que langage *orienté objet*, Python va plus loin et regroupe l'ensemble des fonctions s'appliquant à un type d'arguments donné. On peut y accéder en ajoutant le suffixe « `.` » à l'argument. Voici par exemple toutes les fonctions que l'on peut appliquer à des rationnels :

```
sage: x = 1 / 2
sage: x.[TAB]
x.N               x.base_extend        x.ceil       x.db
x.abs             x.base_ring          x.charpoly   x.denom
x.absolute_norm   x.cartesian_product  x.conjugate  x.denominator
x.additive_order  x.category           x.content    ...
sage: x.denominator()
2
```

Exercice. *Écrire une fonction qui calcule le* n^e *terme de la suite de Fibonacci.*

1.5 Listes-tableaux

En Sage, les listes elles aussi sont héritées de Python ; elle sont un type hybride qui présente les caractéristiques des structures traditionnelles que sont les listes chaînées et les tableaux. Nous les avons déjà effleurées car c'est ce que renvoit la fonction `range`.

On crée des listes arbitraires en notant leurs éléments entre crochets. On peut accéder à tout élément en mettant son indice entre crochets après le nom de la liste (attention, les indices commencent à zéro). Enfin, la longueur d'une liste s'obtient grâce à la commande `len`. On a donc :

```
sage: y = [2, "trois", 5.0]
sage: len(y)
3
sage: y[1]
"trois"
sage: y[2]
5.0
sage: y[3]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-cb46143f779d> in <module>()
----> 1 y[3]

IndexError: list index out of range
```

On peut modifier une liste en changeant l'un de ses éléments, en y rajoutant ou en y enlevant un élément.

```
sage: y = [1, 4, 6]
sage: y.append(8)
sage: y[0] = 2
sage: y.pop()
8
sage: y.pop()
6
sage: y
[2, 4]
```

On peut aussi utiliser l'opérateur de concaténation pour mettre bout-à-bout deux listes mais il faut être conscient que, contrairement aux opérations précédentes, cela crée une nouvelle liste et ne modifie pas celles existantes.

```
sage: y + [6]
[2, 4, 6]
sage: y
[2, 4]
```

Exercice. *Écrire une fonction qui renvoie la liste des diviseurs d'un entier.*
Écrire une fonction qui calcule la somme d'une liste. Pareil pour l'écart-type.
En déduire la liste des entiers $x < 10^6$ dont la somme des diviseurs vaut $2x$.

Exercice. *Écrire une fonction qui renverse une liste, c'est-à-dire renvoie une liste comportant les mêmes éléments mais en ordre inverse.*

On peut alternativement créer des listes en utilisant une boucle `for` interne :

```
sage: [x ** 2 for x in range(1, 10)]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
sage: [n for n in range(1, 1000) if is_prime(2 ** n - 1)]
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607]
```

1.6 Exercices

Exercice. *La fonction* $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *dite d'Ackermann est définie par*

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0, \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0, \\ A(m-1, A(m, n-1)) & \text{si } m > 0 \text{ et } n > 0. \end{cases}$$

Calculer $A(k,k)$ *pour* $k = 1, 2, 3 \ldots$

Exercice. *Une suite* u *est dite de Syracuse si, pour tout entier positif* k*, elle vérifie*

$$u_{k+1} = \begin{cases} u_k/2 & \text{si } u_k \bmod 2 = 0, \\ 3u_k + 1 & \text{si } u_k \bmod 2 = 1. \end{cases}$$

Calculer les cent premiers termes de la suite de Syracuse commençant par 3*. Faire de même pour* 7*. Que remarque-t-on ? Écrire une fonction qui détermine le premier indice pour lequel la suite de Syracuse commençant par* n *atteint* 1*. Comment évolue cette valeur lorsque* n *grandit ?*

Exercice. *L'écriture d'un entier en base* n *héréditaire est similaire à l'écriture en base* n *classique sauf que les exposants y sont aussi écrits en base* n*, ainsi que les exposants des exposants, etc. Par exemple, calculons l'écriture de* 42 *en base* 2 *héréditaire :*

$$\begin{aligned} 42 &= 2^5 + 2^3 + 2^1 \\ &= 2^{2^2+1} + 2^{2^1+1} + 2^1 \\ &= 2^{2^{2^1}+1} + 2^{2^1+1} + 2^1 \end{aligned}$$

Une suite u *est dite de Goodstein si* u_{n+1} *s'obtient en écrivant* u_n *en base* n *héréditaire, remplaçant tous les* n *par des* $n+1$*, et soustrayant* 1*. Par exemple, si* $u_1 = 42$*, alors*

$$\begin{aligned} u_3 &= 3^{3^{3^1}+1} + 3^{3^1+1} + 3^1 - 1 \\ &= 22876792455044 \end{aligned}$$

Calculer les dix premiers termes des suites de Goodstein commençant par 1*,* 2 *et* 3*. Que se passe-t-il pour* 4 *?*

Chapitre 2

Algorithmes et complexité

L'aperçu de Sage du chapitre précédent donne une impression trompeuse de simplicité : on y a manipulé des entiers, des rationnels et des nombres algébriques avec une élégance uniforme. En arriver là demande cependant un énorme travail de la part de Sage, des bibliothèques sur lesquelles il repose et de Python.

Pour fonctionner, un programme ne peut faire appel qu'à un petit nombre d'instructions du microprocesseur ; c'est en les combinant méthodiquement qu'il arrive à offrir des fonctionnalités plus complexes. Le nombre d'instructions élémentaires utilisées par un programme est une bonne approximation du temps de calcul nécessaire à son exécution.

2.1 Calcul matériel et symbolique

Les constructeurs munissent les microprocesseurs d'instructions permettant de calculer certains ensembles d'opérations usuelles, mais sont évidemment limités par la finitude des ressources disponibles, notamment en termes de capacité de stockage et de transmission des données. Ils optimisent leurs circuits pour que l'exécution de chaque instruction prenne un temps essentiellement constant. On distingue trois grandes classes d'instructions :

— **Les opérations logiques** sont exactes et portent sur des éléments de $\{0, 1\}^{64}$; il s'agit notamment des (puissances cartésiennes de) la conjonction, la disjonction, la négation et le décalage.

— **Les opérations arithmétiques** imitent l'arithmétique de $\mathbb{Z}$ mais, en réalité, réalisent celle de $\mathbb{Z}/2^{64}\mathbb{Z}$. En particulier, le produit de deux grands entiers déborde.

— **Les opérations numériques** imitent l'arithmétique de $\mathbb{R}$ mais avec une précision limitée à 64 bits significatifs. Notamment, la suite des puissances de $1/2$ est donc constante à partir d'un certain rang.

Il est naturellement bien plus rapide d'effectuer une opération arithmétique que d'effectuer, sur le même processeur, la chaîne d'opérations logiques permettant d'arriver au même résultat. De manière générale, l'implantation matérielle d'un calcul a une vitesse d'exécution plus élevée que son implantation logicielle, mais aussi un coût bien plus important.

Chaque programme utilise les opérations matérielles qu'il juge adaptées à ses besoins. Par exemple, les logiciels de bureautique n'utilisent que des entiers bien inférieurs à 2^{64} et se contentent donc d'opérations arithmétiques. Les applications de rendu graphique n'ont quant à elles rarement besoin de précision au delà du pixel et utilisent ainsi majoritairement les opérations numériques.

2.2 Calcul logiciel et complexité

Les processeurs offrent donc un jeu minimal d'instructions et c'est en les combinant méthodiquement que les programmes parviennent à réaliser des calculs complexes. Pour les besoins de ce cours, nous ignorerons toute autre considération comme par exemple le coût des appels mémoire. Une approche plus rigoureuse consisterait à utiliser un modèle comme les machines de Turing [3] afin de formaliser la notion de calcul sur ordinateur mais elle nous conduirait à des résultats globalement similaires.

En première approximation, on peut supposer que chaque instruction processeur s'exécute en temps constant. Ainsi le nombre de telles instructions utilisées par un programme serait proportionnel à son temps d'exécution. L'objectif de cette section est d'évaluer cette quantité et pour cela nous préférerons parler d'algorithmes que de programmes ; un algorithme est l'essence d'un programme indépendamment du langage de programmation dans lequel il est écrit ainsi que des détails d'implémentation ayant un impact nul ou négligeable sur son exécution, par exemple le nom des variables.

Rappelons tout d'abord une notation de Landau.

Définition. *Soient f et g deux fonctions d'un ensemble quelconque X dans $\mathbb{R}$. On dit que f est dominée par g et on note $f = O(g)$ lorsque la quantité f/g est bornée.*

Le nombre d'instructions du processeur utilisées par l'exécution d'un algorithme dépend évidemment des valeurs des arguments de cet algorithme et n'est pertinent qu'à un facteur de proportionnalité inconnu près. Nous n'étudierons ainsi que son comportement asymptotique à un « grand O » près.

Définition. *Soit $\mathscr{A}$ un algorithme prenant en argument un paramètre $x \in X$ et notons $f(x)$ le nombre d'instructions du processeur utilisées par l'exécution de $\mathscr{A}(x)$. On appelle borne de complexité de $\mathscr{A}$ toute fonction $g : X \to \mathbb{R}$ vérifiant $f = O(g)$.*

En l'absence d'appels récursifs, une borne de complexité fine s'obtiendra simplement par comptage du nombre d'instructions en prenant bien soin de dérouler les boucles.

Exemple. *Les complexités des programmes ci-dessous sont respectivement $O(1)$ et $O(n)$ et $O(n^2)$.*

```
def carre0(n):
    return n * n

def carre1(n):
    r = 0
    for i in range(n):
        r = r + n
    return r

def carre2(n):
    r = 0
    for i in range(n):
        for i in range(n):
            r = r + 1
    return r
```

Lorsqu'à l'instar de l'exemple ci-dessus le nombre d'instructions par itération est constant, on peut se contenter de le multiplier par le nombre d'itérations (c'est-à-dire la longueur de la boucle) afin d'obtenir le nombre total d'instructions de la boucle. Quand ce n'est pas le cas on est réduit à sommer le nombre d'instructions pour chaque itération.

Exemple. *Considérons le programme :*

```
def sommecarre(n):
    x = 0
    for k in range(n):
        x = x + carre1(k)
    return x
```

La complexité du programme « `carre1(k)` *» ci-dessus étant $O(k)$, c'est le nombre d'instructions par itération. Le total est donc de $\sum_{k=1}^{n} O(k)$. Le facteur multiplicatif implicite du grand-O étant le même pour chaque terme, on peut l'intervertir avec la somme. On obtient alors comme borne de complexité $O(\sum_{k=1}^{n} k) = O(\frac{n(n+1)}{2}) = O(n^2)$.*

Remarquer qu'on aurait obtenu la même borne si on avait grossièrement majoré par $O(n)$ le nombre d'instructions par itération, puis multiplié par la longueur de la boucle. Parfois, cette approche plus simple donnera cependant des bornes grossières.

En présence d'appels récursifs, on procèdera en deux temps. D'abord, on écrira une relation de récurrence exprimant la complexité pour un paramètre donné en fonction de celles pour de plus petits paramètres. Ensuite, on résoudra cette relation de récurrence afin d'obtenir le terme général de la complexité. Si cela s'avère difficile on pourra remplacer la relation par un majorant simple, au risque d'obtenir une borne non fine.

Exemple. *Notons c_b la complexité pour $b \in \mathbb{N}$ du programme :*

```
def multiplication(a,b):
    if b = 0:
        return 0
    else:
        return a + multiplication(a, b - 1)
```

Elle vérifie $c_b = 3 + c_{b-1}$ et $c_0 = 1$ d'où l'on déduit facilement $c_b = 3b + 1 = O(b)$.

Plus généralement, rappelons qu'une suite $(s_n)_{n\in\mathbb{N}}$ est dite arithmético-géométrique lorsqu'elle satisfait une relation de récurrence du type $s_{n+1} = a s_n + b$; son terme général s'écrit alors :

$$s_n = a^n \left(s_0 - \frac{b}{1-a} \right) + \frac{b}{1-a}$$

Exercice. *Donner la complexité de tous les programmes implantés précédemment.*

Considérons à présent les programmes suivants.

```
def fibo1(n):
    if n < 2:
        return n
    else:
        return fibo1(n - 1) + fibo1(n - 2)

def fibo2(n):
    (u, v) = (0, 1)
    for k in range(n):
        (u, v) = (v, u + v)
    return u
```

```
def fibo3_mul(a, b):
    return [
        a[0] * b[0] + a[1] * b[2],
        a[0] * b[1] + a[1] * b[3],
        a[2] * b[0] + a[3] * b[2],
        a[2] * b[1] + a[3] * b[3],
    ]

def fibo3_pow(x, n):
    r = [1, 0, 0, 1]
    while n > 0:
        if n % 2:
            r = fibo3_mul(r, x)
        x = fibo3_mul(x, x)
        n = n // 2
    return r

def fibo3(n):
    a = [0, 1, 1, 1]
    return fibo3_pow(a, n)[1]
```

Mesurons le temps de calcul de ces programmes via la fonction `time()`.

```
sage: time(fibo1(30))
CPU times: user 0.32 s, sys: 2 ms, total: 0.34 s
Wall time: 0.35 s
832040
```

Supposant que les opérations arithmétiques s'effectuent en temps constant, ce qui serait notamment le cas si les calculs étaient réalisés modulo un entier fixé, les complexités des programmes `fibo1`, `fibo2` et `fibo3` sont respectivement $O(\varphi^n)$, $O(n)$ et $O(\log(n))$, avec $\varphi = \frac{1+\sqrt{5}}{2}$. Par une simple règle de proportionnalité, on peut alors estimer le temps de calcul nécessaire à l'exécution de ces programmes pour de grands paramètres ; voir la figure 2.1.

Remarque. *Le temps n'est pas la seule ressource dont un ordinateur ne dispose qu'en quantité limitée. À tout moment, sa mémoire ne peuvent en effet stocker qu'une quantité finie d'information. Il pourra donc être opportun dans certaines situations de considérer la notion de complexité en mémoire parallèlement à celle de complexité en temps.*

2.3 Algorithmes de tri

Comment trieriez-vous une liste d'entiers ?

Tri par sélection. Une manière de procéder est d'identifier le plus petit élément, le mettre en première position, puis d'identifier le plus petit élément restant, le mettre en seconde position, etc. On écrira d'abord une fonction identifiant le plus petit élément d'une liste.

```
def tri_selection(x):
    for i in range(len(x)):
        (k, m) = (i, x[i])
        for j in range(i + 1, len(x)):
            if m > x[j]:
                (k, m) = (j, x[j])
        (x[i], x[k]) = (x[k], x[i])
    return x
```

	`fibo1`		`fibo2`		`fibo3`	
	COMPLEXITÉ	TEMPS	COMPLEXITÉ	TEMPS	COMPLEXITÉ	TEMPS
n	1.618^n		n		$\log(n)$	
10	10^2	0.00003 s				
20	10^4	0.003 s				
30	10^6	0.3 s				
40	10^8	30 s				
50	10^{10}	50 min				
10^2	10^{21}	10^7 ans	10^2	0.00002 s		
10^3			10^3	0.0002 s		
10^4			10^4	0.002 s		
10^5			10^5	0.02 s	5	0.00005 s
10^{10}			10^{10}	30 min	10	0.0001 s
10^{20}			10^{20}	10^6 ans	20	0.0002 s
10^{10^2}					10^2	0.001 s
10^{10^3}					10^3	0.01 s
10^{10^4}					10^4	0.1 s

FIGURE 2.1 – Complexité et estimation du temps d'exécution de différents algorithmes calculant le n^e terme de la suite de Fibonacci modulo un entier fixé.

La complexité de ce programme est $O(n^2)$ où n dénote la longueur de la suite à trier. C'est-à-dire qu'on compare essentiellement tous les couples d'éléments de cette liste, ce qui est loin d'être optimal.

Tri par insertion. Implanter puis étudier pareillement la méthode consistant à insérer l'élément d'indice k dans la liste formée des éléments d'indice $\{0, \ldots, k-1\}$ que l'on supposera triée. Itérer ce processus pour k allant de 1 à $n-1$.

Tri fusion. Nous allons définir un tri de manière récursive. Supposons qu'on sache trier les listes de longueur au plus n ; comment peut-on alors en trier des plus longues ? Si une liste est de longueur au plus $2n$, on peut la découper en deux sous listes que l'on saura alors trier ; il s'agit ensuite de fusionner ces deux sous listes triées en la liste complète triée.

Écrivons donc d'abord un programme qui, étant donné deux listes triées, renvoie leur concaténation triée.

```
def fusion(x, y):
    z = []
    (i, j) = (0, 0)
    while i < len(x) or j < len(y):
        if i == len(x):
            z.append(y[j])
            j += 1
        elif j == len(y):
            z.append(x[i])
            i += 1
        elif x[i] > y[j]:
            z.append(y[j])
            j += 1
```

```
        else:
            z.append(x[i])
            i += 1
    return z
```

On peut alors définir notre tri par récurrence :

```
def tri_fusion(w):
    if len(w) < 2:
        return w
    x = [w[i] for i in range(len(w) // 2)]
    y = [w[i] for i in range(len(w) // 2, len(w))]
    x = tri_fusion(x)
    y = tri_fusion(y)
    z = fusion(x, y)
    return z
```

Notant c_n la complexité de la fonction `tri_fusion` en fonction de la longueur n de la liste donnée, on a la relation $c_n = 2c_{\lceil n/2 \rceil} + O(n)$. Montrer qu'on obtient alors la forme explicite $c_n = O(n \log(n))$. C'est la complexité optimale pour trier une liste de longueur n sur laquelle on ne dispose d'aucune information a priori.

2.4 Algorithmes sur les matrices

On représente une matrice $(m_{ij})_{(i,j)\in\{1,\dots,n\}^2} \in \mathcal{M}_n(\mathbb{R})$ comme une liste M de listes de nombres flottants, le coefficient m_{ij} étant stocké comme `M[i-1][j-1]`.

Implanter des fonctions effectuant les calculs suivantes, évaluer leur complexité et identifier la taille maximale n qu'elles permettent de traiter en moins d'une heure.

1. Calculer la trace d'une telle matrice.
2. Calculer le produit de deux telles matrices.
3. Calculer le déterminant d'une telle matrice.
4. Calculer l'inverse d'une telle matrice.

Chapitre 3

Arithmétique des entiers

Nous avons vu que les microprocesseurs n'implantent l'arithmétique que de $\mathbb{Z}/2^{64}\mathbb{Z}$. Ce chapitre donne un aperçu des techniques mises en œuvre par les logiciels comme Sage afin de représenter fidèlement l'anneau $\mathbb{Z}$. Pour un ouvrage de référence exhaustif sur ce domaine, consulter [16].

3.1 Représentation

Sage permet de travailler directement avec des entiers machines :

```
sage: from ctypes import c_ulong
sage: c_ulong(2 ** 63)
c_ulong(9223372036854775808)
sage: c_ulong(2 ** 64)
c_ulong(0)
```

Leur arithmétique est celle de $\mathbb{Z}/2^{64}\mathbb{Z}$; elle permet de calculer de manière exacte l'addition et la multiplication d'entiers de $\{0, \ldots, 2^{32} - 1\}$. On représentera des entiers arbitrairement grands comme vecteurs d'entiers machines en s'appuyant sur le résultat suivant.

Théorème. *Soit $B \geqslant 2$ un entier fixé. Pour tout entier naturel $n \in \mathbb{N}$ il existe une unique famille $x \in \{0, \ldots, B-1\}^r$ vérifiant $n = \sum_{i=0}^{r-1} x_i B^i$ et $x_{r-1} \neq 0$. Cette famille s'appelle la décomposition de n en base B.*

Toute la difficulté de l'arithmétique multiprécision consiste alors à réduire les opérations arithmétiques portant sur n en une suite d'instructions processeurs portant sur les x_i.

Remarque. *C'est exactement ce que nous faisons lorsque nous calculons à la main sur des entiers décimaux ! Tout problème auquel nous pourrions faire face peut donc être attaqué avec du papier et un crayon pour $B = 10$ et de petites valeurs de n ; il suffira ensuite de généraliser la méthode obtenue à $B = 2^{32}$ et n arbitraire.*

Commençons par un petit échauffement sans grande difficulté.

Exercice. *Écrire une fonction prenant en argument un entier B et une liste L et renvoyant l'entier dont la décomposition en base B est L.*

Écrire une fonction prenant en argument un entier B et un entier n et renvoyant la décomposition de n en base B.

Exercice. *Écrire une fonction comparant deux entiers représentés ainsi ; elle renverra* $+1$ *(respectivement* -1*) si l'entier représenté par son premier argument est plus petit (respectivement plus grand) que celui représenté par le second, et* 0 *en cas d'égalité.*

(On pourra tester les programmes plus facilement pour $B = 10$*.)*

L'addition de deux entiers représentés ainsi s'effectue de manière naturelle : terme à terme, en prenant soin de propager les retenues des coordonnées dont la valeur dépasse $B - 1$. En Sage, on peut écrire :

```
B = 2 ** 32

def addition(x, y):
    m = len(x)
    n = len(y)
    z = [0] * (1 + max(m, n))
    for i in range(len(z)):
        if i < m:
            z[i] += x[i]
        if i < n:
            z[i] += y[i]
        if z[i] >= B:
            z[i + 1] += z[i] // B
            z[i] = z[i] % B
    return z
```

Exercice. *Écrire une fonction qui calcule la différence de deux entiers représentés comme vecteurs d'entiers machines. On pourra supposer que le premier entier est supérieur au second.*

3.2 Multiplication

La multiplication peut s'effectuer en exploitant la formule classique

$$\sum_{i=0}^{n-1} x_i B^i \cdot \sum_{j=0}^{m-1} x_j B^j = \sum_{k=0}^{nm-1} \left(\sum_{i+j=k} x_i x_j \right) B^k$$

mais le problème des retenues est moins évident que pour l'addition. Le programme ci-dessous implante cette formule et propage les retenues des vecteurs formés d'au plus 2^{31} entiers machines.

```
def multiplication(x, y):
    m = len(x)
    n = len(y)
    z = [0] * (1 + m + n)
    for i in range(len(x)):
        for j in range(len(y)):
            k = i + j
            z[k] += x[i] * y[j]
            while z[k] >= B:
                z[k + 1] += z[k] // B
                z[k] = z[k] % B
                k += 1
    return z
```

Exercice. *Quelle est la complexité de cet algorithme en fonction de* `len(x)` *et* `len(y)` *?*

Afin de calculer $(aB+b)\cdot(cB+d) = \left(acB^2 + (ad+bc)B + bd\right)$ l'algorithme ci-dessus réalise quatre multiplications d'entiers machines. Cependant, une fois ac et bd calculés, le terme $ad+bc$ s'obtient en seulement une multiplication sous la forme $(a+b)(c+d) - ac - bd$; comme la multiplication est une opération plus coûteuse que l'addition, il est avantageux d'exploiter cette formule qui utilise trois multiplications plutôt que quatre. L'itération de cette technique sur des entiers multiprécision s'appelle la méthode de multiplication de Karatsuba et a pour complexité $O(n^{\log_2 3})$.

```
def karatsuba(x, y):
    k = len(x) // 2
    if k < 3:
        return multiplication(x, y)
    x0 = x[0:k]
    y0 = y[0:k]
    x1 = x[k : len(x)]
    y1 = y[k : len(y)]
    z0 = karatsuba(x0, y0)
    z2 = karatsuba(x1, y1)
    z1 = karatsuba(addition(x0, x1), addition(y0, y1))
    z1 = soustraction(soustraction(z1, z0), z2)
    z = (len(x) + len(y)) * [0]
    for i in range(len(z)):
        if i in range(len(z0)):
            z[i] += z0[i]
        if i - k in range(len(z1)):
            z[i] += z1[i - k]
        if i - 2 * k in range(len(z2)):
            z[i] += z2[i - 2 * k]
        if z[i] >= B:
            z[i + 1] += z[i] // B
            z[i] = z[i] % B
    return z
```

Remarque. *L'algorithme de Schönhage–Strassen [4] exploite la transformée de Fourier dans les corps finis afin de multiplier deux entiers de n bits en temps* $O\left(n\log(n)\log(\log(n))\right)$*; des avancées successives débouchèrent récemment [17] sur l'obtention d'un algorithme de complexité asymptotique* $O(n\log(n))$*, quantité conjecturée optimale.*

3.3 Exponentiation

Pour calculer la puissance k^{e} d'un entier x, on peut multiplier x avec lui-même $k-1$ fois comme il suit :

```
def puissance(x, k):
    r = 1
    for i in range(k):
        r = r * x
    return r
```

Exercice. *Quel est la complexité du programme ci-dessus ?*

Remarque. *Rappelez vous que c'est l'opération élémentaire de multiplication de deux entiers dans $\mathbb{Z}/2^{64}\mathbb{Z}$ qui s'exécute en temps constant, pas la multiplication d'entiers arbitraires dans $\mathbb{Z}$. Faites toujours bien attention à la taille des entiers avec lesquels vous travaillez si elle n'est pas bornée.*

Si l'exposant est une puissance de deux, on peut calculer la puissance bien plus rapidement en calculant x^2 puis $(x^2)^2$ puis $((x^2)^2)^2$, etc. Cette méthode marche aussi pour des exposants entiers arbitraires en les décomposant en base deux ; cela donne le programme suivant :

```
def puissance(x, k):
    r = 1
    y = x
    m = k
    while m > 0:
        if m % 2 == 1:
            r = r * y
        m = m // 2
        y = y * y
    return r
```

Exercice. *Quel est la complexité du programme ci-dessus ?*

Lorsque ce n'est pas le résultat entier qui nous intéresse, mais seulement le résultat modulo un entier p, il est évidemment intéressant de réduire les valeurs de r et y modulo p à chaque itération de sorte à ce que ces entiers restent de taille bornée.

3.4 PGCD et Bézout

Pour obtenir le plus grand diviseur commun (PGCD) de deux entiers, on peut calculer l'ensemble des diviseurs de chacun de ces entiers (stocké sous forme de liste) puis calculer leur intersection.

Exercice. *Sachant qu'un entier n a typiquement $O(\log^{\log 2} n)$ diviseurs, quelle est la complexité de cette méthode ?*

Une méthode plus rapide qui n'utilise pas de mémoire est l'algorithme d'Euclide : il consiste à calculer le reste x_2 de la division euclidienne de x_0 par x_1, puis le reste x_3 de la division de x_2 par x_1, etc. Le PGCD des deux entiers originaux est la dernière valeur de cette suite avant qu'elle se stabilise à zéro.

```
def euclide(x, y):
    while y > 0:
        (x, y) = (y, x % y)
    return x
```

Exercice. *Démontrer la correction du programme ci-dessus en trouvant un invariant de boucle adéquat. Quel est sa complexité ?*

On peut aussi calculer les coefficients de Bézout u et v pour lesquels $ux + vy = \mathrm{pgcd}(x, y)$. Ceci permet de calculer l'inverse de x modulo y, lorsqu'il existe, c'est-à-dire lorsque leur PGCD est l'unité. La méthode pour ce faire s'appelle l'algorithme d'Euclide étendu.

```
def bezout(a, b):
    (x, u, v) = (a, 1, 0)
    (y, s, t) = (b, 0, 1)
```

```
    while y > 0:
        q = x // y
        (x, y) = (y, x - q * y)
        (u, s) = (s, u - q * s)
        (v, t) = (t, v - q * t)
    return (x, u, v)
```

On pourra en démontrer la correction en observant que les égalités $x = au + bv$ et $y = as + bt$ sont vérifiées en début et fin de chaque itération.

3.5 Primalité

Si p est un nombre premier, nous savons à présent effectuer toutes les opérations élémentaires de $\mathbb{Z}/p\mathbb{Z}$, à savoir l'addition, la multiplication, l'exponentiation modulaire et l'inversion modulaire (grâce aux coefficients de Bézout). Pour construire ces anneaux, il nous reste à déterminer si un nombre n donné est premier.

La méthode naïve pour ce faire consiste à vérifier qu'aucun entier $d \leqslant \sqrt{n}$ autre que 1 ne divise n ; sa complexité est linéaire en n, c'est-à-dire exponentielle en sa taille $\log(n)$, et elle n'est donc efficace que pour de petits entiers :

```
def premier(n):
    if n < 2:
        return False
    for d in range(2, sqrt(n) + 1):
        if n % d == 0:
            return False
    return True
```

Parmi les méthodes permettant de tester la primalité en temps polynomial, on ignore en pratique celles qui sont déterministes [13] au profit d'algorithmes probabilistes plus rapides exploitant le fait suivant.

Proposition (dit *petit théorème de Fermat*). *Pour tout nombre premier p et tout entier relatif x on a l'égalité $x^p = x \bmod p$.*

La réciproque est fausse : il existe de (rares) entiers composés n (dits de Carmichael) tels que pour tout x on ait $x^n = x \bmod n$, par exemple $n = 561$. Un léger raffinement du théorème ci-dessus suffit toutefois à obtenir une équivalence.

Théorème (Miller [6]). *L'anneau $\mathbb{Z}/n\mathbb{Z}$ est un corps si et seulement si tous ses éléments non nuls sont des racines de l'unité.*

Cependant, lorsque n est composé, la densité des non racines de l'unité n'est pas suffisamment élevée pour permettre de conclure rapidement. Il suffit cependant d'y rajouter les témoins du surnombre de racines du polynôme $X^2 - 1$; on obtient le critère suivant :

Théorème (Rabin [8]). *Pour $n > 4$, l'ensemble des entiers $x \in \{1, 2, \ldots, n-1\}$ vérifiant*

$$x^{n-1} \neq 1 \bmod n \qquad \vee \qquad \exists k \in \mathbb{N}^\star, 2^k \mid n-1 \;\wedge\; x^{\frac{n-1}{2^k}} = 1 \bmod n \;\wedge\; x^{\frac{n-1}{2^{k-1}}} \neq \pm 1 \bmod n$$

est vide lorsque n est premier et de cardinal au moins $\frac{3}{4}(n-1)$ lorsque n est composé.

En écrivant la condition d'existence sur k de manière plus commode à évaluer de manière itérative, on obtient l'algorithme ci-dessous, appelé test de primalité de Miller–Rabin.

```
def premier(n):
    if n < 5:
        return [False, False, True, True, False][n]
    (t, s) = (n - 1, 0)
    while t % 2 == 0:
        (t, s) = (t // 2, s + 1)
    for i in range(floor(log(n))):
        x = randrange(2, n)
        x = power_mod(x, t, n)
        for k in range(s):
            y = power_mod(x, 2, n)
            if y == 1 and x != 1 and x != n - 1:
                return False
            x = y
        if x != 1:
            return False
    return True
```

Proposition. *Si n est premier, ce programme renvoie systématiquement `True`. Si n est composé, ce programme renvoie `False` avec probabilité* $1 - 4^{-\lfloor \log n \rfloor} \sim 1 - \frac{1}{n^2}$.

Exercice. *L'un des deux entiers* 51991 *et* 51997 *n'est pas premier ; lequel ?*

3.6 Factorisation

Comprendre la structure de l'anneau $\mathbb{Z}/n\mathbb{Z}$ revient à factoriser l'entier n ; on a alors $\mathbb{Z}/n\mathbb{Z} = \prod \mathbb{Z}/p^{v_p(n)}\mathbb{Z}$ ce qui nous ramène à une arithmétique déjà implantée. Cette simple motivation donne un aperçu de la portée du problème de la factorisation. Il a suscité un vif intérêt et un développement actif ces dernières décennies. Nous n'avons dans ce cours la prétention que d'en donner un bref aperçu.

Exercice. *Implanter une méthode naïve de factorisation. La fonction Python prendra en argument un entier positif et renverra la liste de ses facteurs premiers. En analyser alors la complexité.*

```
def factorisation(n):
    L = []
    for p in range(2, n + 1):
        if is_prime(p):
            while n % p == 0:
                L.append(p)
                n = n // p
    return L
```

Le problème de la factorisation peut en réalité se ramener à celui de la recherche de facteur. Cette observation nous permet d'adopter une approche plus générique et plus modulaire qui va faciliter notre étude de ce sujet. On écrira donc :

```
def facteur(n):
    for f in range(2, floor(sqrt(n)) + 1):
        if n % f == 0:
            return f
```

```
def factorisation(n):
    if n == 1:
        return []
    if is_prime(n):
        return [n]
    f = facteur(n)
    return factorisation(f) + factorisation(n // f)
```

Lemme. *La complexité de `factorisation(n)` est au plus $\log(n)$ fois celle de `facteur(n)`.*

Comme c'est le cas pour la primalité, les meilleurs algorithmes de factorisation sont probabilistes. Commençons donc par considérer une version probabiliste du programme ci-dessus.

```
def facteur(n):
    while True:
        f = randrange(2, floor(sqrt(n) + 1))
        if n % f == 0:
            return f
```

Si n n'est pas premier il admet un facteur $p \leqslant \sqrt{n}$. La probabilité de le trouver lors d'une itération est $\frac{1}{\sqrt{n}-1}$. Pour le trouver avec probabilité au moins un demi, il faut donc effectuer au plus k itérations où :

$$\begin{aligned} & 1-\left(1-\frac{1}{\sqrt{n}-1}\right)^k \geqslant \frac{1}{2} \\ \iff \quad & k\log\left(1-\frac{1}{\sqrt{n}-1}\right) \leqslant -\log(2) \\ \implies \quad & k\left(-\frac{1}{\sqrt{n}}+o\left(\frac{1}{\sqrt{n}}\right)\right) \leqslant -\log(2) \end{aligned}$$

Soit encore $k \geqslant \sqrt{n}\log(2)(1+o(1))$. Comme nous pouvions nous en douter, l'approche naïve probabiliste est donc d'efficacité essentiellement identique à l'approche naïve déterministe. Dans l'objectif d'atteindre de meilleures complexités, rappelons certains résultats bien connus du cours de probabilité.

Proposition. *Soit un réel $\alpha > 0$. En tirant $\alpha\sqrt{n}$ éléments de manière uniforme avec remise dans un ensemble de cardinal n, la probabilité qu'un élément ait été choisi deux fois converge vers $1 - e^{-\alpha^2/2}$ lorsque n tend vers l'infini.*

Démonstration. Les tirages sans répétition correspondent aux fonctions injectives de $\{1,\dots,\alpha\sqrt{n}\}$ dans $\{1,\dots,n\}$. Il y en a $\frac{n!}{(n-\alpha\sqrt{n})!}$ pour un total de $n^{\alpha\sqrt{n}}$ fonctions. La limite suivante s'obtient alors par la formule de Stirling.

$$\frac{n!}{(n-\alpha\sqrt{n})!}\cdot\frac{1}{n^{\alpha\sqrt{n}}} \longrightarrow e^{-\alpha^2/2}$$

□

Exemple (dit *paradoxe des anniversaires*). *Les années comptent 365 jours. Dans une classe de 23 élèves, la probabilité que deux aient le même anniversaire est supérieure à un demi.*

Pour trouver des facteurs non triviaux d'un entier n, on peut exploiter ce paradoxe en cherchant des couples d'entiers $(\alpha,\beta) \in \{1,\dots,n\}^2$ pour lesquels $\operatorname{pgcd}(\alpha-\beta, n) \neq 1, n$. Par

analogie avec l'exemple ci-dessus, les élèves sont des entiers modulo n et leurs anniversaires leurs résidus modulo les facteurs de n. Lorsque n n'est pas premier, son plus petit facteur f vérifie $f \leqslant \sqrt{n}$. En considérant $n^{1/4}$ entiers, la probabilité que deux d'entre eux soient égaux modulo f est donc de 40%. C'est la méthode de factorisation de Shanks [5].

```
def facteur(n):
    L = []
    while True:
        x = randint(1, n)
        for l in L:
            g = gcd(x - l, n)
            if g > 1:
                return g
        L.append(x)
```

Exercice. *Déterminer la complexité en temps et en mémoire de cet algorithme.*

Afin d'améliorer cette complexité tant en temps qu'en mémoire, la méthode de Pollard [7] consiste à chercher les couples (α, β) parmi les valeurs d'une fonction pseudo-aléatoire $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ qui préserve les résidus ; on prendra typiquement pour f une fonction polynômiale de petit degré. Pour chercher ces couples efficacement, on exploite l'observation suivante.

Lemme. *Soit f une fonction préservant les résidus modulo p. Si $f^a(1) = f^b(1) \bmod p$ et $a \geqslant 2b$, alors $f^{2(a-b)}(1) = f^{(a-b)}(1) \bmod p$.*

Démonstration. Composer par $f^{(a-2b)}$. □

On peut donc se contenter de tester les couples $\left(\alpha = f^k(1), \beta = f^{2k}(1)\right)$ pour $k \in \mathbb{N}$.

```
def collision(f, n, m):
    k = 0
    x = f(0)
    y = f(f(0))
    while gcd(x - y, n) == 1 and k < m:
        x = f(x)
        y = f(f(x))
        k = k + 1
    return k

def gcdij(f, n, k, m):
    x = 0
    y = 0
    for i in range(k):
        x = f(x)
    while gcd(x - y, n) == 1 and i < m:
        x = f(x)
        y = f(y)
        i = i + 1
    return gcd(x - y, n)

def facteur(n):
    if n % 2 == 0:
        return 2
    m = floor(3 * n ** 0.25)
```

```
    c = 0
    while True:
        c = c + 1
        f = lambda x: (x * x + c) % n
        k = collision(f, n, m)
        if k < m:
            g = gcdij(f, n, k, m)
            if g > 1 and g < n:
                return g
```

Proposition. *Soit n un entier composé. On suppose que chaque instance de la fonction f se comporte comme un ensemble de tirages aléatoires préservant les résidus. Le programme ci-dessus trouve un facteur non trivial de n en temps $O(k\,n^{1/4})$ avec probabilité $1 - e^{-9k/2}$.*

Les algorithmes ci-dessus sont très efficaces pour trouver des facteurs premiers de petite taille. Asymptotiquement, toutefois, leurs complexités sont exponentielles en la taille des paramètres, à savoir le nombre de bits de l'entier n à factoriser. L'idée majeure menant aux algorithmes sous exponentiels est d'exploiter le théorème ci-dessous [9].

Théorème (Canfield–Erdős–Pomerance). *Quelque soit $c > 0$, lorsque $n \to \infty$, le nombre d'entiers de $\{1, \dots, n\}$ n'admettant aucun facteur premier plus grand que $L(n)^c$ équivaut à*

$$\frac{n}{L(n)^{\frac{1}{2c}+o(1)}} \qquad \textit{avec} \qquad L(x) = \exp\sqrt{\log(x)\log(\log(x))}.$$

L'algorithme de Kraitchik [2] construit un couple (α, β) d'entiers vérifiant $\alpha^2 = \beta^2 \bmod n$ en combinant plusieurs relations plus faciles à obtenir puis en éliminant leurs facteurs non carrés. Il obtient alors une décomposition $n \mid (\alpha - \beta)(\alpha + \beta)$ qui, si n est composée, est triviale pour au plus la moitié des couples (α, β).

Le théorème ci-dessus est utilisé pour construire ces relations : soit P l'ensemble des nombres premiers inférieurs à $L(n)$; pour des entiers x uniformément distribués dans $\{1, \dots, n\}$, décomposer $x^2 \bmod n$ comme produit d'éléments de P nécessite essentiellement $L(n)$ opérations et, d'après le théorème, réussit une fois toutes les $L(n)^{\frac{1}{2}+o(1)}$ en moyenne.

```
def relation(y, P):
    C = []
    if y == 0:
        return []
    for p in P:
        c = 0
        while y % p == 0:
            c += 1
            y //= p
        C.append(c)
    if y != 1:
        return []
    return C
```

Il nous faut alors de relation du type $x^2 = \prod_{p \in P} p^c$ que d'éléments de P sinon plus.

```
def relations(n, P):
    R = []
    while len(R) < 2 * len(P):
        x = randrange(ceil(sqrt(n)), n)
```

```
        r = relation(x * x % n, P)
        if r == []:
            continue
        R.append([x, r])
    return R
```

Une fois suffisamment de relations obtenues, on élimine les facteurs non carrés, c'est-à-dire les exposants c impairs, par des techniques d'algèbre linéaire.

```
def kraitchik(n):
    P = prime_range(2, floor(exp(sqrt(log(n)))))
    R = relations(n, P)
    M = matrix(Integers(2), [r[1] for r in R])
    for e in basis(kernel(M)):
        x = Integers(n)(1)
        y = Integers(n)(1)
        for i in range(len(e)):
            if e[i] == 1:
                x *= R[i][0]
        for i in range(len(P)):
            k = sum([R[j][1][i] for j in range(len(e)) if e[j] == 1])
            y *= power_mod(P[i], k // 2, n)
        z = gcd(x - y, n) % n
        if z > 1:
            return z
```

Chapitre 4

Arithmétique des polynômes

Les entiers relatifs maîtrisés, tournons nous vers des structures plus complexes comme les anneaux de polynômes sur ceux-ci. L'arithmétique des polynômes admet de fortes similarités avec celle des entiers multiprécision, mais se trouve largement facilité par l'absence de propagation de retenue.

4.1 Représentation

On se repose dorénavant sur les fonctionnalités de Sage concernant les nombres entiers, c'est-à-dire qu'on supposera qu'il implante fidèlement la structure d'anneau ordonné de $\mathbb{Z}$ sans se préoccuper des techniques sous-jacentes. Nous n'oublierons pas pour autant le contenu du chapitre précédent et notamment la complexité en temps des opérations arithmétiques sur des entiers arbitrairement grands.

Construisons à présent des objets plus avancés en nous appuyant sur cette base. En guide d'apéritif on se propose de commencer par les nombres rationnels.

Exercice. *Supposant acquise une implantation fidèle de $\mathbb{Z}$ et de son arithmétique, décrire comment représenter les nombres rationnels et implanter les opérations du corps ordonné $\mathbb{Q}$, à savoir la comparaison, la somme, le produit et l'inverse. Il sera opportun d'y rajouter une fonction simplifiant la représentation donnée qui pourra notamment servir préalablement à tout test d'égalité.*

En représentant le rationnel $\frac{p}{q}$ avec $p \in \mathbb{Z}$ et $q \in \mathbb{N}^\star$ comme le couple (p, q), on obtient :

```
def simplifie(x):
    p, q = x
    d = gcd(p, q)
    return (p // d, q // d)

def somme(x, y):
    p, q = x
    r, s = y
    return (p * s + r * q, q * s)

def inferieur(x, y):
    p, q = x
    r, s = y
    return p * s <= r * q
```

On représentera naturellement un polynôme de $\mathbb{Z}[x]$ comme une liste par l'identification :

$$c_0 + c_1 x + c_2 x^2 + \cdots + c_d x^d \qquad \longmapsto \qquad [c_0, c_1, c_2, \ldots, c_d]$$

On écrira donc ainsi :

```
def degre(P):
    d = len(P) - 1
    while P[d] == 0 and d > 0:
        d -= 1
    return d
```

Exercice. *Implanter les opérations somme, produit et division euclidienne. (On supposera le diviseur unitaire de sorte que cette opération soit cantonnée à $\mathbb{Z}[x]$.)*

```
def somme(P, Q):
    m = len(P)
    n = len(Q)
    R = [0] * max(m, n)
    for k in range(m):
        R[k] += P[k]
    for k in range(n):
        R[k] += Q[k]
    return R

def produit(P, Q):
    m = len(P)
    n = len(Q)
    R = [0] * (m + n - 1)
    for i in range(m):
        for j in range(n):
            R[i + j] += P[i] * Q[j]
    return R

def division(P, D):
    m = degre(P)
    n = degre(D)
    Q = [0] * (m - n + 1)
    R = copy(P)
    while m >= n:
        c = R[m] / D[d]
        d = m - d
        Q[d] = c
        M = [0] * d + [-c]
        R = somme(R, produit(M, D)
        m = degre(R)
    return (Q, R)
```

Exercice. *Discuter des avantages et des inconvénients à enlever systématiquement les zéros en fin de liste dans les résultats des fonctions ci-dessus.*

Exercice. *On envisage deux techniques afin d'implanter l'arithmétique de $\mathbb{Q}[x]$:*

1. *Définir $\mathbb{Q}$ et ses opérations comme cela a été fait plus haut puis réimplanter les opérations polynômiales au dessus de $\mathbb{Q}$ plutôt de $\mathbb{Z}$.*

2. *Remarquer que tout élément de* $\mathbb{Q}[x]$ *s'écrit* $\frac{1}{q}P(x)$ *avec* $q \in \mathbb{N}^\star$ *et* $P(x) \in \mathbb{Z}[x]$*; il s'agit alors d'étendre aux couples* $(q, P(x))$ *les opérations de* $\mathbb{Z}[x]$ *ci-dessus.*

Discuter des avantages et des inconvénients de chaque approche.

Nous avons vu en détail l'empilement $\mathbb{Z}/2^{64}\mathbb{Z} \to \mathbb{Z} \to \mathbb{Q} \to \mathbb{Q}[x]$ et toute construction algébrique aurait pareillement vocation à être réalisée comme superposition modulaire d'étapes élémentaires. D'un point de vue pratique, cela crée inexorablement une certaine lourdeur linguistique. Nous allons de nouveau la balayer en adoptant désormais comme polynômes les objets et fonctions fournis directement par Sage. Avant toute chose, il faudra définir l'anneau de polynôme sous-jacent, ce qui est réflexion faite très pertinent car les propriétés de $\mathbb{R}[x]$ et $\mathbb{Z}[x]$ sont bien différentes, notamment en ce qui concerne l'irréductibilité. On écrira :

```
QQx.<x> = PolynomialRing(RationalField())
```

La variable « `x` » représente ainsi un générateur de l'anneau $\mathbb{Q}[x]$. On peut alors définir et manipuler des polynômes en cette variable. Sage les stocke comme une liste de coefficients à laquelle la fonction `list()` permet d'accéder.

```
sage: P = x ** 2 - 1
sage: P.list()
[-1, 0, 1]
```

Commençons par écrire quelques programmes élémentaires pour manipuler ces fonctions.

Exercice. *Écrire un programme qui évalue un polynôme donné en un nombre rationnel donné. En déterminer la complexité. Implémenter à présent la méthode de Horner qui, pour évaluer* $a_n x^n + a_{n-1}x^{n-1} + \cdots a_0$ *en* α*, consiste à calculer* a_n *puis* $a_n\alpha + a_{n-1}$ *puis* $(a_n\alpha + a_{n-1})\alpha + a_{n-2}$*, etc. Quelle est sa complexité ?*

Exercice. *Écrire un programme qui calcule le PGCD de deux polynômes. L'étendre alors afin de calculer aussi les coefficients de Bézout.*

À l'avenir, plutôt que de recourir à vos propres programmes, vous gagnerez à utiliser les fonctions natives équivalentes ci-dessous proposées par Sage et dont l'usage est parfaitement documenté dans l'aide intégrée.

```
sage: P = x^3 + x + 1
sage: P.subs(x=2)
11
sage: P.derivative()
3*x^2 + 1
sage: P.integral()
1/4*x^4 + 1/2*x^2 + x
sage: P.quo_rem(x^2 - 2*x + 1)
(x + 2, 4*x - 1)
```

4.2 Facteurs carrés

Nous pouvons déjà combiner les opérations vues ci-dessus pour résoudre le problème de la factorisation en produit de polynômes sans facteurs carrés, une première étape vers le problème de la factorisation en produit de polynômes irreductibles ainsi que vers le problème de recherche des racines.

Théorème. *Tout polynôme non constant P sur un corps de caractéristique zéro admet la décomposition :*

$$P = \mathrm{gcd}(P, P')\frac{P}{\mathrm{gcd}(P, P')}$$

où le second facteur est un polynôme sans facteur carré.

La preuve s'obtient aisément.

Exercice. *Écrire un programme prenant en argument un polynôme P et renvoyant une liste de polynômes sans facteurs carrés dont le produit vaut exactement P.*

4.3 Racines réelles

Afin de déterminer les racines réelles d'un polynôme à une précision donnée, nous allons nous appuyer sur la théorie de Sturm [1]. Son résultat principal est le suivant.

Théorème. *Soit $P \in \mathbb{R}[x]$ un polynôme dont les racines réelles sont simples. posons*

$$P_0 = P \qquad P_1 = P' \qquad P_{k+1} = -\mathrm{reste}(P_{k-1}, P_k)$$

et notons ω la fonction qui a un nombre $x \in \mathbb{R}$ associe le nombre de changements de signe de la suite $P_k(x)$. Alors, sur tout intervalle $]\alpha, \beta]$, le polynôme P admet exactement $\omega(\alpha) - \omega(\beta)$ racines.

Démonstration. La quantité ω est constante sur tout intervalle ne contenant des racines d'aucun P_k. Si α est une racine de P_k avec $k \geqslant 1$, alors $P_{k-1}(\alpha)$ et $P_{k+1}(\alpha)$ sont de signes distincts puisque $P_{k-1} = P_k Q - P_{k+1}$ et non nuls car $\mathrm{pgcd}(P_k, P_{k+1}) = \mathrm{pgcd}(P_{k-1}, P_k) = \mathrm{pgcd}(P, P')$ n'admet aucune racine réelle ; ainsi, ω est constante sur un voisinage de α. Si α est une racine de P_0, alors P_0 et $P_1 = P_0'$ sont de signes distincts à gauche de α et de même signe à droite de α, donc ω est décrémentée en franchissant α. □

Exercice. *Écrire un programme qui calcule le nombre de racines d'un polynôme P supposé sans facteur carré sur un intervalle $[\alpha, \beta[$ donné.*

```
def suite(P):
    S = [P, P.derivative()]
    while S[-1] != 0:
        Q, R = S[-2].quo_rem(S[-1])
        S.append(-R)
    return S

def omega(S, a):
    c = 0
    v = S[0].subs(a)
    for P in S:
        w = P.subs(a)
        if v * w < 0:
            c = c + 1
        if w != 0:
            v = w
    return c
```

Des approximations numériques des racines de P peuvent dont être obtenues par dichotomie, c'est-à-dire en découpant récursivement l'intervalle de recherche $[\alpha, \beta[$ en $[\alpha, \frac{\alpha+\beta}{2}[\cup[\frac{\alpha+\beta}{2}, \beta[$ et en ne considérant que ceux de ces deux sous intervalles qui contiennent des racines.

Exercice. *Écrire un programme qui détermine des approximations à* $0,01$ *près de toutes les racines d'un polynôme* P *supposé sans facteur carré sur un intervalle* $[\alpha, \beta[$ *donné.*

```
def dichotomie(S, a, b, p):
    if omega(S, a) == omega(S, b):
        return []
    elif a - b < p:
        return [a]
    else:
        c = (a + b) / 2
        return dichotomie(S, a, c, p) + dichotomie(S, c, b, p)

def racines(P):
    if P.degree() == 0:
        return []
    a = RealField(3000)(10 ^ 60)
    G = gcd(P, P.derivative())
    S = suite(P // G)
    return dichotomie(S, -a, a, 1 / a) + racines(G)
```

La fonction intégrée à Sage s'utilise comme il suit.

```
sage: P = x^3 + x + 1
sage: P.roots()
[]
sage: P.roots(ring=RR)
[(-0.682327803828019, 1)]
sage: P.roots(ring=CC)
[(-0.682327803828019, 1),
 (0.341163901914010 - 1.16154139999725*I, 1),
 (0.341163901914010 + 1.16154139999725*I, 1)]
```

4.4 Factorisation sur les corps finis

Soit $P \in \mathbb{F}_q[x]$ un polynôme à factoriser. En Sage, prenons par exemple :

```
FF.<x> = PolynomialRing(FiniteField(7))
P = FF.random_element(degree=17)
```

Plus haut, nous avons exploité l'égalité $P = \mathrm{pgcd}(P, P') \cdot \frac{P}{\mathrm{pgcd}(P,P')}$ de manière itérée pour nous ramener au cas de polynômes sans facteur carré. Sur un corps de caractéristique $p > 0$, un polynôme non constant peut toutefois être de dérivée nulle.

Lemme. *Un polynôme* $P \in \mathbb{F}_q[x]$ *vérifie* $P' = 0$ *si et seulement s'il existe* $Q \in \mathbb{F}_q[x]$ *tel que* $P = Q(x^p)$*, où* p *dénote la caractéristique de* $\mathbb{F}_q$*.*

```
def lemmeQ(P):
    c = P.list()
    p = P.parent().base_ring().characteristic()
    return sum([c[p * i] * x ** i for i in range(len(c) // p)])
```

On peut alors utiliser la même méthode que précédemment pour décomposer un polynôme P donné en produit de polynômes sans facteur carré, quitte à se ramener à Q lorsque P' s'annule.

```
def factorisation(P):
    if P.degree() == 0:
        return []
    if P.degree() == 1:
        return [P]
    p = P.parent().base_ring().characteristic()
    D = P.derivative()
    if D == 0:
        Q = lemmeQ(P)
        F = factorisation(Q)
        return [f.subs(x=x ** p) for f in F]
    G = gcd(P, D)
    return factorisation(G) + factorisation_simple(P // G)
```

Afin de décomposer plus avant le polynôme donné, on utilise la théorie des anneaux.

Proposition. *Pour tout $k > 0$ le polynôme $x^{q^k} - x$ est le produit de tous les polynômes unitaires irréductibles dont le degré divise k dans $\mathbb{F}_q[x]$.*

Démonstration. Soit P un polynôme irréductible de $\mathbb{F}_q[x]$ de degré divisant par k. Toutes ses racines sont simples et définies dans l'extension $\mathbb{F}_{q^k}$; elles sont donc fixées par l'endomorphisme de Frobenius de ce corps, à savoir $x \mapsto x^{q^k}$, et P divise donc $x^{q^k} - x$.

Inversement, si un facteur irréductible de $x^{q^k} - x$ n'était pas de degré divisant k alors ses racines ne seraient pas définies dans $\mathbb{F}_{q^k}$ et donc pas fixées par $x \mapsto x^{q^k}$. □

Ce résultat permet de décomposer un polynôme en produit de polynômes dont les facteurs irréductibles sont de même degrés.

```
def factorisation_simple(P):
    q = P.parent().base_ring().cardinality()
    F = []
    k = 1
    while P.degree() > 0:
        xx = power_mod(x, q ** k, P) - x
        G = gcd(P, xx)
        if G.degree() > 0:
            F += factorisation_degre(G, k)
            P = P // G
        k += 1
    return F
```

Nous sommes donc ramené au cas de polynômes $P \in \mathbb{F}_q[x]$ dont tous les facteurs irréductibles P_i sont distincts et de même degré k, c'est-à-dire qu'on a

$$\mathbb{F}_q[x]/(P) \simeq \prod \mathbb{F}_q[x]/(P_i)$$

où les facteurs du membre de droite sont tous des corps à q^k éléments ; supposant la caractéristique impair, leurs groupes multiplicatifs sont donc cycliques à $q^k - 1$ éléments. Dans chacun de ces groupes, le morphisme $x \mapsto x^{\frac{q^k-1}{2}}$ admet un noyau de cardinal $\frac{q^k-1}{2}$. Si h est un élément uniformément distribué dans $\mathbb{F}_q[x]/(P)$, le polynôme $h^{\frac{q^k-1}{2}} - 1$ s'annule donc avec probabilité $1/2$ dans chacun des facteurs de manière indépendante.

```
def factorisation_degre(P, k):
    A = P.parent()
    q = A.base_ring().cardinality()
    if P.degree() == 0:
        return []
    if P.degree() == k:
        return [P]
    h = A.random_element(degree=P.degree() - 1)
    H = power_mod(h, (q ** k - 1) // 2, P) - 1
    G = gcd(P, H)
    return factorisation_degre(G, k) + factorisation_degre(P // G, k)
```

Rationnels via Hensel ? LLL ?

Chapitre 5

Calcul numérique

Les ordinateurs manipulent des grandeurs purement discrètes et sont à ce titre profondément inaptes aux mathématiques continues. On peut cependant leur en inculquer quelques rudiments en contournant cette limitation par l'une des deux approches que voici :

1. **Calcul formel.** En se restreignant à un sous-ensemble discret que l'on sait manipuler des grandeurs continues considérées, par exemple les polynômes ou encore les expressions algébriques en un nombre fini de fonctions usuelles. L'enjeu est alors le développement de méthodes applicables aux sous-ensembles les plus vastes possibles ; penser notamment aux classes de fonctions que vous savez intégrer.
2. **Calcul numérique.** En assimilant une grandeur continue à la valeur discrète la plus proche. La difficulté est alors de majorer rigoureusement les termes d'erreurs resultants de ces approximations.

Ce chapitre présente quelques outils élémentaires du calcul numérique.

5.1 Notions de fonctions

Sage offre deux notions de fonctions bien distinctes.

Les expressions algébriques. C'est la notion qui s'apparente le plus à votre vision des fonctions mathématiques. Chaque fonction y est représentée par l'arborescence décrivant le résultant comme combinaison de fonctions usuelles et de variables abstraites.

```
sage: var('x,y')
sage: f(x, y) = x ** y - x - 2
sage: solve(f(x, 3) == 0, x)
[x == -1/2*(1/9*sqrt(26)*sqrt(3) + 1)^(1/3)*(I*sqrt(3) + 1) -
       1/6*(-I*sqrt(3) + 1)/(1/9*sqrt(26)*sqrt(3) + 1)^(1/3),
 x == -1/2*(1/9*sqrt(26)*sqrt(3) + 1)^(1/3)*(-I*sqrt(3) + 1) -
       1/6*(I*sqrt(3) + 1)/(1/9*sqrt(26)*sqrt(3) + 1)^(1/3),
 x == (1/9*sqrt(26)*sqrt(3) + 1)^(1/3) +
       1/3/(1/9*sqrt(26)*sqrt(3) + 1)^(1/3)]
sage: solve(f(x, 7) == 0, x)
[0 == x^7 - x - 2]
```

Cette notion est adaptée au calcul formel et on en voit justement les limites : seules les équations polynômiales de degré inférieur ou égal à cinq sont résolubles par radicaux en général. Pour manipuler la racine de $f(x, 7)$, il faut développer d'autres techniques.

Les procédures informatiques. Cette notion correspond à celle de programme que nous avons utilisée indifféremment jusqu'à présent.

```
sage: def f(x, y):
....:     return x ** y - x - 2
....:
sage: find_root(lambda x: f(x, 3), 0, 2)
1.5213797068045676
sage: find_root(lambda x: f(x, 7), 0, 2)
1.1796938907415153
```

Cette notion est adaptée au calcul numérique que nous allons maintenant développer.

5.2 Limites

On sait parfaitement déterminer le comportement des suites vérifiant une récurrence de la forme $x_{n+1} = f(x_n)$ lorsque f est une fonction affine. C'est toutefois un problème entier dans le cas général et un système de calcul formel ne peut espérer qu'y répondre de manière approchée grâce au calcul numérique. Prenons l'exemple phare que voici.

Définition. *La suite logistique de paramètre $\lambda \in [0,4]$ est définie par :*

$$\begin{cases} x_0 = 1/2, \\ x_{n+1} = \lambda x_n (1 - x_n). \end{cases}$$

Exercice. *Conjecturer son comportement asymptotique pour $\lambda = 8/3$. De même pour $\lambda = 10/3$.*

On pourra plus généralement considérer qu'un réel ℓ est limite d'une suite $(x_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier n pour lequel $\{x_{n+1}, x_{n+2}, \ldots, x_{n+N}\} \subset [\ell - \varepsilon, \ell + \varepsilon]$ avec par exemple $N = 10$, $\varepsilon = 10^{-4}$ et $n \leqslant 10^4$; cela peut être pertinent en sciences physiques si ces quantités peuvent être considérées comme négligeables. Lorsqu'on mettra en œuvre cette approche il faudra néanmoins garder à l'esprit son manque de rigueur mathématique : la condition précédente n'est qu'une approximation et peut être vérifiée même quand (x_n) diverge grossièrement.

```
def valeurs(l, n, N):
    L = []
    x = 0.5
    for k in range(n + N):
        x = l * x * (1 - x)
        if k >= n:
            L.append(x)
    return L

valeurs(8 / 3, 100, 10)
valeurs(10 / 3, 100, 10)

def limite(l):
    n = 10 ** 4
    N = 10
    e = 10 ** (-4)
    L = valeurs(l, n, N)
    if max(L) - min(L) < e:
        return L[N - 1]
    else:
        return None
```

Notre programme identifie comme prévu une limite pour la suite logistique de paramètre $\lambda = 8/3$ mais pas $\lambda = 10/3$. Pour traiter avec davantage de finesse ce second cas, rappelons la notion de valeur d'adhérence qui généralise celle de limite.

Définition. *On dit que $\ell \in \mathbb{R}$ est une valeur d'adhérence de la suite $x \in \mathbb{R}^{\mathbb{N}}$ lorsqu'on a*

$$\forall \varepsilon > 0, \forall N \in \mathbb{N}, \exists n \geqslant N, |x_n - \ell| < \varepsilon.$$

Proposition. *Les valeurs d'adhérence ne sont autres que les limites des sous suites.*

Exercice. *Écrire une fonction prenant en argument un réel λ et renvoyant (une approximation de) l'ensemble des valeurs d'adhérences de la suite logistique.*

```
def adherences(l):
    n = 10 ** 4
    L = valeurs(l, n, n)
    L.sort()
    r = 0
    R = []
    for k in range(n // 10):
        if abs(L[10 * k] - r) > 0.001:
            r = L[10 * k]
        elif not r in R:
            R.append(r)
    return R
```

Voir la figure 5.1 pour un tracé de cet ensemble en fonction du paramètre λ ; pour le réaliser en Sage on procède comme il suit en constituant simplement une liste de points à afficher.

```
P = []
for i in range(10 ** 3):
    l = 2.5 + 1.5 * i / 10 ** 3
    P += [(l, y) for y in adherences(l)]

show(points(P, pointsize=1))
```

5.3 Zéros

L'objectif de cette section est de résoudre numériquement une équation du type $f(x) = 0$ où f dénote une fonction réelle d'une variable réelle que l'on sait évaluer. On cherche ainsi un nombre réel α vérifiant $0 \in f(]\alpha - \varepsilon, \alpha + \varepsilon[)$ où la quantité $\varepsilon > 0$ dénote l'erreur tolérée.

Méthode de la dichotomie

La première méthode que nous allons décrire s'appuie sur le théorème des valeurs intermédiaires et fonctionne donc dès que la fonction f est continue. Son principe est identique à celui que nous avons vu dans le cadre de la recherche d'un élément dans une liste triée : diviser par deux l'espace de recherche à chaque itération.

Afin de chercher un zéro de f sur l'intervalle $[a, b]$ à ε près on procède alors ainsi :

```
def dichotomie(f, a, b, epsilon):
    fa = f(a)
    fb = f(b)
```

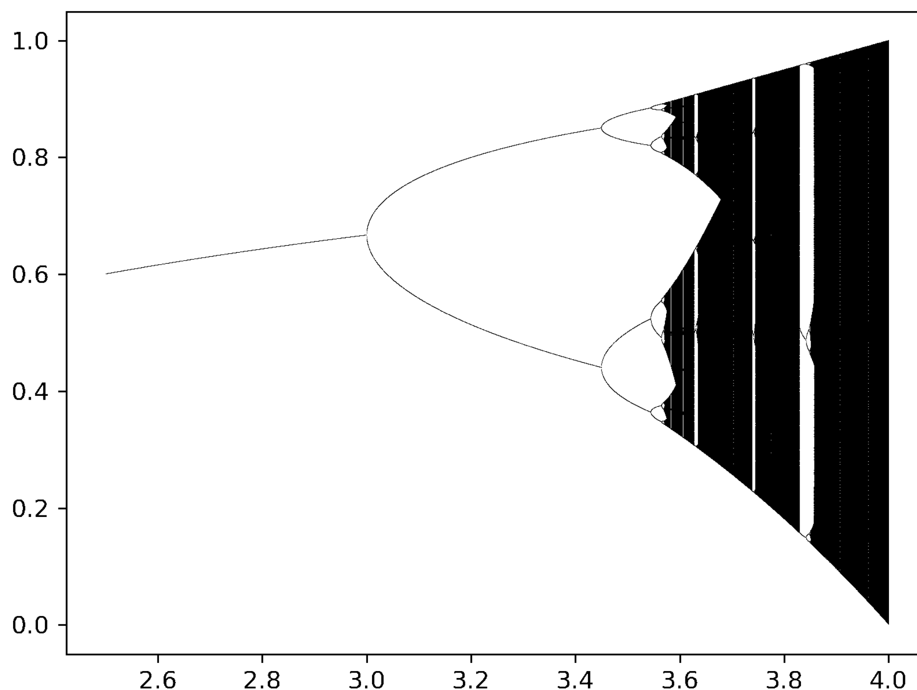

FIGURE 5.1 – Valeurs d'adhérence de la suite logistique pour $\lambda \in [5/2, 4]$.

```
    while b - a > epsilon:
        c = (a + b) / 2
        fc = f(c)
        if fa * fc < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc
    return a
```

Exercice. *Utiliser cette méthode pour calculer des valeurs approchées de $\sqrt{2}$ et de π.*

Exercice. *Combien de temps cela vous prendrait-il pour calculer $\sqrt{2}$ à mille chiffres significatifs avec cette méthode ?*

Chaque itération divise par deux la longueur de l'intervalle contenant la racine cherchée et augmente ainsi la précision d'un bit significatif. Il faut donc $\log\left(\frac{b-a}{\varepsilon}\right)$ itérations afin d'obtenir le résultat désiré.

Remarque. *Par défaut les réels utilisés par Sage sont de type* `binary64`*; ce type est manipulé directement par le processeur et donc stocké sur 64 bits, ce qui limite à seize le nombre de chiffres significatifs. Afin de calculer mille chiffres significatifs, il faut appliquer au calcul numérique les mêmes techniques de calcul multiprécision que nous avons mises en œuvre pour le calcul formel. En Sage, on peut faire :*

```
R = RealField(4000)
dichotomie(lambda x: x ^ 2 - 2, R(0), R(2), R(10 ** -1000))
```

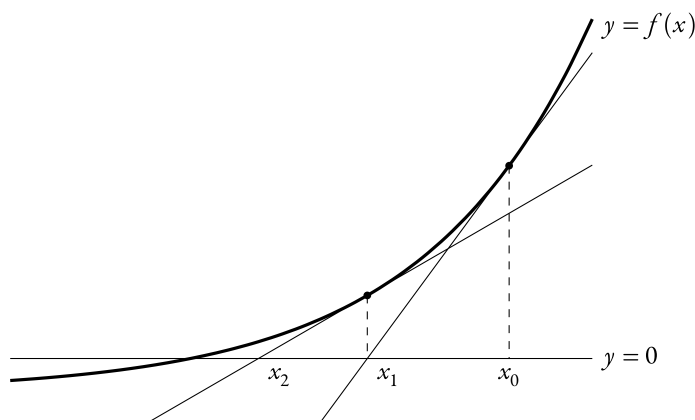

FIGURE 5.2 – Deux itérations de la méthode de Newton.

Méthode de Newton

Cette seconde méthode exploite la formule de Taylor à l'ordre un et fonctionne dès que la fonction f est de classe $\mathscr{C}^2$. Son principe est d'approcher f par ses tangentes et donc d'approcher le zéro recherché par les zéros de ces droites. Voir la figure 5.2.

Définition. *Soit f une fonction réelle de classe $\mathscr{C}^2$. On appelle suite de Newton pour cette fonction toute suite vérifiant la relation de récurrence $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.*

Théorème. *Soit α un zéro d'une fonction f de classe $\mathscr{C}^2$ pour lequel $f'(\alpha) \neq 0$. Il existe un voisinage V de α tel que toute suite de Newton à valeur initiale dans V converge quadratiquement vers α.*

Démonstration. Le développement de Taylor de f en α donne l'existence d'une suite y telle que

$$0 = f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2} f''(y_n)(\alpha - x_n)^2;$$

divisant par $f'(x_n)$ on obtient

$$\frac{f(x_n)}{f'(x_n)} + (\alpha - x_n) = -\frac{1}{2}\frac{f''(y_n)}{f'(x_n)}(\alpha - x_n)^2$$

c'est-à-dire

$$\alpha - x_{n+1} = -\frac{1}{2}\frac{f''(y_n)}{f'(x_n)}(\alpha - x_n)^2$$

or, par continuité, f'' et $1/f'$ sont bornés au voisinage de α. □

Afin de chercher un zéro de f (de dérivée g) au voisinage de a en n itérations on procède alors ainsi :

```
def newton(f, g, a, n):
    for i in range(n):
        a = a - f(a) / g(a)
    return a
```

La preuve ci-dessus montre que, lorsque a est suffisamment proche du zéro recherché α, la précision en nombre de bits significatifs double à chaque itération. Il faut donc $c+\log\left(\log\left(\frac{1}{\varepsilon}\right)\right)$ itérations pour obtenir le résultat désiré, où la constante c ne dépend que de f et de α.

Remarque. *Les suites de Newton sont utiles même lorsque les hypothèses du théorème ci-dessus ne sont pas vérifiées. Par exemple, si la dérivée s'annule, alors la converge est toujours possible, même si pas nécessairement quadratique.*

Exercice. *Montrer que la suite définie par $x_0 = 1$ et $x_{n+1} = \cos(x_n)$ converge. Calculer une approximation de sa limite. Comparer les vitesses de convergence de la dichotomie et de Newton.*

Lorsque la dérivée g de f n'est pas efficacement évaluable on peut l'approcher en posant $g(a) = \frac{1}{\varepsilon}(f(a+\varepsilon) - f(a))$. On pourra en exercice adapter le théorème et sa preuve à ce cas.

Exercice. *Pour $s \in]1,\infty[$ on pose $\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$. Calculer une approximation de l'unique préimage de 2 par ζ. Comparer les vitesses de convergence de la dichotomie et de Newton.*

5.4 Intégration

On considère ici le problème consistant à calculer une valeur approchée de l'intégrale d'une fonction réelle f donnée sur un segment $[a,b]$ donné. Une première approche, populairement baptisée *méthode des rectangles*, repose sur le théorème suivant vu en cours d'analyse.

Théorème (sommes de Riemann). *Soit f une fonction de classe $\mathscr{C}^1([a,b],\mathbb{R})$. On a*

$$\left|\varepsilon \sum_{k=0}^{\left\lfloor \frac{b-a}{\varepsilon} \right\rfloor} f(a+k\varepsilon) - \int_a^b f(x)dx\right| \leqslant \varepsilon\left(\frac{b-a}{2}\|f'\|_\infty + \|f\|_\infty\right)$$

c'est-à-dire que la somme tend vers l'intégrale avec un terme d'erreur en $O(\varepsilon)$.

Démonstration. Pour tout $\alpha \in [a,b]$ on a

$$\begin{aligned}\left|\varepsilon f(\alpha) - \int_\alpha^{\alpha+\varepsilon} f(x)dx\right| &= \left|\int_\alpha^{\alpha+\varepsilon} (f(\alpha) - f(x))\,dx\right| \\ &\leqslant \int_\alpha^{\alpha+\varepsilon} \|f'\|_\infty\,|\alpha - x|\,dx \\ &\leqslant \|f'\|_\infty \frac{\varepsilon^2}{2}\end{aligned}$$

et il ne reste qu'à sommer ces majorations pour $\alpha = a + k\varepsilon$ lorsque k parcours $\{0,\ldots,n\}$ avec $n = \left\lfloor \frac{b-a}{\varepsilon} \right\rfloor$ puis d'y ajouter le terme en $\|f\|_\infty$ qui borne l'intégrale restante $\int_{a+n\varepsilon}^b f(x)dx$. □

Remarque. *Si f est seulement supposée continue alors la somme de Riemann converge toujours vers l'intégrale (en utilisant l'uniforme continuité plutôt que les accroissements finis) mais on ne peut pas exprimer facilement le terme d'erreur; ce cas est donc d'intérêt moindre en calcul numérique.*

Exercice. *Écrire une fonction `integrale(f,a,b)` calculant une approximation de l'intégrale de la fonction réelle f sur le segment $[a,b]$ en utilisant la méthode des rectangles. Déduire une valeur approchée de π en utilisant $f(x) = \sqrt{1-x^2}$.*

Pour raffiner cette technique d'intégration numérique on peut approcher f sur $[\alpha, \alpha+\varepsilon]$ non pas par la constance $f(\alpha)$ mais par la fonction affine

$$x \longmapsto f(\alpha) + \frac{x-\alpha}{\varepsilon}\left(f(\alpha+\varepsilon) - f(\alpha)\right);$$

on obtient ainsi la méthode dite *des trapèzes*.

Théorème. *Soit f une fonction de classe $\mathscr{C}^2([a,b],\mathbb{R})$. On a*

$$\left|\varepsilon \sum_{k=0}^{\lfloor \frac{b-a}{\varepsilon} \rfloor} \left(\frac{f(a+k\varepsilon) + f(a+(k+1)\varepsilon)}{2}\right) - \int_a^b f(x)dx\right| = O(\varepsilon^2).$$

Remarque. *La somme ci-dessus est fortuitement identique à celle correspondant à la méthode des rectangles aux termes en $f(\alpha)$ et $f(\alpha+n\varepsilon)$ près. Ces termes à eux seuls améliorent donc la convergence de linéaire en quadratique !*

Si cela peut paraître astucieux, la méthode des trapèzes forme le premier échelon d'une famille de méthodes bien générales consistant à approcher f par des polynômes de degré k fixé. Pour $k=0$ on a la méthode des rectangles, pour $k=1$ celle des trapèzes et pour $k=2$ celle des paraboles communément attribuée à Simpson.

Exercice. *Calculer le volume $V_n(r)$ d'une boule de rayon r en dimension n.*

Il vérifie les relations $V_n(r) = r^n V_n(1)$ et $V_{n+1}(1) = \int_{-1}^1 V_n(\sqrt{1-x^2})dx$.

Quel est le volume des sphères correspondantes ?

Exercice. *Mesurer la superficie de la partie du plan (x,y) définie par $x^2+y^4<1$.*

On peut utiliser la fonction `numerical_integral` comme il suit.

```
sage: f = lambda x: sin(x) / exp(x)
sage: numerical_integral(f, 0, 1)
(0.24583700700023747, 2.7293390547659943e-15)
```

5.5 Équations différentielles

Le principe consistant à approcher une fonction par ses tangentes permet aussi de résoudre numériquement les équations différentielles ordinaires, technique connue sous le nom de méthode d'Euler.

Théorème. *Soit une fonction y vérifiant l'équation différentielle ordinaire $y'(x) = f(x, y(x))$ et la condition initiale $y(x_0) = y_0$. Si f est de classe $\mathscr{C}^1$ alors il existe $\eta > 0$ tel que, pour tout $\varepsilon > 0$, les suites définies par*

$$\begin{cases} x_{n+1} = x_n + \varepsilon \\ y_{n+1} = y_n + \varepsilon f(x_n, y_n) \end{cases}$$

vérifient $|y_n - y(x_n)| < \varepsilon$ pour tout $n \in \left\{1, \ldots, \lfloor \frac{\eta}{\varepsilon} \rfloor\right\}$.

Exercice. *Calculer ainsi une approximation de la fonction exponentielle.*

Exercice. *Calculer des fonctions y solutions de l'équation différentielle $y' = \frac{y}{x(1+y)}$. Qu'observez-vous lorsque $x \to 0$? Et lorsque $y \to -1$? Voir la figure 5.3.*

Cette méthode s'applique aussi aux équations différentielles ordinaires d'ordre supérieur en les écrivant comme des équations différentielles ordinaires du premier ordre vectorielles.

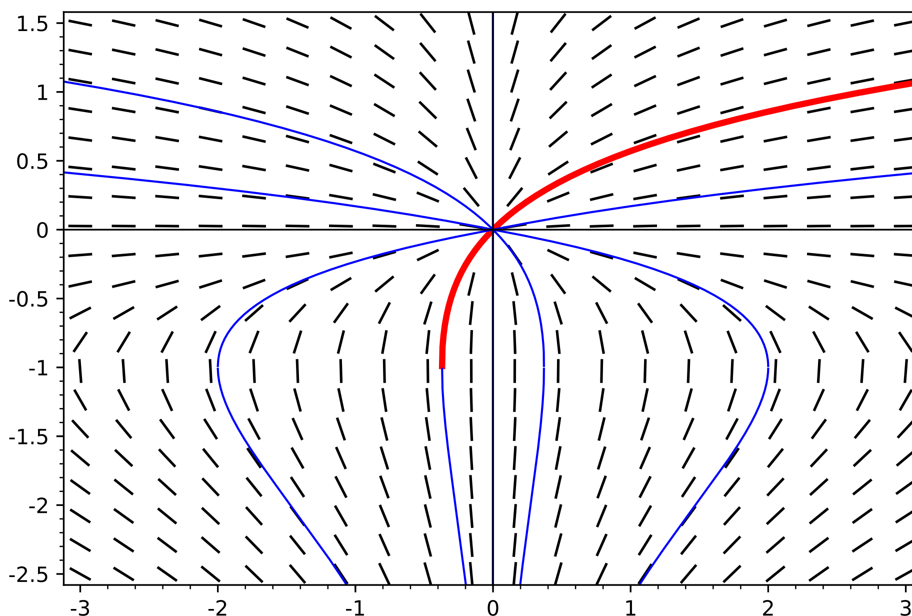

FIGURE 5.3 – Champ de vecteurs et solutions de l'équation différentielle $y' = \frac{y}{x(1+y)}$. En rouge, la fonction W de Lambert.

Exercice. *Calculer ainsi une approximation du sinus.*

Exercice (méthode de Runge–Kutta). *Refaire les exercices précédents en exploitant l'approximation d'ordre supérieur :*

$$y(t+\varepsilon) \approx y(t) + \varepsilon \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \qquad \text{où} \qquad \begin{cases} k_1 = f(t, y(t)) \\ k_2 = f(t + \frac{\varepsilon}{2}, y(t) + \frac{\varepsilon}{2} k_1) \\ k_3 = f(t + \frac{\varepsilon}{2}, y(t) + \frac{\varepsilon}{2} k_2) \\ k_4 = f(t + \varepsilon, y(t) + \varepsilon k_3) \end{cases}$$

Comparer les vitesses de convergence de la méthode d'Euler et de cette nouvelle méthode.

5.6 Tracés et animations

Pour la vulgarisation il est utile d'animer des tracés de fonctions. Nous allons ici exploiter cette technique afin d'illustrer différentes notions vues en licence.

Exemple. *Convergence uniforme :*

```
a = animate(
    [plot([1, 1 + sin(n * x) / n], 0, 1) for n in range(1, 20)],
    xmin=0,
    xmax=1,
    ymin=0,
    ymax=2,
)
a.show()
```

Convergence simple :

```
b = animate(
    [plot([1, 1 + 1 / x / n], 0, 1) for n in range(1, 20)],
    xmin=0,
    xmax=1,
    ymin=0,
    ymax=2,
)
b.show()
```

Exercice. *Tracer la fonction qui à un réel positif t associe l'intégrale qui définit la fonction gamma, entre 0 et N, et animer ce tracé pour des valeurs de N croissantes.*

```
x, t = var("x,t")
c = animate(
    [
        plot([gamma(x), integral_numerical(t ^ (x - 1) * exp(-t), 0, N)], 0.2, 4)
        for N in range(1, 10)
    ],
    xmin=0.2,
    xmax=4,
    ymin=0,
    ymax=6,
)
c.show()
```

Exercice. *On rappelle que la transformée de Fourier d'une fonction rélle intégrable f est*

$$\mathscr{F}(f) : t \longmapsto \int_{-\infty}^{\infty} f(s) e^{-its} ds$$

et que la série de Fourier de $f : [-\pi, \pi] \to \mathbb{R}$ à l'ordre n s'écrit alors

$$t \longmapsto \sum_{k=-n}^{n} \frac{\mathscr{F}(f)(k)}{2\pi} e^{ikt}.$$

Animer le tracé des séries de Fourier de la fonction $f = \mathrm{id}_{[-\pi,\pi]}$ en fonction de l'ordre $n \in \{0, \ldots, 10\}$. Que dire de la convergence aux extrémités de l'intervalle ? Tracer la fonction $\mathscr{F}(\mathscr{F}(f))$.

```
def integral_complex(f, a, b):
    r = integral_numerical(lambda x: real(f(x)), a, b)[0]
    i = integral_numerical(lambda x: imag(f(x)), a, b)[0]
    return r + I * i
```

Exercice. *Trouver un polynôme $P \in \mathbb{R}[x]$ de degré cinq minimisant la quantité $\int_{-1}^{1} |P(x) - |x|| \, dx$.*

Exercice. *Animer la convergence de la série $\sum_{k\in\mathbb{Z}} \frac{1}{(x-k)^2}$ vers la fonction $\frac{\pi^2}{\sin(\pi x)^2}$.*

Démonstration. Le spectre de l'opérateur associant à $h \in \mathscr{C}^0([0,1], \mathbb{R})$ la fonction $x \in [0,1] \mapsto h\left(\frac{x}{2}\right) + h\left(\frac{x+1}{2}\right)$ est inclus dans $[-2, 2]$. L'appliquer alors à la différence des deux fonctions ci-dessus. □

Exercice. *L'ensemble de Cantor est celui des réels de* $[0,1]$ *dont l'écriture en base* 3 *ne contient pas le chiffre* 1*. On peut le définir comme la limite de la suite d'ensembles* C_n *vérifiant* $C_0 = [0,1]$ *et, si* C_n *est l'union disjointe d'intervalles* $[a,b]$*, alors* C_{n+1} *est celle des intervalles* $[a, \frac{2a+b}{3}] \cup [\frac{a+2b}{3}, b]$*. Animer le tracer de la fonction indicatrice de* C_n*, pour* $n \in \{0, 10\}$*.*

L'escalier de Cantor est une fonction réelle continue f *définie sur* $[0,1]$*; pour calculer* $f(x)$*, écrire* x *en base* 3*, remplacer tout les chiffres situés après un* 1 *par des* 0*, puis remplacer tous les* 2 *par des* 1 *et interpréter le résultat en base deux. Notons qu'elle est uniformément continue mais pas absolument continue ; en outre elle est dérivable presque partout mais de dérivée nulle. On peut la définir comme la limite de la suite de fonctions* f_n *: la fonction* f_0 *est l'identité et, pour tout* $n \in \mathbb{N}$*, la fonction* f_{n+1} *est identique à* f_n *sur tout intervalle (non réduit à un point) sur lequel elle est constante. Sur tout intervalle (maximal)* $[a; b]$ *où* f_n *n'est pas constante, alors* f_{n+1} *vaut* $\frac{1}{2}(f(a)+f(b))$ *sur* $[\frac{2a+b}{3}, \frac{a+2b}{3}]$ *et est linéaire et continue sur* $[a, \frac{2a+b}{3}]$ *et* $[\frac{a+2b}{3}, b]$*.*

```
c = [0, 1]
d = [0, 0, 1]
CD = [(c, d)]
for n in range(10):
    (e, f) = ([], [0])
    for i in range(len(c)):
        if d[i + 1] > 0:
            e.append(c[i])
            f.append(d[i + 1])
        else:
            e.append(c[i])
            e.append((2 * c[i] + c[i + 1]) / 3)
            e.append((c[i] + 2 * c[i + 1]) / 3)
            f.append(0)
            f.append((d[i] + d[i + 2]) / 2)
            f.append(0)
    CD.append((e, f))
    (c, d) = (e, f)

def escalier(x, n):
    (c, d) = CD[n]
    i = 0
    while x > c[i + 1]:
        i = i + 1
    if d[i + 1] > 0:
        return d[i + 1]
    return ((x - c[i]) * d[i + 2] + (c[i + 1] - x) * d[i]) / (c[i + 1] - c[i])

plot(lambda x: escalier(x, 4), 0, 1)
E = animate(
    [plot(lambda x: escalier(x, n), 0, 1) for n in range(11)],
    xmin=0,
    xmax=1,
    ymin=0,
    ymax=1,
)
E.show()
```

Chapitre 6

Applications en cryptographie

6.1 Chiffrement

6.2 Signature

6.3 Partage de clefs

6.4 Attaques

Bibliographie

Afin de laisser transparaître la dimension historique du développement des concepts théoriques et de leur mise en œuvre effective abordés dans ce cours, les références ci-dessous sont énumérées dans l'ordre chronologique.

[1] Jacques Charles François STURM.
« Mémoire sur la résolution des équations numériques ».
Bulletin des sciences de Férussac 11 (1829), pages 419-422.

[2] Maurice KRAITCHIK. « Analyse indéterminée du second degré et factorisation ».
Théorie des Nombres. Tome 2. Gauthier-Villars, 1926.

[3] Alan Mathison TURING.
« On computable numbers, with an application to the Entscheidungsproblem ».
Proceedings of the London Mathematical Society 42.2 (1937), pages 230-265.
DOI : 10.1112/plms/s2-42.1.230.

[4] Arnold SCHÖNHAGE et Volker STRASSEN.
« Schnelle Multiplikation großer Zahlen ». *Computing* 7.3-4 (1971), pages 281-292.
DOI : 10.1007/BF02242355.

[5] Daniel SHANKS. « Class number, a theory of factorization, and genera ».
1969 Number Theory Institute. Édité par Donald J. LEWIS. Tome 20.
Proceedings of Symposia in Pure Mathematics. American Mathematical Society, 1971, pages 415-440.

[6] Gary Lee MILLER. « Riemann's hypothesis and tests for primality ».
Symposium on Theory of Computing — STOC 1975. Édité par William C. ROUNDS, Nancy MARTIN, Jack W. CARLYLE et Michael A. HARRISON.
Association for Computing Machinery, 1975, pages 234-239.
DOI : 10.1145/800116.803773.

[7] John M. POLLARD. « A Monte Carlo method for factorization ».
BIT Numerical Mathematics 15.3 (1975), pages 331-334.
DOI : 10.1007/BF01933667.

[8] Michael Oser RABIN. « Probabilistic algorithm for testing primality ».
Journal of Number Theory 12.1 (1980), pages 128-138.
DOI : 10.1016/0022-314X(80)90084-0.

[9] Earl CANFIELD, Paul ERDŐS et Carl POMERANCE.
« On a problem of Oppenheim concerning 'factorisatio numerorum' ».
Journal of Number Theory 17.1 (1983), pages 1-28.
DOI : 10.1016/0022-314X(83)90002-1.

[10] *PARI/GP*.
A computer algebra system designed for fast computations in number theory.
The PARI Group. 1985. URL : http://pari.math.u-bordeaux.fr/.

[11] *R*. A Language and Environment for Statistical Computing.
R Development Core Team. 1993. URL : http://www.r-project.org/.

[12] *Singular*. A computer algebra system for polynomial computations.
University of Kaiserslautern. 1997. URL : http://www.singular.uni-kl.de/.

[13] Manindra AGRAWAL, Neeraj KAYAL et Nitin SAXENA. « PRIMES is in P ».
Annals of Mathematics 160.2 (2004), pages 781-793.
DOI : 10.4007/annals.2004.160.781.

[14] William Arthur STEIN et al. *Sage Documentation*. The Sage Development Team. 2005.
Chapitre Sage Tutorial. URL : http://www.sagemath.org/doc/tutorial/.

[15] William Arthur STEIN et al. *Sage Mathematics Software*.
The Sage Development Team. 2005. URL : http://www.sagemath.org/.

[16] Richard Peirce BRENT et Paul ZIMMERMANN. *Modern Computer Arithmetic*.
Monographs on Applied and Computational Mathematics.
Cambridge University Press, 2010. ISBN : 0-521-19469-5.

[17] David HARVEY et Joris VAN DER HOEVEN. *Integer multiplication in time O(n log n)*.
2019. URL : https://hal.archives-ouvertes.fr/hal-02070778.