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

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 [6] en théorie des nombres, de R [8] en statistiques, ou encore de Singular [11] en algèbre commutative. Le logiciel Sage [13] 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 [12].

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
8
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)]
```

Attention. La version de Python que Sage utilise considère les entiers commençant par un zéro comme écrit en base huit. On a donc :

```
sage: 011
9
```

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.
```
sage: sqrt?
Type:       function
String Form:<function sqrt at 0x492ec80>
File:       /opt/sage/local/lib/python2.7/site-packages/sage/functions/other.py
Definition: 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 place holder, and is
     always ignored or passed to the sqrt function for 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(-1)
      I
      sage: sqrt(2)
      sqrt(2)
      sage: sqrt(2)^2
      2
      sage: sqrt(4)
      2
      sage: sqrt(4,all=True)
      [2, -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: 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: print 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: print n
10
```

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

Exercice. *Un nombre premier est dit de Mersenne s'il est de la forme* $2^n - 1$. *Identifier les autant de tels nombres que possible.*

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 vecteurs

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=[2,3,5,7]
sage: len(y)
4
sage: y[0]
2
sage: y[3]
7
sage: y[4]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-cb46143f779d> in <module>()
----> 1 y[4]

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=[2,3,5,7]
sage: y.append(11)
sage: y[0]=1
sage: y.remove(0)
sage: y.append(11)
sage: y
[1,3,5,7,11]
```

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+[11]
[1,3,5,7,11]
sage: y
[1,3,5,7]
```

Exercice. *Écrire une fonction qui renvoit 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 renvoit 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 Goostein 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.

En mathématiques, bastion de la rigueur, on a besoin d'exactitude inconditionnelle, quel

qu'en soit le prix en temps de calcul. Nous verrons notamment que l'on peut représenter $\mathbb{Z}$ fidèlement en utilisant des listes d'éléments de $\mathbb{Z}/2^{64}\mathbb{Z}$, approche connue sous le nom d'« arithmétique multiprécision ». Des techniques plus générales permettent de représenter l'anneau $\mathbb{Z}[x]$, le corps $\mathbb{Q}(\sqrt{2})$, ou encore certaines classes de fonctions avec leurs opérateurs de dérivation et d'intégration.

2.2 Calcul logiciel et complexité

Un programme n'est qu'une suite d'instructions destinées au processeur. En première approximation, on supposera que toutes s'exécutent en temps constant. On estimera donc l'efficacité d'une méthode de calcul en comptant le nombre d'opérations élémentaires qu'elle réalise sur les entiers (addition, soustraction, multiplication, comparaison) et sur la mémoire (stockage et lecture à adresses arbitraires) ; on étudiera alors le comportement asymptotique de cette quantité lorsque la taille des paramètres tend vers l'infini.

La théorie de la complexité formalise cela rigoureusement en considérant des ordinateurs idéalisés appelés « machines de Turing » ; ce modèle diffère sensiblement du matériel existant mais on obtient généralement des complexités comparables quel que soit le modèle de machine retenu. Nous nous contenterons donc de la définition floue ci-dessous.

Définition. *La complexité d'un programme est le nombre d'opérations élémentaires qu'il effectue.*

Dans ce cours, on se contentera d'évaluer le comportement asymptotique de cette quantité lorsque les paramètres dont dépend le programme, à savoir ses arguments, grandissent. Pour cela, on utilisera les notations de Landau rappelées ci-dessous.

Définition. *Soient f et g deux fonctions de $\mathbb{N} \to \mathbb{R}$. On dit que*
- *f est dominée par g, et on note $f = O(g)$, lorsque la quantité f/g est bornée.*
- *f et g sont équivalentes, et on note $f \sim g$, lorsque la quantité f/g a pour limite* 1.
- *f est négligeable devant g, et on note $f = o(g)$, lorsque la quantité f/g a pour limite* 0.

Exemple. *La complexité du programme*

```
sage: def multiplication(a,b):
sage:     r=0
sage:     for i in range(0,a):
sage:         r=r+b
sage:     return r
```

est $O(a)$ alors que celle du programme

```
sage: def multiplication(a,b):
sage:     return a*b
```

est $O(1)$.

Exercice. *Donner la complexité des programmes implantés au premier chapitre.*

Exercice. *Quels sont les complexités des programmes ci-dessous ?*

```
sage: def fibonacci(n):
sage:     u=0
sage:     v=1
sage:     while(n>0):
sage:         (u,v)=(v,u+v)
sage:         n=n-1
sage:     return u
```

```
sage: def fibonacci(n):
sage:     m=matrix([[0,1],[1,1]])
sage:     v=matrix([[0],[1]])
sage:     return (m**n*v)[0,0]
```

Le temps n'est pas la seule ressource dont on ne dispose qu'en quantité limitée. À tout moment, les ordinateurs ne peuvent en effet stocker qu'une quantité finie d'information.

2.3 Tri naïf

Comment trieriez-vous une liste d'entiers ? Une manière de procéder serait 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. Écrivez d'abord une fonction identifiant le plus petit élément d'une liste. De quoi avez-vous alors besoin pour intervertir cet élément avec le premier ?

```
sage: def tri_naif(x):
sage:     for i in range(0,len(x)):
sage:         (k,m)=(i,x[i])
sage:         for j in range(i+1,len(x)):
sage:             if m>x[j]: (k,m)=(j,x[j])
sage:         (x[i],x[k])=(x[k],x[i])
sage:     return x
```

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.

2.4 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.

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

On peut alors définir notre tri par récurrence :

```
sage: def tri_fusion(w):
sage:     if len(w)<2: return w
sage:     x=[w[i] for i in range(0,len(w)//2)]
sage:     y=[w[i] for i in range(len(w)//2,len(w))]
sage:     x=tri_fusion(x)
sage:     y=tri_fusion(y)
sage:     z=fusion(x,y)
sage:     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.

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 [2].

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(9223372036854775808L)
sage: c_ulong(2**64)
c_ulong(0L)
```

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 $\{0, \ldots, 2^{32} - 1\}$. Nous représenterons donc de grands entiers comme vecteurs d'entiers machines par la correspondance que donne la décomposition en base $B = 2^{32}$:

$$(x_0, x_1, \ldots, x_{n-1}) \longleftrightarrow \sum_{i=0}^{n-1} x_i B^i$$

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 :

```
sage: B=2**32
sage: def addition(x,y):
sage:     z=[0 for i in range(0,max(len(x),len(y))+1)]
sage:     for i in range(0,len(z)):
sage:         if i<len(x): z[i]+=x[i]
sage:         if i<len(y): z[i]+=y[i]
```

```
sage:         if z[i]>=B:
sage:             z[i+1]+=z[i]//B
sage:             z[i]=z[i]%B
sage:     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.

```
sage: B=2**32
sage: def multiplication(x,y):
sage:     z=[0 for i in range(0,len(x)+len(y)+1)]
sage:     for i in range(0,len(x)):
sage:         for j in range(0,len(y)):
sage:             k=i+j
sage:             z[k]+=x[i]*y[j]
sage:             while z[k]>=B:
sage:                 z[k+1]+=z[k]//B
sage:                 z[k]=z[k]%B
sage:                 k+=1
sage:     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 $O(n^{\log_2 3})$.

```
sage: B=2**32
sage: def karatsuba(x,y):
sage:     k=len(x)//2
sage:     if k<3: return multiplication(x,y)
sage:     x0=[x[i] for i in range(0,k)]
sage:     y0=[y[i] for i in range(0,k)]
sage:     x1=[x[i] for i in range(k,len(x))]
sage:     y1=[y[i] for i in range(k,len(y))]
sage:     z0=karatsuba(x0,y0)
sage:     z2=karatsuba(x1,y1)
sage:     z1=karatsuba(addition(x0,x1),addition(y0,y1))
```

```
sage:     z1=soustraction(soustraction(z1,z0),z2)
sage:     z=[0 for i in range(0,len(x)+len(y)+1)]
sage:     for i in range(0,len(z)):
sage:         if i in range(0,len(z0)): z[i]+=z0[i]
sage:         if i-k in range(0,len(z1)): z[i]+=z1[i-k]
sage:         if i-2*k in range(0,len(z2)): z[i]+=z2[i-2*k]
sage:         if z[i]>=B:
sage:             z[i+1]+=z[i]//B
sage:             z[i]=z[i]%B
sage:     return z
```

3.3 Exponentiation

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

```
sage: def puissance(x,n):
sage:     r=1;
sage:     for i in range(0,n):
sage:         r=r*x
sage:     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 :

```
sage: def puissance(x,n):
sage:     r=1;
sage:     y=x;
sage:     m=n;
sage:     while m>0:
sage:         if m%2==1: r=r*y;
sage:         m=m//2;
sage:         y=y*y;
sage:     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.

```
sage: def euclide(x,y):
sage:     while y>0:
sage:         (x,y)=(y,x%y)
sage:     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 $ux + vy = \text{pgcd}(x, y)$. 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.

```
sage: def bezout(x,y):
sage:     u=1; s=0
sage:     v=0; t=1
sage:     while y>0:
sage:         q=x//y
sage:         (x,y)=(y,x-q*y)
sage:         (u,v)=(v,u-q*v)
sage:         (s,t)=(t,s-q*t)
sage:     return (x,u,s)
```

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 :

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

Parmi les méthodes permettant de tester la primalité en temps polynomial, on ignore en pratique celles qui sont déterministes [1] au profit d'algorithmes probabilistes plus rapides exploitant le fait suivant.

Proposition (dit « petit théorème de Fermat »). *Soit p un nombre premier. Pour tout $x \in \mathbb{Z}$ on a $x^p = x \bmod p$.*

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 [5]). *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. Une légère relaxation de ces conditions donne le critère suivant :

Théorème (Rabin [9]). *Pour $n > 4$, l'ensemble des éléments non nuls $x \in \mathbb{Z}/n\mathbb{Z}$ vérifiant $x^{n-1} \neq 1$ ou $\operatorname{pgcd}(x^{(n-1)/2^j} - 1, n) \neq 1$, n pour un certain $j \in \mathbb{N}$ est vide lorsque n est premier et de cardinal au moins $\frac{3}{4}(n-1)$ lorsque n est composé.*

En écrivant la condition sur le PGCD de manière efficace à évaluer, on obtient l'algorithme ci-dessous, appelé test de primalité de Miller–Rabin.

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

Proposition. *Si n est premier, ce programme renvoie systématiquement* `True`*. Si n est composé, ce programme renvoie* `False` *avec probabilité $1 - 4^{-k}$.*

On peut donc ajuster la valeur de k selon la certitude désirée.

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 $\mathbb{Z}/n\mathbb{Z} = \prod \mathbb{Z}/p^{\alpha_p}\mathbb{Z}$ 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 la prétention d'en donner dans de cours qu'un bref aperçu.

Proposition. *Le nombre de fonctions injectives de $\{1, \dots, x\}$ dans $\{1, \dots, y\}$ est $\prod_{i=0}^{x-1}(y-i)$, pour un total de y^x fonctions.*

Corollaire. *Si, dans un ensemble de cardinal y, on choisit $\sqrt{y}$ éléments au hasard avec répétition, la probabilité qu'un même élément ait été choisi deux fois converge vers $e^{-1/2} \approx 0,6$ lorsque y tend vers l'infini.*

Démonstration. Poser $x = \sqrt{y}$ dans la proposition et utiliser la formule de Stirling. □

Exemple (dit « paradoxe des anniversaires »). *Les années comptent* 365 *jours. Dans une classe de* 20 *élèves, il y a environ* 40% *de chance que deux aient le même anniversaire.*

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 $\mathrm{pgcd}(\alpha - \beta, n) \neq 1, n$. 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 est inférieur à $y = \sqrt{n}$. En considérant $x = \sqrt{y} = n^{1/4}$ entiers, la probabilité que la différence de deux d'entre eux soit un facteur non trivial de n est donc de 40%. C'est la méthode de factorisation de Shanks [10].

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

Un fois un facteur trouvé, on divise et on recommence :

```
sage: def facteurs(n):
sage:     if n==1: return []
sage:     if premier(n,ceil(99*log(n))): return [n]
sage:     f=facteur(n)
sage:     return facteurs(f)+facteurs(n//f)
```

L'inconvénient majeur de la méthode de Shanks est son important coût en mémoire. Pour corriger ce défaut, la méthode de Pollard [7] consiste à chercher les couples (α, β) parmi les valeurs d'une fonction pseudo-aléatoire $f : \{1, \dots, n\} \to \{1, \dots, n\}$ qui préserve les résidus ; pour chercher ces couples efficacement, on exploite l'observation :

$$\text{Si } f^a(1) = f^b(1) \text{ et } a \geqslant 2b, \text{ alors } f^{2(a-b)}(1) = f^{(a-b)}(1).$$

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

```
sage: def facteur(n):
sage:     for c in range(1,n):
sage:         x=1
sage:         y=1
sage:         k=0
sage:         m=floor(99*n**0.5)
sage:         while gcd(x-y,n)%n<2:
sage:             k=k+1
sage:             if k>m: break
sage:             x=(x*x+c)%n
sage:             y=(y*y+c)%n
sage:             y=(y*y+c)%n
sage:         if gcd(x-y,n)%n<2: continue
sage:         x=1
sage:         y=1
sage:         for i in range(0,k):
sage:             y=(y*y+c)%n
```

```
sage:         for i in range(0,k):
sage:             if gcd(x-y,n)%n>1: return gcd(x-y,n)
sage:             x=(x*x+c)%n
sage:             y=(y*y+c)%n
sage:     return 0
```

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 [3].

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 [4] 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.

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

```
sage: def relations(n,P):
sage:     R=[]
sage:     while len(R)<2*len(P):
sage:         x=randrange(ceil(sqrt(n)),n)
sage:         r=relation(x*x%n,P)
sage:         if r==[]: continue
sage:         R.append([x,r])
sage:     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.

```
sage: def kraitchik(n):
sage:     P=prime_range(2,floor(exp(sqrt(log(n)))))
```

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

Chapitre 4

Arithmétique des polynômes

Une fois les anneaux d'entiers élémentaires maîtrisés, il est naturel de se tourner vers 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 : si l'on stocke un polynôme comme la liste de ses coefficients, alors l'addition consiste simplement à effectuer cette opération coefficient par coefficient.

Exercice. *Écrire un programme qui, étant donnés deux polynômes représentés par la liste de leurs coefficients, renvoie la liste des coefficients du polynôme produit.*

Faire de même pour le PGCD. Et pour les coefficients de Bézout.

4.1 Représentation

Avant que Sage puisse manipuler des polynômes, il faut définir l'anneau sous-jacent. Comme vous le savez, les propriétés de $\mathbb{R}[x]$ et $\mathbb{Z}[x]$ sont bien différentes, notamment en ce qui concerne l'irréductibilité.

```
sage: QQx.<x>=PolynomialRing(QQ)
```

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

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

Afin de se familiariser avec ces fonctions, nous allons commencer par écrire deux programmes relativement élémentaires.

Exercice. *Écrire un programme qui calcule la dérivée d'un polynôme donnée.*

Exercice. *Écrire un programme qui calcule le quotient et le reste de la division euclidienne d'un polynôme P par un autre polynôme Q donné.*

```
sage: def division(P,Q):
sage:     q=0
sage:     while P.degree()>=Q.degree():
```

```
sage:         c=P.coeffs()[P.degree()]/Q.coeffs()[Q.degree()]*x**(P.degree()-
Q.degree())
sage:         P-=c*Q
sage:         q+=c
sage:     return (q,P)
```

4.2 Méthode de Sturm

Soit P un polynôme de $\mathbb{R}[X]$. On souhaite en déterminer les racines à une précision donnée. Nous allons pour cela nous appuyer sur la théorie de Sturm dont le résultat principal est le suivant.

Théorème. *Soit P un polynôme de $\mathbb{R}[x]$ sans facteur carré; posons*

$$P_0 = P \qquad P_1 = P' \qquad P_{n+1} = -\text{reste}(P_{n-1}, P_n)$$

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

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

```
QQX.<X>=PolynomialRing(QQ)

def suite(P):
    d=P.degree()
    S=[0 for i in range(0,d+1)]
    S[0]=P
    S[1]=P.derivative()
    for i in range(2,d+1):
        if S[i-1]==0:
            break
        S[i]=-(S[i-2]%S[i-1])
    return S

def sigma(S,a):
    c=0
    v=S[0].subs(a)
    d=S[0].degree()
    for i in range(1,d+1):
        w=S[i].subs(a)
        if w!=0:
            if v*w<0:
                c=c+1
            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(P,a,b,p):
    S=suite(P)
    while b-a>p:
        c=(a+b)/2
        n=sigma(S,a)-sigma(S,c)
        m=sigma(S,c)-sigma(S,b)
        if n==0:
            a=c
        elif m==0:
            b=c
        else:
            return dichotomie(P,a,c,p)+dichotomie(P,c,b,p)
    return [a]
```

Remarquons d'abord que, ses racines multiples étant communes à P et à P', on peut aisément les isoler en calculant $\text{pgcd}(P, P')$ ce que l'on sait faire efficacement ; cela nous ramène à considérer le cas de polynômes P sans facteur carré. La théorie de Sturm [14] s'applique alors.

```
def racines(P):
    if P.degree()==0:
        return []
    G=gcd(P,P.derivative())
    a=RealField(300)(10^60)
    return dichotomie(P//G,-a,a,1/a)+racines(G)
```

4.3 Factorisation sur les corps finis

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

```
sage: FF.<X>=PolynomialRing(FiniteField(7))
sage: P=FF.random_element(degree=17)
```

Plus haut, nous avons exploité l'égalité $P = \text{pgcd}(P, P') \cdot \frac{P}{\text{pgcd}(P,P')}$ 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 $\mathbb{F}_q$.*

```
sage: def lemmeQ(P):
sage:     c=P.coeffs()
sage:     p=P.parent().base_ring().characteristic()
sage:     return sum([c[p*i]*X**i for i in range(0,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.

```
sage: def factorisation(P):
sage:     if P.degree()==0: return []
sage:     if P.degree()==1: return [P]
sage:     p=P.parent().base_ring().characteristic()
sage:     D=P.derivative()
sage:     if D==0:
sage:         Q=lemmeQ(P)
```

```
sage:         F=factorisation(Q)
sage:         return [f.subs(X=X**p) for f in F]
sage:     G=gcd(P,D)
sage:     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.

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

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

— En se restreignant à un sous-ensemble discret des grandeurs continues considérées, par exemple les polynômes ou encore les expressions algébriques en un nombre fini de fonctions usuelles. Cela constitue du **calcul formel** proprement dit. 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.

— En discrétisant ces grandeurs. Cela constitue du **calcul numérique**. Toute la difficulté est alors de majorer rigoureusement les termes d'erreurs des approximations effectuées.

Ce chapitre présente quelques outils élémentaires du calcul numérique. Nous les implanterons nous même avant d'indiquer les fonctions Sage correspondantes. Par soucis de complétude, nous mentionnerons aussi les fonctions formelles associées lorsqu'elles existent.

5.1 Fonctions

Sage possède deux notions de « fonction » bien différentes. En premier lieu, celle de **programme informatique** que jusqu'à présent nous avons utilisée exclusivement ; c'est la plus polyvalente et la mieux adaptée au calcul numérique.

```
sage: def fonction(x,y): return x+y
sage: fonction==lambda x,y:x+y
```

Elle s'oppose à la notion d'**expression algébrique** qui n'est autre qu'une arborescence décrivant une combinaison de fonctions usuelles et de variables abstraites. C'est l'objet du calcul formel proprement dit.

```
sage: var('x','y')
sage: f(x,y)=x+y
```

On évitera autant que possible de recourir à cette seconde notion. Comme mentionné en introduction, elle ne se prête aux calculs que dans certains cas restreints ; par exemple, pour trouver les zéros d'un polynôme :

```
sage: var('x')
sage: solve(x**3-x-2,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(x**7-x-2,x)
[0 == x^7 - x - 2]
```

On aura donc principalement recours au calcul numérique et, pour davantage de flexibilité, on préfèrera spécifier toute fonction sous forme de programme informatique. On obtiendra ainsi par exemple :

```
sage: find_root(lambda x:x^7-x-2,0,2)
1.1796938907415153
```

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'on trouve un entier n pour lequel $\{x_{n+1}, x_{n+2}, \ldots, x_{n+N}\} \subset [\ell - \varepsilon, \ell + \varepsilon]$ avec par exemple $N = 10$ et $\varepsilon = 0.001$; cela peut être pertinent en sciences physiques si ces valeurs majorent les comportements que l'on considère négligeables. Lorsqu'on choisira de mettre en œuvre cette technique il faudra cependant être pleinement conscient qu'elle n'a aucun sens rigoureux : la condition précédente peut être vérifiée même quand (x_n) diverge grossièrement.

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*

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

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.*

```
sage: def adherences(l):
....:     L=[]
....:     x=0.5
....:     for k in range(0,10^4):
....:         x=l*x*(1-x)
....:     for k in range(0,10^4):
....:         L.append(x)
....:         x=l*x*(1-x)
....:     L.sort()
....:     r=0
```

```
....:     R=[]
....:     for k in range(0,10^3):
....:         if abs(L[10*k]-r)>0.001:
....:             r=L[10*k]
....:         else:
....:             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.

```
sage: G=[]
....: for ll in range(1,10^3):
....:     l=2.5+1.5*ll/10^3
....:     G+=[(l,y) for y in adherences(l)]
....:
sage: show(points(G,pointsize=1))
```

5.3 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*

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

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 $\{0,\dots,n\}$ avec $n=\lfloor\frac{b-a}{\varepsilon}\rfloor$ 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*

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

Remarque. *La somme ci-dessus est fortuitement identique à celle correspondant à la méthode des trapèzes, 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 $V_{n+1}(r) = \int_{-r}^{r} V_n(\sqrt{1-x^2})dx$.
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)
```

5.4 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)
    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
```

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.

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

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(0,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.

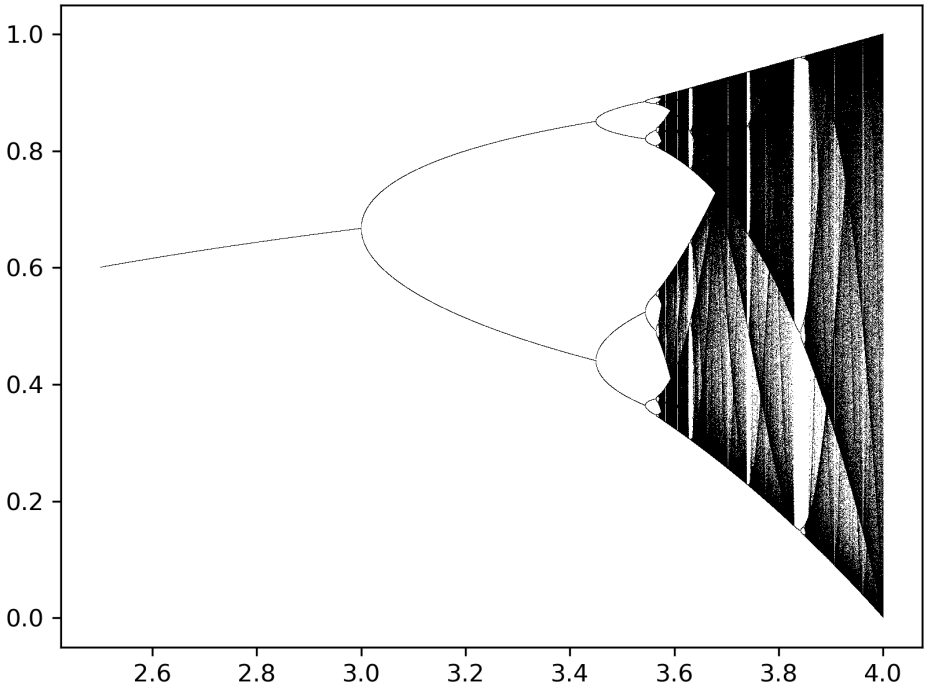

FIGURE 5.1 – Valeurs d'adhérence de la suite logistique pour $\lambda \in [5/2, 4]$.

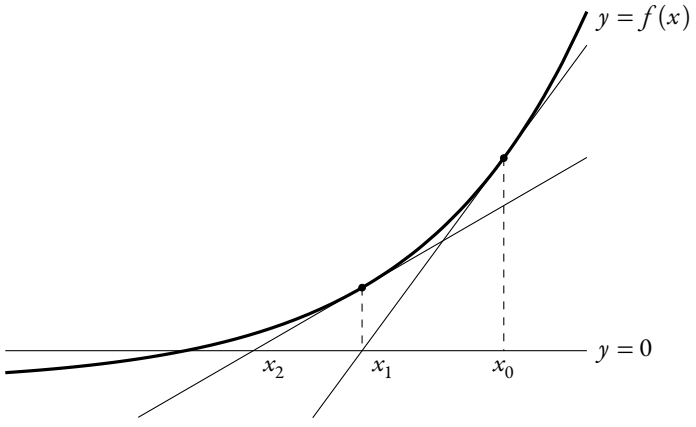

FIGURE 5.2 – Deux itérations de la méthode de Newton.

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.*

5.5 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 $n \in \left\{1, \dots, \left\lfloor \frac{\eta}{\varepsilon} \right\rfloor\right\}$.

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.

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 :*

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

Convergence simple :

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

```
sage: var('x,t')
sage: c=animate([
sage:   plot([gamma(x),integral_numerical(t^(x-1)*exp(-t),0,N)],0.2,4)
sage:   for N in range(1,10)],xmin=0.2,xmax=4,ymin=0,ymax=6)
sage: 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 $n \in \{0, \dots, 10\}$. Que dire de la convergence aux extrémités de l'intervalle ? Tracer la fonction $\mathscr{F}(\mathscr{F}(f))$.

```
sage: def integral_complex(f,a,b):
sage:   r=integral_numerical(lambda x:real(f(x)),a,b)[0]
sage:   i=integral_numerical(lambda x:imag(f(x)),a,b)[0]
sage:   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 $[\frac{a+2b}{3}, b]$.

```
sage: c=[0,1]
sage: d=[0,0,1]
sage: CD=[(c,d)]
sage: for n in range(0,10):
sage:     (e,f)=([],[0])
sage:     for i in range(0,len(c)):
sage:         if d[i+1]>0:
sage:             e.append(c[i])
sage:             f.append(d[i+1])
sage:         else:
sage:             e.append(c[i])
sage:             e.append((2*c[i]+c[i+1])/3)
sage:             e.append((c[i]+2*c[i+1])/3)
sage:             f.append(0)
sage:             f.append((d[i]+d[i+2])/2)
sage:             f.append(0)
sage:     CD.append((e,f))
sage:     (c,d)=(e,f)
sage:
sage: def escalier(x,n):
sage:     (c,d)=CD[n]
sage:     i=0
sage:     while x>c[i+1]: i=i+1
sage:     if d[i+1]>0: return d[i+1]
sage:     return ((x-c[i])*d[i+2]+(c[i+1]-x)*d[i])/(c[i+1]-c[i])
sage:
sage: plot(lambda x:escalier(x,4),0,1)
sage: E=animate([
sage:   plot(lambda x:escalier(x,n),0,1)
sage:   for n in range(0,11)],xmin=0,xmax=1,ymin=0,ymax=1)
sage: E.show()
```

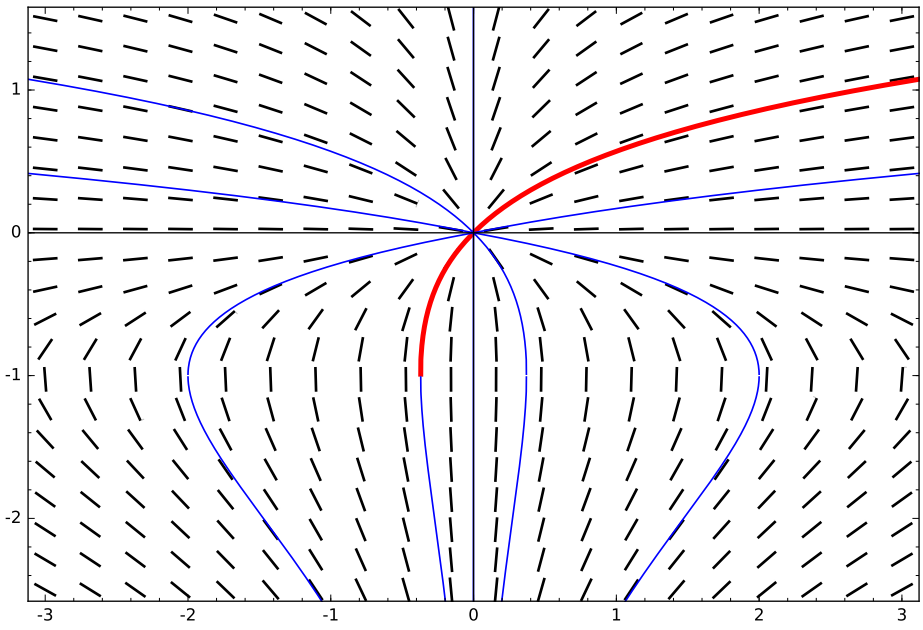

FIGURE 5.3 – Champ de vecteurs et solutions de l'équation différentielle $y' = \frac{y}{x(1+y)}$. En rouge, la fonction W de Lambert.

Chapitre 6

Applications en cryptographie

6.1 Chiffrement

6.2 Signature

6.3 Partage de clefs

6.4 Attaques

Bibliographie

[1] Manindra Agrawal, Neeraj Kayal et Nitin Saxena. "PRIMES is in P".
In : *Annals of Mathematics* 160.2 (2004), pages 781-793.
doi : 10.4007/annals.2004.160.781.

[2] Richard P. Brent et Paul Zimmermann. *Modern Computer Arithmetic*.
Monographs on Applied and Computational Mathematics.
Cambridge University Press, 2010. isbn : 0-521-19469-5.

[3] Earl Canfield, Paul Erdős et Carl Pomerance.
"On a problem of Oppenheim concerning 'factorisatio numerorum"'.
In : *Journal of Number Theory* 17.1 (1983), pages 1-28.
doi : 10.1016/0022-314X(83)90002-1.

[4] Maurice Kraitchik. "Analyse indéterminée du second degré et factorisation". In :
Théorie des Nombres. Tome 2. Gauthier-Villars, 1926.

[5] Gary L. Miller. "Riemann's hypothesis and tests for primality".
In : *Symposium on Theory of Computing — STOC 1975*.
Édité par William C. Rounds et al. Association for Computing Machinery, 1975,
pages 234-239. doi : 10.1145/800116.803773.

[6] *PARI/GP*.
A computer algebra system designed for fast computations in number theory.
The PARI Group. 2014. url : http://pari.math.u-bordeaux.fr/.

[7] John M. Pollard. "A Monte Carlo method for factorization".
In : *BIT Numerical Mathematics* 15.3 (1975), pages 331-334.
doi : 10.1007/BF01933667.

[8] *R*. A Language and Environment for Statistical Computing.
R Development Core Team. 2008. url : http://www.r-project.org/.

[9] Michael O. Rabin. "Probabilistic algorithm for testing primality".
In : *Journal of Number Theory* 12.1 (1980), pages 128-138.
doi : 10.1016/0022-314X(80)90084-0.

[10] Daniel Shanks. "Class number, a theory of factorization, and genera".
In : *1969 Number Theory Institute*. Édité par Donald J. Lewis. Tome 20.
Proceedings of Symposia in Pure Mathematics. American Mathematical Society, 1971,
pages 415-440.

[11] *Singular*. A computer algebra system for polynomial computations.
University of Kaiserslautern. 2014. url : http://www.singular.uni-kl.de/.

[12] William A. Stein et al. *Sage Documentation*. The Sage Development Team. 2014.
Chapitre Sage Tutorial. url : http://www.sagemath.org/doc/tutorial/.

[13] William A. Stein et al. *Sage Mathematics Software*.
The Sage Development Team. 2014. URL : `http://www.sagemath.org/`.

[14] Jacques Charles François Sturm.
"Mémoire sur la résolution des équations numériques".
In : *Bulletin des sciences de Férussac* 11 (1829), pages 419-422.