

ALGORITHMES

COURS DE MATHÉMATIQUES
PREMIÈRE ANNÉE



Algorithmes

Ce recueil regroupe différents chapitres sur les mathématiques en lien avec l'informatique.

Sommaire

1	Algorithmes et mathématiques	1
1	Premiers pas avec Python	1
2	Écriture des entiers	5
3	Calculs de sinus, cosinus, tangente	11
4	Les réels	15
5	Arithmétique – Algorithmes récursifs	20
6	Polynômes – Complexité d'un algorithme	24
2	Zéros des fonctions	31
1	La dichotomie	31
2	La méthode de la sécante	36
3	La méthode de Newton	39
3	Cryptographie	43
1	Le chiffrement de César	43
2	Le chiffrement de Vigenère	47
3	La machine Enigma et les clés secrètes	50
4	La cryptographie à clé publique	56
5	L'arithmétique pour RSA	59
6	Le chiffrement RSA	63
4	Calcul formel	69
1	Premiers pas avec Sage	69
2	Structures de contrôle avec Sage	75
3	Suites récurrentes et preuves formelles	80
4	Suites récurrentes et visualisation	85
5	Algèbre linéaire	95
6	Courbes et surfaces	103
7	Calculs d'intégrales	113
8	Polynômes	117
9	Équations différentielles	124

Algorithmes et mathématiques

Vidéo ■ partie 1. Premiers pas avec Python

Vidéo ■ partie 2. Ecriture des entiers

Vidéo ■ partie 3. Calculs de sinus, cosinus, tangente

Vidéo ■ partie 4. Les réels

Vidéo ■ partie 5. Arithmétique - Algorithmes récursifs

Vidéo ■ partie 6. Polynômes - Complexité d'un algorithme

1. Premiers pas avec Python

Dans cette partie on vérifie d'abord que Python fonctionne, puis on introduira les boucles (`for` et `while`), le test `if ... else ...` et les fonctions.

1.1. Hello world !

Pour commencer testons si tout fonctionne !

Travaux pratiques 1.

1. Définir deux variables prenant les valeurs 3 et 6.
2. Calculer leur somme et leur produit.

Voici à quoi cela ressemble :

Code 1 (*hello-world.py*).

```
>>> a=3
>>> b=6
>>> somme = a+b
>>> print(somme)
9
>>> # Les résultats
>>> print("La somme est", somme)
La somme est 9
>>> produit = a*b
>>> print("Le produit est", produit)
Le produit est 18
```

On retient les choses suivantes :

- On affecte une valeur à une variable par le signe égal =.
- On affiche un message avec la fonction `print()`.
- Lorsque qu'une ligne contient un dièse #, tout ce qui suit est ignoré. Cela permet d'insérer des commentaires, ce qui est essentiel pour relire le code.

Dans la suite on omettra les symboles >>>. Voir plus de détails sur le fonctionnement en fin de section.

1.2. Somme des cubes

Travaux pratiques 2.

1. Pour un entier n fixé, programmer le calcul de la somme $S_n = 1^3 + 2^3 + 3^3 + \dots + n^3$.
2. Définir une fonction qui pour une valeur n renvoie la somme $\Sigma_n = 1 + 2 + 3 + \dots + n$.
3. Définir une fonction qui pour une valeur n renvoie S_n .
4. Vérifier, pour les premiers entiers, que $S_n = (\Sigma_n)^2$.

1.

Code 2 (*somme-cubes.py (1)*).

```
n = 10
somme = 0
for i in range(1,n+1):
    somme = somme + i*i*i
print(somme)
```

Voici ce que l'on fait pour calculer S_n avec $n = 10$.

- On affecte d'abord la valeur 0 à la variable `somme`, cela correspond à l'initialisation $S_0 = 0$.
 - Nous avons défini une **boucle** avec l'instruction `for` qui fait varier i entre 1 et n .
 - Nous calculons successivement S_1, S_2, \dots en utilisant la formule de récurrence $S_i = S_{i-1} + i^3$. Comme nous n'avons pas besoin de conserver toutes les valeurs des S_i alors on garde le même nom pour toutes les sommes, à chaque étape on affecte à `somme` l'ancienne valeur de la somme plus i^3 : `somme = somme + i*i*i`.
 - `range(1,n+1)` est l'ensemble des entiers $\{1, 2, \dots, n\}$. C'est bien les entiers **strictement inférieurs** à $n + 1$. La raison est que `range(n)` désigne $\{0, 1, 2, \dots, n - 1\}$ qui contient n éléments.
2. Nous savons que $\Sigma_n = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$ donc nous n'avons pas besoin de faire une boucle :

Code 3 (*somme-cubes.py (2)*).

```
def somme_entiers(n):
    return n*(n+1)/2
```

Une **fonction** en informatique est similaire à une fonction mathématique, c'est un objet qui prend en entrée des variables (dites variables formelles ou variables muettes, ici n) et retourne une valeur (un entier, une liste, une chaîne de caractères,... ici $\frac{n(n+1)}{2}$).

3. Voici la fonction qui retourne la somme des cubes :

Code 4 (*somme-cubes.py (3)*).

```
def somme_cubes(n):
    somme = 0
```

```

for i in range(1,n+1):
    somme = somme + i**3
return somme

```

4. Et enfin on vérifie que pour les premiers entiers $S_n = \left(\frac{n(n+1)}{2}\right)^2$, par exemple pour $n = 12$:

Code 5 (*somme-cubes.py (4)*).

```

n = 12
if somme_cubes(n) == (somme_entiers(n)**2):
    print("Pour n=", n, "l'assertion est vraie.")
else:
    print("L'assertion est fausse!")

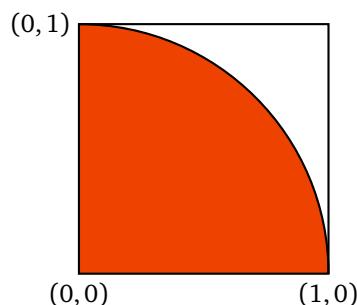
```

On retient :

- Les puissances se calculent aussi avec `**` : 5^2 s'écrit `5*5` ou `5**2`, 5^3 s'écrit `5*5*5` ou `5**3`...
- Une fonction se définit par `def ma_fonction(variable)` : et se termine par `return resultat`.
- `if condition: ... else: ...` exécute le premier bloc d'instructions si la condition est vraie ; si la condition est fausse cela exécute l'autre bloc.
- Exemple de conditions
 - `a < b` : $a < b$,
 - `a <= b` : $a \leq b$,
 - `a == b` : $a = b$,
 - `a != b` : $a \neq b$.
- Attention ! Il est important de comprendre que `a==b` vaut soit vrai ou faux (on compare a et b) alors qu'avec `a=b` on affecte dans a la valeur de b .
- Enfin en Python (contrairement aux autres langages) c'est l'indentation (les espaces en début de chaque ligne) qui détermine les blocs d'instructions.

1.3. Calcul de π au hasard

Nous allons voir qu'il est possible de calculer les premières décimales de π par la méthode de Monte-Carlo, c'est à dire avec l'aide du hasard. On considère le carré de côté 1, le cercle de rayon 1 centré à l'origine, d'équation $x^2 + y^2 = 1$, et la portion de disque dans le carré (voir la figure).



Travaux pratiques 3.

1. Calculer l'aire du carré et de la portion de disque.
2. Pour un point (x, y) tiré au hasard dans le carré, quelle est la probabilité que le point soit en fait dans la portion de disque ?

3. Tirer un grand nombre de points au hasard, compter ceux qui sont dans la portion de disque.
4. En déduire les premières décimales de π .

Voici le code :

Code 6 (*pi-hasard.py*).

```
import random          # Module qui génère des nombres aléatoires

Tir = 0                # Numéro du tir
NbTirDansLeDisque = 0  # Nombre de tirs dans le disque

while (Tir < 1000):
    Tir = Tir + 1
    # On tire au hasard un point (x,y) dans [0,1] x [0,1]
    x = random.random()
    y = random.random()
    if (x*x+y*y <= 1):    # On est dans le disque
        NbTirDansLeDisque = NbTirDansLeDisque + 1

MonPi = 4*NbTirDansLeDisque / Tir
print("Valeur \u00e9xp\u00e9imentale \u00e0 \u00c9tendue : \u2022%0.3f" %MonPi)
```

Commentaires :

- Un petit calcul prouve que l'aire de la portion de disque est $\frac{\pi}{4}$, l'aire du carré est 1. Donc la probabilité de tomber dans le disque est $\frac{\pi}{4}$.
- Pour tirer un nombre au hasard on utilise une fonction `random()` qui renvoie un nombre réel de l'intervalle $[0, 1[$. Bien sûr à chaque appel de la fonction `random()` le nombre obtenu est différent !
- Cette fonction n'est pas connue par défaut de Python, il faut lui indiquer le nom du **module** où elle se trouve. En début de fichier on ajoute `import random` pour le module qui gère les tirages au hasard. Et pour indiquer qu'une fonction vient d'un module il faut l'appeler par `module.fonction()` donc ici `random.random()` (module et fonction portent ici le même nom !).
- La boucle est `while condition: ...` Tant que la condition est vérifiée les instructions de la boucle sont exécutées. Ici `Tir` est le compteur que l'on a initialisé à 0. Ensuite on commence à exécuter la boucle. Bien sûr la première chose que l'on fait dans la boucle est d'incrémenter le compteur `Tir`. On continue jusqu'à ce que l'on atteigne 999. Pour `Tir=1000` la condition n'est plus vraie et le bloc d'instructions du `while` n'est pas exécuté. On passe aux instructions suivantes pour afficher le résultat.
- À chaque tir on teste si on est dans la portion de disque ou pas à l'aide de l'inégalité $x^2 + y^2 \leqslant 1$.
- Cette méthode n'est pas très efficace, il faut beaucoup de tirs pour obtenir le deux premières décimales de π .

1.4. Un peu plus sur Python

- Le plus surprenant avec Python c'est que c'est **l'indentation** qui détermine le début et la fin d'un bloc d'instructions. Cela oblige à présenter très soigneusement le code.
- Contrairement à d'autres langages on n'a pas besoin de déclarer le type de variable. Par exemple lorsque l'on initialise une variable par `x=0`, on n'a pas besoin de préciser si `x` est un entier ou un réel.
- Nous travaillerons avec la version 3 (ou plus) de Python, que l'on appelle par `python` ou `python3`. Pour savoir si vous avez la bonne version tester la commande `4/3`. Si la réponse est `1.3333...` alors tout est ok. Par contre avec les versions 1 et 2 de Python la réponse est `1` (car il considérait que c'est

quotient de la division euclidienne de deux entiers).

- La première façon de lancer Python est en ligne de commande, on obtient alors l'invite >>> et on tape les commandes.
- Mais le plus pratique est de sauvegarder ses commandes dans un fichier et de faire un appel par `python monfichier.py`
- Vous trouverez sans problème de l'aide et des tutoriels sur internet !

Mini-exercices. 1. Soit le produit $P_n = (1 - \frac{1}{2}) \times (1 - \frac{1}{3}) \times (1 - \frac{1}{4}) \times \cdots \times (1 - \frac{1}{n})$. Calculer une valeur approchée de P_n pour les premiers entiers n .

2. Que vaut la somme des entiers i qui apparaissent dans l'instruction `for i in range(1,10)`. Idem pour `for i in range(11)`. Idem pour `for i in range(1,10,2)`. Idem pour `for i in range(0,10,2)`. Idem pour `for i in range(10,0,-1)`.
3. On considère le cube $[0,1] \times [0,1] \times [0,1]$ et la portion de boule de rayon 1 centrée à l'origine incluse dans ce cube. Faire les calculs de probabilité pour un point tiré au hasard dans le cube d'être en fait dans la portion de boule. Faire une fonction pour le vérifier expérimentalement.
4. On lance deux dés. Expérimenter quelle est la probabilité que la somme soit 7, puis 6, puis 3 ? Quelle est la probabilité que l'un des deux dés soit un 6 ? d'avoir un double ? La fonction `randint(a, b)` du module `random` retourne un entier k au hasard, vérifiant $a \leq k \leq b$.
5. On lance un dé jusqu'à ce que l'on obtienne un 6. En moyenne au bout de combien de lancer s'arrête-t-on ?

2. Écriture des entiers

Nous allons faire un peu d'arithmétique : le quotient de la division euclidienne `//`, le reste `%` (modulo) et nous verrons l'écriture des entiers en base 10 et en base 2. Nous utiliserons aussi la notion de listes et le module `math`.

2.1. Division euclidienne et reste, calcul avec les modulo

La division euclidienne de a par b , avec $a \in \mathbb{Z}$ et $b \in \mathbb{Z}^*$ s'écrit :

$$a = bq + r \quad \text{et} \quad 0 \leq r < b$$

où $q \in \mathbb{Z}$ est le **quotient** et $r \in \mathbb{N}$ est le **reste**.

En Python le quotient se calcule par : `a // b`. Le reste se calcule par `a % b`. Exemple : `14 // 3` retourne 4 alors que `14 % 3` (lire 14 modulo 3) retourne 2. On a bien $14 = 3 \times 4 + 2$.

Les calculs avec les modulus sont très pratiques. Par exemple si l'on souhaite tester si un entier est pair, ou impair cela revient à un test modulo 2. Le code est `if (n%2 == 0): ... else: ...`. Si on besoin de calculer $\cos(n\frac{\pi}{2})$ alors il faut discuter suivant les valeurs de `n%4`.

Appliquons ceci au problème suivant :

Travaux pratiques 4.

Combien y-a-t-il d'occurrences du chiffre 1 dans les nombres de 1 à 999 ? Par exemple le chiffre 1 apparaît une fois dans 51 mais deux fois dans 131.

Code 7 (`nb-un.py`).

```
NbDeUn = 0
for N in range(1,999+1):
```

```

ChiffreUnite = N % 10
ChiffreDizaine = (N // 10) % 10
ChiffreCentaine = (N // 100) % 10
if (ChiffreUnite == 1):
    NbDeUn = NbDeUn + 1
if (ChiffreDizaine == 1):
    NbDeUn = NbDeUn + 1
if (ChiffreCentaine == 1):
    NbDeUn = NbDeUn + 1
print("Nombre d'occurrences du chiffre '1':", NbDeUn)

```

Commentaires :

- Comment obtient-on le chiffre des unités d'un entier N ? C'est le reste modulo 10, d'où l'instruction `ChiffreUnite = N % 10`.
- Comment obtient-on le chiffre des dizaines ? C'est plus délicat, on commence par effectuer la division euclidienne de N par 10 (cela revient à supprimer le chiffre des unités, par exemple si $N = 251$ alors $N // 10$ retourne 25). Il ne reste plus qu'à calculer le reste modulo 10, (par exemple `(N // 10) % 10` retourne le chiffre des dizaines 5).
- Pour le chiffre des centaines on divise d'abord par 100.

2.2. Écriture des nombres en base 10

L'écriture décimale d'un nombre, c'est associer à un entier N la suite de ses chiffres $[a_0, a_1, \dots, a_n]$ de sorte que a_i soit le i -ème chiffre de N . C'est-à-dire

$$N = a_n 10^n + a_{n-1} 10^{n-1} + \cdots + a_2 10^2 + a_1 10 + a_0 \quad \text{et } a_i \in \{0, 1, \dots, 9\}$$

a_0 est le chiffre des unités, a_1 celui des dizaines, a_2 celui des centaines,...

Travaux pratiques 5.

1. Écrire une fonction qui à partir d'une liste $[a_0, a_1, \dots, a_n]$ calcule l'entier N correspondant.
2. Pour un entier N fixé, combien a-t-il de chiffres ? On pourra s'aider d'une inégalité du type $10^n \leq N < 10^{n+1}$.
3. Écrire une fonction qui à partir de N calcule son écriture décimale $[a_0, a_1, \dots, a_n]$.

Voici le premier algorithme :

Code 8 (*decimale.py (1)*).

```

def chiffres_vers_entier(tab):
    N = 0
    for i in range(len(tab)):
        N = N + tab[i] * (10 ** i)
    return N

```

La formule mathématique est simplement $N = a_n 10^n + a_{n-1} 10^{n-1} + \cdots + a_2 10^2 + a_1 10 + a_0$. Par exemple `chiffres_vers_entier([4, 3, 2, 1])` renvoie l'entier 1234.

Expliquons les bases sur les *listes* (qui s'appelle aussi des *tableaux*)

- En Python une liste est présentée entre des crochets. Par exemple pour `tab = [4, 3, 2, 1]` alors on accède aux valeurs par `tab[i]` : `tab[0]` vaut 4, `tab[1]` vaut 3, `tab[2]` vaut 2, `tab[3]` vaut 1.

- Pour parcourir les éléments d'un tableau le code est simplement `for x in tab`, x vaut alors successivement 4, 3, 2, 1.
- La longueur du tableau s'obtient par `len(tab)`. Pour notre exemple `len([4,3,2,1])` vaut 4. Pour parcourir toutes les valeurs d'un tableau on peut donc aussi écrire `for i in range(len(tab))`, puis utiliser `tab[i]`, ici i variant ici de 0 à 3.
- La liste vide est seulement notée avec deux crochets : `[]`. Elle est utile pour initialiser une liste.
- Pour ajouter un élément à une liste `tab` existante on utilise la fonction `append`. Par exemple définissons la liste vide `tab = []`, pour ajouter une valeur à la fin de la liste on saisit : `tab.append(4)`. Maintenant notre liste est `[4]`, elle contient un seul élément. Si on continue avec `tab.append(3)`. Alors maintenant notre liste a deux éléments : `[4, 3]`.

Voici l'écriture d'un entier en base 10 :

Code 9 (*decimale.py (2)*).

```
def entier_vers_chiffres(N):
    tab = []
    n = floor(log(N,10))    # le nombre de chiffres est n+1
    for i in range(0,n+1):
        tab.append((N // 10 ** i) % 10)
    return tab
```

Par exemple `entier_vers_chiffres(1234)` renvoie le tableau `[4, 3, 2, 1]`. Nous avons expliqué tout ce dont nous avions besoin sur les listes au-dessus, expliquons les mathématiques.

- Décomposons \mathbb{N}^* sous la forme $[1, 10[\cup [10, 100[\cup [100, 1000[\cup [1000, 10000[\cup \dots$. Chaque intervalle est du type $[10^n, 10^{n+1}[$. Pour $N \in \mathbb{N}^*$ il existe donc $n \in \mathbb{N}$ tel que $10^n \leq N < 10^{n+1}$. Ce qui indique que le nombre de chiffres de N est $n + 1$.
Par exemple si $N = 1234$ alors $1000 = 10^3 \leq N < 10000 = 10^4$, ainsi $n = 3$ et le nombre de chiffres est 4.
- Comment calculer n à partir de N ? Nous allons utiliser le logarithme décimal \log_{10} qui vérifie $\log_{10}(10) = 1$ et $\log_{10}(10^i) = i$. Le logarithme est une fonction croissante, donc l'inégalité $10^n \leq N < 10^{n+1}$ devient $\log_{10}(10^n) \leq \log_{10}(N) < \log_{10}(10^{n+1})$. Et donc $n \leq \log_{10}(N) < n + 1$. Ce qui indique donc que $n = E(\log_{10}(N))$ où $E(x)$ désigne la partie entière d'un réel x .

2.3. Module math

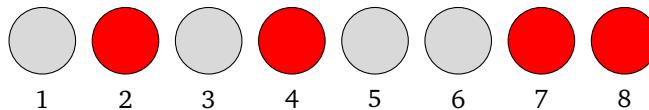
Quelques commentaires informatiques sur un module important pour nous. Les fonctions mathématiques ne sont pas définies par défaut dans Python (à part $|x|$ et x^n), il faut faire appel à une librairie spéciale : le module `math` contient les fonctions mathématiques principales.

<code>abs(x)</code>	$ x $
<code>x ** n</code>	x^n
<code>sqrt(x)</code>	\sqrt{x}
<code>exp(x)</code>	e^x
<code>log(x)</code>	$\ln x$ logarithme népérien
<code>log(x,10)</code>	$\log x$ logarithme décimal
<code>cos(x), sin(x), tan(x)</code>	$\cos x, \sin x, \tan x$ en radians
<code>acos(x), asin(x), atan(x)</code>	$\arccos x, \arcsin x, \arctan x$ en radians
<code>floor(x)</code>	partie entière $E(x)$: plus grand entier $n \leq x$ (<code>floor</code> = plancher)
<code>ceil(x)</code>	plus petit entier $n \geq x$ (<code>ceil</code> = plafond)

- Comme on aura souvent besoin de ce module on l'appelle par le code `from math import *`. Cela signifie que l'on importe toutes les fonctions de ce module et qu'en plus on n'a pas besoin de préciser que la fonction vient du module `math`. On peut écrire `cos(3.14)` au lieu `math.cos(3.14)`.
- Dans l'algorithme précédent nous avions utilisé le logarithme décimal `log(x,10)`, ainsi que la partie entière `floor(x)`.

2.4. Écriture des nombres en base 2

On dispose d'une rampe de lumière, chacune des 8 lampes pouvant être allumée (rouge) ou éteinte (gris).



On numérote les lampes de 0 à 7. On souhaite contrôler cette rampe : afficher toutes les combinaisons possibles, faire défiler une combinaison de la gauche à droite (la “chenille”), inverser l'état de toutes les lampes,... Voyons comment l'écriture binaire des nombres peut nous aider. L'**écriture binaire** d'un nombre c'est son écriture en base 2.

Comment calculer un nombre qui est écrit en binaire ? Le chiffre des “dizaines” correspond à 2 (au lieu de 10), le chiffre des “centaines” à $4 = 2^2$ (au lieu de $100 = 10^2$), le chiffres des “milliers” à $8 = 2^3$ (au lieu de $1000 = 10^3$),... Pour le chiffre des unités cela correspond à $2^0 = 1$ (de même que $10^0 = 1$).

Par exemple 10011_b vaut le nombre 19. Car

$$10011_b = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 2 + 1 = 19.$$

De façon générale tout entier $N \in \mathbb{N}$ s'écrit de manière unique sous la forme

$$N = a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_2 2^2 + a_1 2 + a_0 \quad \text{et} \quad a_i \in \{0, 1\}$$

On note alors $N = a_n a_{n-1} \dots a_1 a_0$ (avec un indice b pour indiquer que c'est son écriture binaire).

Travaux pratiques 6.

1. Écrire une fonction qui à partir d'une liste $[a_0, a_1, \dots, a_n]$ calcule l'entier N correspondant à l'écriture binaire $a_n a_{n-1} \dots a_1 a_0$.
2. Écrire une fonction qui à partir de N calcule son écriture binaire sous la forme $[a_0, a_1, \dots, a_n]$.

La seule différence avec la base 10 c'est que l'on calcule avec des puissances de 2.

Code 10 (*binaire.py (1)*).

```
def binaire_vers_entier(tab):
    N = 0
    for i in range(len(tab)):
        N = N + tab[i] * (2 ** i)
    return N
```

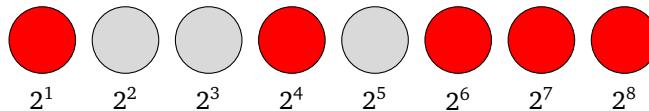
Idem pour le sens inverse où l'on a besoin du logarithme en base 2, qui vérifie $\log_2(2) = 1$ et $\log_2(2^i) = i$.

Code 11 (*binaire.py (2)*).

```
def entier_vers_binaire(N):
    tab = []
    n = floor(log(N,2)) # le nombre de chiffres est n+1
    for i in range(0,n+1):
        tab.append((N // 2 ** i) % 2)
    return tab
```

Maintenant appliquons ceci à notre problème de lampes. Si une lampe est allumée on lui attribut 1, et si elle est éteinte 0. Pour une rampe de 8 lampes on code $[a_0, a_1, \dots, a_7]$ l'état des lampes.

Par exemple la configuration suivante :



est codé $[1, 0, 0, 1, 0, 1, 1, 1]$ ce qui correspond au nombre binaire $11101001_b = 233$.

Travaux pratiques 7.

1. Faire une boucle qui affiche toutes les combinaisons possibles (pour une taille de rampe donnée).
2. Quelle opération mathématique élémentaire transforme un nombre binaire $a_n \dots a_1 a_0 \ b$ en $a_n \dots a_1 a_0 0 \ b$ (décalage vers la gauche et ajout d'un 0 à la fin) ?
3. Soit $N' = a_n a_{n-1} \dots a_1 a_0 0 \ b$ (une écriture avec $n+2$ chiffres). Quelle est l'écriture binaire de $N' \pmod{2^{n+1}}$? (C'est une écriture avec $n+1$ chiffres.)
4. En déduire un algorithme qui pour une configuration donnée de la rampe, fait permuter cycliquement (vers la droite) cette configuration. Par exemple $[1, 0, 1, 0, 1, 1, 1, 0]$ devient $[0, 1, 0, 1, 0, 1, 1, 1]$.
5. Quelle opération mathématique élémentaire permet de passer d'une configuration à son opposée (une lampe éteinte s'allume, et réciproquement). Par exemple si la configuration était $[1, 0, 1, 0, 1, 1, 1, 0]$ alors on veut $[0, 1, 0, 1, 0, 0, 0, 1]$. (Indication : sur cet exemple calculer les deux nombres correspondants et trouver la relation qui les lie.)

1. Il s'agit d'abord d'afficher les configurations. Par exemple si l'on a 4 lampes alors les configurations sont $[0, 0, 0, 0], [1, 0, 0, 0], [0, 1, 0, 0], [1, 1, 0, 0], \dots, [1, 1, 1, 1]$. Pour chaque lampe nous avons deux choix (allumé ou éteint), il y a $n+1$ lampes donc un total de 2^{n+1} configurations. Si l'on considère ces configurations comme des nombres écrits en binaire alors l'énumération ci-dessus correspond à compter $0, 1, 2, 3, \dots, 2^{n+1} - 1$.

D'où l'algorithme :

Code 12 (*binaire.py (3)*).

```
def configurations(n):
    for N in range(2**(n+1)):
        print(entier_vers_binaire_bis(N,n))
```

Où `entier_vers_binaire_bis(N,n)` est similaire à `entier_vers_binaire(N)`, mais en affichant aussi les zéros non significatifs, par exemple 7 en binaire s'écrit 111_b , mais codé sur 8 chiffres on ajoute devant des 0 non significatifs : 00000111_b .

2. En écriture décimale, multiplier par 10 revient à décaler le nombre initial et rajouter un zéro. Par exemple $10 \times 19 = 190$. C'est la même chose en binaire ! Multiplier un nombre par 2 revient sur l'écriture à un décalage vers la gauche et ajout d'un zéro sur le chiffre des unités. Exemple : $19 = 10011_b$ et $2 \times 19 = 38$ donc $2 \times 10011_b = 100110_b$.
3. Partant de $N = a_n a_{n-1} \dots a_1 a_0$ b . Notons $N' = 2N$, son écriture est $N' = a_n a_{n-1} \dots a_1 a_0 0$ b . Alors $N' \pmod{2^{n+1}}$ s'écrit exactement $a_{n-1} a_{n-2} \dots a_1 a_0 0$ b et on ajoute a_n qui est le quotient de N' par 2^{n+1} .
Preuve : $N' = a_n \cdot 2^{n+1} + a_{n-1} \cdot 2^n + \dots + a_0 \cdot 2$. Donc $N' \pmod{2^{n+1}} = a_{n-1} \cdot 2^n + \dots + a_0 \cdot 2$. Donc $N' \pmod{2^{n+1}} + a_n = a_{n-1} \cdot 2^n + \dots + a_0 \cdot 2 + a_n$.
4. Ainsi l'écriture en binaire de $N' \pmod{2^{n+1}} + a_n$ s'obtient comme permutation circulaire de celle de N . D'où l'algorithme :

Code 13 (*binaire.py (4)*).

```
def decalage(tab):
    N = binaire_vers_entier(tab)
    n = len(tab)-1 # le nombre de chiffres est n+1
    NN = 2*N % 2**(n+1) + 2*N // 2**(n+1)
    return entier_vers_binaire_bis(NN,n)
```

5. On remarque que si l'on a deux configurations opposées alors leur somme vaut $2^{n+1} - 1$: par exemple avec $[1, 0, 0, 1, 0, 1, 1, 1]$ et $[0, 1, 1, 0, 1, 0, 0, 0]$, les deux nombres associés sont $N = 11101001_b$ et $N' = 00010110_b$ (il s'agit juste de les réécrire de droite à gauche). La somme est $N + N' = 11101001_b + 00010110_b = 11111111_b = 2^8 - 1$. L'addition en écriture binaire se fait de la même façon qu'en écriture décimale et ici il n'y a pas de retenue. Si M est un nombre avec $n+1$ fois le chiffres 1 alors $M + 1 = 2^{n+1}$. Exemple si $M = 11111_b$ alors $M + 1 = 100000_b = 2^5$; ainsi $M = 2^5 - 1$. Donc l'opposé de N est $N' = 2^{n+1} - 1 - N$ (remarquez que dans $\mathbb{Z}/(2^{n+1} - 1)\mathbb{Z}$ alors $N' \equiv -N$).

Cela conduit à :

Code 14 (*binaire.py (5)*).

```
def inversion(tab):
    N = binaire_vers_entier(tab)
    n = len(tab)-1 # le nombre de chiffres est n+1
    NN = 2**n+1 - N
    return entier_vers_binaire_bis(NN,n)
```

-  **Mini-exercices.** 1. Pour un entier n fixé, combien y-a-t-il d'occurrences du chiffre 1 dans l'écriture des nombres de 1 à n ?
2. Écrire une fonction qui calcule l'écriture décimale d'un entier, sans recourir au log (une boucle `while` est la bienvenue).

3. Écrire un algorithme qui permute cycliquement une configuration de rampe vers la droite.
4. On dispose de $n + 1$ lampes, chaque lampe peut s'éclairer de trois couleurs : vert, orange, rouge (dans cet ordre). Trouver toutes les combinaisons possibles. Comment passer toutes les lampes à la couleur suivante ?
5. Générer toutes les matrices 4×4 n'ayant que des 0 et des 1 comme coefficients. On codera une matrice sous la forme de lignes $[[1, 1, 0, 1], [0, 0, 1, 0], [1, 1, 1, 1], [0, 1, 0, 1]]$.
6. On part du point $(0, 0) \in \mathbb{Z}^2$. A chaque pas on choisit au hasard une direction Nord, Sud, Est, Ouest. Si on va au Nord alors on ajoute $(0, 1)$ à sa position (pour Sud on ajoute $(0, -1)$; pour Est $(1, 0)$; pour Ouest $(-1, 0)$). Pour un chemin d'une longueur fixée de n pas, coder tous les chemins possibles. Caractériser les chemins qui repassent par l'origine. Calculer la probabilité p_n de repasser par l'origine. Que se passe-t-il lorsque $n \rightarrow +\infty$?
7. Écrire une fonction, qui pour un entier N , affiche son écriture en chiffres romains : $M = 1000, D = 500, C = 100, X = 10, V = 5, I = 1$. Il ne peut y avoir plus de trois symboles identiques à suivre.

3. Calculs de sinus, cosinus, tangente

Le but de cette section est le calcul des sinus, cosinus, et tangente d'un angle par nous même, avec une précision de 8 chiffres après la virgule.

3.1. Calcul de Arctan x

Nous aurons besoin de calculer une fois pour toute $\text{Arctan}(10^{-i})$, pour $i = 0, \dots, 8$, c'est-à-dire que l'on cherche les angles $\theta_i \in]-\frac{\pi}{2}, \frac{\pi}{2}[$ tels que $\tan \theta_i = 10^{-i}$. Nous allons utiliser la formule :

$$\text{Arctan } x = \sum_{k=0}^{+\infty} (-1)^k \frac{x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots$$

Travaux pratiques 8.

1. Calculer $\text{Arctan } 1$.
2. Calculer $\theta_i = \text{Arctan } 10^{-i}$ (avec 8 chiffres après la virgule) pour $i = 1, \dots, 8$.
3. Pour quelles valeurs de i , l'approximation $\text{Arctan } x \simeq x$ était-elle suffisante ?

Code 15 (*tangente.py* (1)).

```
def mon_arctan(x,n):
    somme = 0
    for k in range(0,n+1):
        if (k%2 == 0): # si k est pair signe +
            somme = somme + 1/(2*k+1) * (x ** (2*k+1))
        else:           # si k est impair signe -
            somme = somme - 1/(2*k+1) * (x ** (2*k+1))
    return somme
```

- La série qui permet de calculer $\text{Arctan } x$ est une somme infinie, mais si x est petit alors chacun des termes $(-1)^k \frac{x^{2k+1}}{2k+1}$ est très très petit dès que k devient grand. Par exemple si $0 \leq x \leq \frac{1}{10}$ alors $x^{2k+1} \leq \frac{1}{10^{2k+1}}$ et

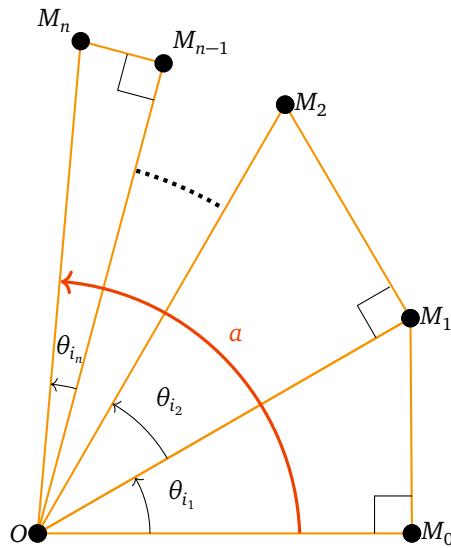
donc pour $k \geq 4$ nous aurons $\left|(-1)^k \frac{x^{2k+1}}{2k+1}\right| < 10^{-9}$. Chacun des termes suivants ne contribue pas aux 8 premiers chiffres après la virgule. Attention : il se pourrait cependant que la somme de beaucoup de termes finissent par y contribuer, mais ce n'est pas le cas ici (c'est un bon exercice de le prouver).

- Dans la pratique on calcule la somme à un certain ordre $2k+1$ jusqu'à ce que les 8 chiffres après la virgule ne bougent plus. Et en fait on s'aperçoit que l'on a seulement besoin d'utiliser $\text{Arctan } x \simeq x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7}$.
- Pour $i \geq 4$, $\text{Arctan } x \simeq x$ donne déjà 8 chiffres exacts après la virgule !

On remplit les valeurs des angles θ_i obtenus dans une liste nommée theta.

3.2. Calcul de $\tan x$

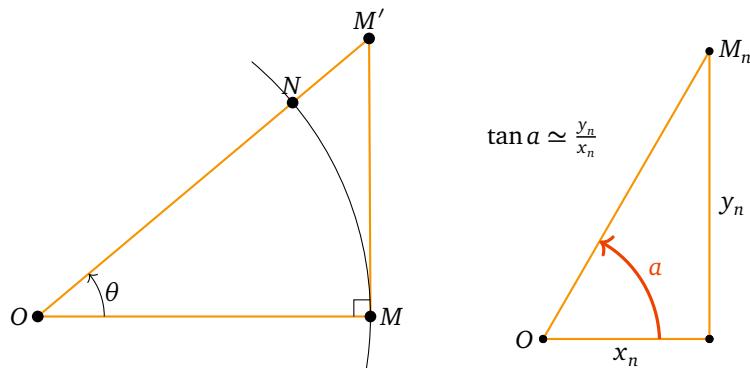
Le principe est le suivant : on connaît un certain nombre d'angles avec leur tangente : les angles θ_i (calculés ci-dessus) avec par définition $\tan \theta_i = 10^{-i}$. Fixons un angle $a \in [0, \frac{\pi}{2}]$. Partant du point $M_0 = (1, 0)$, nous allons construire des points M_1, M_2, \dots, M_n jusqu'à ce que M_n soit (à peu près) sur la demi-droite correspondant à l'angle a . Si M_n a pour coordonnées (x_n, y_n) alors $\tan a = \frac{y_n}{x_n}$. L'angle pour passer d'un point M_k à M_{k+1} est l'un des angles θ_i .



Rappelons que si l'on a un point $M(x, y)$ alors la rotation centrée à l'origine et d'angle θ envoie $M(x, y)$ sur le point $M'(x', y')$ avec

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{c'est-à-dire} \quad \begin{cases} x' = x \cos \theta - y \sin \theta \\ y' = x \sin \theta + y \cos \theta \end{cases}$$

Pour un point M , on note M' le point de la demi-droite $[ON)$ tel que les droites (OM) et (MM') soient perpendiculaires en M .



Travaux pratiques 9.

1. (a) Calculer la longueur OM' .
 (b) En déduire les coordonnées de M' .
 (c) Exprimez-les uniquement en fonction de x, y et $\tan \theta$.
2. Faire une boucle qui décompose l'angle a en somme d'angles θ_i (à une précision de 10^{-8} ; avec un minimum d'angles, les angles pouvant se répéter).
3. Partant de $M_0 = (1, 0)$ calculer les coordonnées des différents M_k , jusqu'au point $M_n(x_n, y_n)$ correspondant à l'approximation de l'angle a . Renvoyer la valeur $\frac{y_n}{x_n}$ comme approximation de $\tan a$.

Voici les préliminaires mathématiques :

- Dans le triangle rectangle OMM' on a $\cos \theta = \frac{OM}{OM'}$ donc $OM' = \frac{OM}{\cos \theta}$.
- D'autre part comme la rotation d'angle θ conserve les distances alors $OM = ON$. Si les coordonnées de M' sont (x'', y'') alors $x'' = \frac{1}{\cos \theta} x'$ et $y'' = \frac{1}{\cos \theta} y'$.
- Ainsi

$$\begin{cases} x'' = \frac{1}{\cos \theta} x' = \frac{1}{\cos \theta} (x \cos \theta - y \sin \theta) = x - y \tan \theta \\ y'' = \frac{1}{\cos \theta} y' = \frac{1}{\cos \theta} (x \sin \theta + y \cos \theta) = x \tan \theta + y \end{cases}$$

Autrement dit :

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} 1 & -\tan \theta \\ \tan \theta & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Voici une boucle simple pour décomposer l'angle θ : on commence par retirer le plus grand angle θ_0 autant de fois que l'on peut, lorsque ce n'est plus possible on passe à l'angle θ_1, \dots

Code 16 (*tangente.py (2)*).

```
i = 0
while (a > precision):      # boucle tant que la precision pas atteinte
    while (a < theta[i]):    # choix du bon angle theta_i à soustraire
        i = i+1
    a = a - theta[i]         # on retire l'angle theta_i et on recommence
```

Ici *precision* est la précision souhaité (pour nous 10^{-9}). Et le tableau *theta* contient les valeurs des angles θ_i .

Posons $x_0 = 1, y_0 = 0$ et $M_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$. Alors on définit par récurrence $M_{k+1} = P(\theta_i) \cdot M_k$ où $P(\theta) = \begin{pmatrix} 1 & -\tan \theta \\ \tan \theta & 1 \end{pmatrix}$. Les θ_i sont ceux apparaissant dans la décomposition de l'angle en somme de θ_i , donc on connaît $\tan \theta_i = 10^{-i}$. Ainsi si l'on passe d'un point M_k à M_{k+1} par un angle θ_i on a simplement :

$$\begin{cases} x_{k+1} = x_k - y_k \cdot 10^{-i} \\ y_{k+1} = x_k \cdot 10^{-i} + y_k \end{cases}$$

La valeur $\frac{y_n}{x_n}$ est la tangente de la somme des angles θ_i , donc une approximation de $\tan a$.

Le code est maintenant le suivant.

Code 17 (*tangente.py (3)*).

```
def ma_tan(a):
    precision = 10**(-9)
    i = 0 ; x = 1 ; y = 0
    while (a > precision):
```

```

while (a < theta[i]):
    i = i+1
    newa = a - theta[i]          # on retire l'angle theta_i
    newx = x - (10**(-i))*y    # on calcule le nouveau point
    newy = (10**(-i))*x + y
    x = newx
    y = newy
    a = newa
return y/x                      # on renvoie la tangente

```

Commentaires pour conclure :

- En théorie il ne faut pas confondre «précision» et «nombre de chiffres exacts après la virgule». Par exemple 0.999 est une valeur approchée de 1 à 10^{-3} près, mais aucun chiffre après la virgule n'est exact. Dans la pratique c'est la précision qui importe plus que le nombre de chiffres exacts.
- Notez à quel point les opérations du calcul de $\tan x$ sont simples : il n'y a quasiment que des additions à effectuer. Par exemple l'opération $x_{k+1} = x_k - y_k \cdot 10^{-i}$ peut être fait à la main : multiplier par 10^{-i} c'est juste décaler la virgule à droite de i chiffres, puis on additionne. C'est cet algorithme «CORDIC» qui est implémenté dans les calculatrices, car il nécessite très peu de ressources. Bien sûr, si les nombres sont codés en binaire on remplace les 10^{-i} par 2^{-i} pour n'avoir qu'à faire des décalages à droite.

3.3. Calcul de $\sin x$ et $\cos x$

Travaux pratiques 10.

Pour $0 \leq x \leq \frac{\pi}{2}$, calculer $\sin x$ et $\cos x$ en fonction de $\tan x$. En déduire comment calculer les sinus et cosinus de x .

Solution : On sait $\cos^2 x + \sin^2 x = 1$, donc en divisant par $\cos^2 x$ on trouve $1 + \tan^2 x = \frac{1}{\cos^2 x}$. On en déduit que pour $0 \leq x \leq \frac{\pi}{2}$ $\cos x = \frac{1}{\sqrt{1+\tan^2 x}}$. On trouve de même $\sin x = \frac{\tan x}{\sqrt{1+\tan^2 x}}$. Donc une fois que l'on a calculé $\tan x$ on en déduit $\sin x$ et $\cos x$ par un calcul de racine carrée. Attention c'est valide car x est compris entre 0 et $\frac{\pi}{2}$. Pour un x quelconque il faut se ramener par les formules trigonométriques à l'intervalle $[0, \frac{\pi}{2}]$.

- Mini-exercices.**
1. On dispose de billets de 1, 5, 20 et 100 euros. Trouvez la façon de payer une somme de n euros avec le minimum de billets.
 2. Faire un programme qui pour **n'importe quel** $x \in \mathbb{R}$, calcule $\sin x$, $\cos x$, $\tan x$.
 3. Pour $t = \tan \frac{x}{2}$ montrer que $\tan x = \frac{2t}{1-t^2}$. En déduire une fonction qui calcule $\tan x$. (Utiliser que pour x assez petit $\tan x \approx x$).
 4. Modifier l'algorithme de la tangente pour qu'il calcule aussi directement le sinus et le cosinus.

4. Les réels

Dans cette partie nous allons voir différentes façons de calculer la constante γ d'Euler. C'est un nombre assez mystérieux car personne ne sait si γ est un nombre rationnel ou irrationnel. Notre objectif est d'avoir le plus de décimales possibles après la virgule en un minimum d'étapes. Nous verrons ensuite comment les ordinateurs stockent les réels et les problèmes que cela engendre.

4.1. Constante γ d'Euler

Considérons la *suite harmonique* :

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

et définissons

$$u_n = H_n - \ln n.$$

Cette suite (u_n) admet une limite lorsque $n \rightarrow +\infty$: c'est la *constante γ d'Euler*.

Travaux pratiques 11.

1. Calculer les premières décimales de γ . Sachant que $u_n - \gamma \sim \frac{1}{2n}$, combien de décimales exactes peut-on espérer avoir obtenues ?
2. On considère $v_n = H_n - \ln(n + \frac{1}{2} + \frac{1}{24n})$. Sachant $v_n - \gamma \sim -\frac{1}{48n^3}$, calculer davantage de décimales.

Code 18 (*euler.py (1)*).

```
def euler1(n):
    somme = 0
    for i in range(n, 0, -1):
        somme = somme + 1/i
    return somme - log(n)
```

Code 19 (*euler.py (2)*).

```
def euler2(n):
    somme = 0
    for i in range(n, 0, -1):
        somme = somme + 1/i
    return somme - log(n+1/2+1/(24*n))
```

Vous remarquez que la somme est calculée à partir de la fin. Nous expliquerons pourquoi en fin de section.

4.2. 1000 décimales de la constante d'Euler

Il y a deux techniques pour obtenir plus de décimales : (i) pousser plus loin les itérations, mais pour avoir 1000 décimales de γ les méthodes précédentes sont insuffisantes ; (ii) trouver une méthode encore plus efficace. C'est ce que nous allons voir avec la méthode de Bessel modifiée.

Soit

$$w_n = \frac{A_n}{B_n} - \ln n \quad \text{avec} \quad A_n = \sum_{k=1}^{E(an)} \left(\frac{n^k}{k!} \right)^2 H_k \quad \text{et} \quad B_n = \sum_{k=0}^{E(an)} \left(\frac{n^k}{k!} \right)^2$$

où $\alpha = 3.59112147\dots$ est la solution de $\alpha(\ln \alpha - 1) = 1$ et $E(x)$ désigne la partie entière. Alors

$$|w_n - \gamma| \leq \frac{C}{e^{4n}}$$

où C est une constante (non connue).

Travaux pratiques 12.

1. Programmer cette méthode.
2. Combien d'itérations faut-il pour obtenir 1000 décimales ?
3. Utiliser le module `decimal` pour les calculer.

Voici le code :

Code 20 (*euler.py (3)*).

```
def euler3(n):
    alpha = 3.59112147
    N = floor(alpha*n)      # Borne des sommes
    A = 0 ; B = 0
    H = 0
    for k in range(1,N+1):
        c = ( (n**k)/factorial(k) ) ** 2    # Coefficient commun
        H = H + 1/k                          # Somme harmonique
        A = A + c*H
        B = B + c
    return A/B - log(n)
```

Pour obtenir N décimales il faut résoudre l'inéquation $\frac{C}{e^{4n}} \leq \frac{1}{10^N}$. On «passe au log» pour obtenir $n \geq \frac{N \ln(10) + \ln(C)}{4}$. On ne connaît pas C mais ce n'est pas si important. Moralement pour une itération de plus on obtient (à peu près) une décimale de plus (c'est-à-dire un facteur 10 sur la précision !). Pour $n \geq 800$ on obtient 1000 décimales exactes de la constante d'Euler :

```
0,
57721566490153286060651209008240243104215933593992 35988057672348848677267776646709369470632917467495
14631447249807082480960504014486542836224173997644 92353625350033374293733773767394279259525824709491
60087352039481656708532331517766115286211995015079 84793745085705740029921354786146694029604325421519
0587755326733139925401296742051375413954911168510 28079842348775872050384310939973613725530608893312
67600172479537836759271351577226102734929139407984 30103417771778088154957066107501016191663340152278
93586796549725203621287922655595366962817638879272 68013243101047650596370394739495763890657296792960
10090151251959509222435014093498712282479497471956 46976318506676129063811051824197444867836380861749
45516989279230187739107294578155431600500218284409 60537724342032854783670151773943987003023703395183
28690001558193988042707411542227819716523011073565 83396734871765049194181230004065469314299929777956
93031005030863034185698032310836916400258929708909 85486825777364288253954925873629596133298574739302
```

Pour obtenir plus de décimales que la précision standard de Python, il faut utiliser le module `decimal` qui permet de travailler avec une précision arbitraire fixée.

4.3. Un peu de réalité

En mathématique un réel est un élément de \mathbb{R} et son écriture décimale est souvent infinie après la virgule : par exemple $\pi = 3,14159265\dots$ Mais bien sûr un ordinateur ne peut pas coder une infinité d'informations. Ce qui se rapproche d'un réel est un **nombre flottant** dont l'écriture est :

$$\underbrace{\pm 1,234567890123456789}_{\text{mantisse}} e \underbrace{\pm 123}_{\text{exposant}}$$

pour $\pm 1,234\dots \times 10^{\pm 123}$. La **mantisso** est un nombre décimal (positif ou négatif) appartenant à $[1, 10[$ et l'exposant est un entier (lui aussi positif ou négatif). En Python la mantisse à une précision de 16 chiffres après la virgule.

Cette réalité informatique fait que des erreurs de calculs peuvent apparaître même avec des opérations simples. Pour voir un exemple de problème faites ceci :

Travaux pratiques 13.

Poser $x = 10^{-16}$, $y = x + 1$, $z = y - 1$. Que vaut z pour Python ?

Comme Python est très précis nous allons faire une routine qui permet de limiter drastiquement le nombre de chiffres et mettre en évidence les erreurs de calculs.

Travaux pratiques 14.

1. Calculer l'exposant d'un nombre réel. Calculer la mantisse.
2. Faire une fonction qui ne conserve que 6 chiffres d'un nombre (6 chiffres en tout : avant + après la virgule, exemple 123,456789 devient 123,456).

Voici le code :

Code 21 (*reels.py (1)*).

```
precision = 6                      # Nombre de décimales conservées
def tronquer(x):
    n = floor(log(x,10))           # Exposant
    m = floor( x * 10 ** (precision-1 - n)) # Mantisse
    return m * 10 ** (-precision+1+n)      # Nombre tronqué
```

Comme on l'a déjà vu auparavant l'exposant se récupère à l'aide du logarithme en base 10. Et pour tronquer un nombre avec 6 chiffres, on commence par le décaler vers la gauche pour obtenir 6 chiffres avant la virgule (123,456789 devient 123456,789) il ne reste plus qu'à prendre la partie entière (123456) et le redécaler vers la droite (pour obtenir 123,456).

Absorption

Travaux pratiques 15.

1. Calculer `tronquer(1234.56 + 0.007)`.
2. Expliquer.

Chacun des nombres 1234,56 et 0,007 est bien un nombre s'écrivant avec moins de 6 décimales mais leur somme 1234,567 a besoin d'une décimale de plus, l'ordinateur ne retient pas la 7-ème décimale et ainsi le résultat obtenu est 1234,56. Le 0,007 n'apparaît pas dans le résultat : il a été victime d'une **absorption**.

Élimination

Travaux pratiques 16.

1. Soient $x = 1234,8777$, $y = 1212,2222$. Calculer $x - y$ à la main. Comment se calcule la différence $x - y$ avec notre précision de 6 chiffres ?
2. Expliquer la différence.

Comme $x - y = 22,6555$ qui n'a que 6 chiffres alors on peut penser que l'ordinateur va obtenir ce résultat. Il n'en est rien, l'ordinateur ne stocke pas x mais `tronquer(x)`, idem pour y . Donc l'ordinateur effectue en fait le calcul suivant : `tronquer(tronquer(x) - tronquer(y))`, il calcule donc $1234,87 - 1212,22 = 22,65$. Quel est le problème ? C'est qu'ensuite l'utilisateur considère –à tort– que le résultat est calculé avec une précision de 6 chiffres. Donc on peut penser que le résultat est $22,6500$ mais les 2 derniers chiffres sont une pure invention.

C'est un phénomène d'**élimination**. Lorsque l'on calcule la différence de deux nombres proches, le résultat a en fait une précision moindre. Cela peut être encore plus dramatique avec l'exemple $\delta = 1234,569 - 1234,55$ la différence est $0,01900$ alors que l'ordinateur retournera $0,01000$. Il y a presque un facteur deux, et on aura des problèmes si l'on a besoin de diviser par δ .

Signalons au passage une erreur d'interprétation fréquente : ne pas confondre la **précision** d'affichage (exemple : on calcule avec 10 chiffres après la virgule) avec l'**exactitude** du résultat (combien de décimales sont vraiment exactes ?).

Conversion binaire – décimale

Enfin le problème le plus troublant est que les nombres flottants sont stockés en écriture binaire et pas en écriture décimale.

Travaux pratiques 17.

Effectuer les commandes suivantes et constater !

1. `sum = 0` puis `for i in range(10): sum = sum + 0.1`. Que vaut `sum` ?
2. `0.1 + 0.1 == 0.2` et `0.1 + 0.1 + 0.1 == 0.3`
3. `x = 0.2 ; print("0.2\u00b7en\u00b7Python\u00b7%.25f" %x)`

La raison est simple mais néanmoins troublante. L'ordinateur ne stocke pas $0,1$, ni $0,2$ en mémoire mais le nombre en écriture binaire qui s'en rapproche le plus.

En écriture décimale, il est impossible de coder $1/3 = 0,3333\dots$ avec un nombre fini de chiffres après la virgule. Il en va de même ici : l'ordinateur ne peut pas stocker exactement $0,2$. Il stocke un nombre en écriture binaire qui s'en rapproche le plus ; lorsqu'on lui demande d'afficher le nombre stocké, il retourne l'écriture décimale qui se rapproche le plus du nombre stocké, mais ce n'est plus $0,2$, mais un nombre très très proche :

`0.2000000000000000111022302...`

4.4. Somme des inverses des carrés

Voyons une situation concrète où ces problèmes apparaissent.

Travaux pratiques 18.

1. Faire une fonction qui calcule la somme $S_n = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{n^2}$.
2. Faire une fonction qui calcule cette somme mais en utilisant seulement une écriture décimale à 6 chiffres (à l'aide de la fonction `tronquer()` vue au-dessus).
3. Reprendre cette dernière fonction, mais en commençant la somme par les plus petits termes.
4. Comparez le deux dernières méthodes, justifier et conclure.

La première fonction ne pose aucun problème et utilise toute la précision de Python.

Dans la seconde on doit, à chaque calcul, limiter notre précision à 6 chiffres (ici 1 avant la virgule et 5 après).

Code 22 (*reels.py (2)*).

```
def somme_inverse_carres_tronq(n):
    somme = 0
    for i in range(1,n+1):
        somme = tronquer(somme + tronquer(1/(i*i)))
    return somme
```

Il est préférable de commencer la somme par la fin :

Code 23 (*reels.py (3)*).

```
def somme_inverse_carres_tronq_inv(n):
    somme = 0
    for i in range(n,0,-1):
        somme = tronquer(somme + tronquer(1/(i*i)))
    return somme
```

Par exemple pour $n = 100\ 000$ l'algorithme `somme_inverse_carres_tronq()` (avec écriture tronquée, sommé dans l'ordre) retourne 1,64038 alors que l'algorithme `somme_inverse_carres_tronq_inv()` (avec la somme dans l'ordre inverse) on obtient 1,64490. Avec une précision maximale et n très grand on doit obtenir 1,64493... (en fait c'est $\frac{\pi^2}{6}$).

Notez que faire grandir n pour l'algorithme `somme_inverse_carres_tronq()` n'y changera rien, il bloque à 2 décimales exactes après la virgule : 1,64038 ! La raison est un phénomène d'absorption : on rajoute des termes très petits devant une somme qui vaut plus de 1. Alors que si l'on part des termes petits, on ajoute des termes petits à une somme petite, on garde donc un maximum de décimales valides avant de terminer par les plus hautes valeurs.

- Mini-exercices.**
1. Écrire une fonction qui approxime la constante α qui vérifie $\alpha(\ln \alpha - 1) = 1$. Pour cela poser $f(x) = x(\ln x - 1) - 1$ et appliquer la méthode de Newton : fixer u_0 (par exemple ici $u_0 = 4$) et $u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$.
 2. Pour chacune des trois méthodes, calculer le nombre approximatif d'itérations nécessaires pour obtenir 100 décimales de la constante γ d'Euler.
 3. Notons $C_n = \frac{1}{4n} \sum_{k=0}^{2n} \frac{[(2k)!]^3}{(k!)^4 (16n)^{2k}}$. La formule de Brent-McMillan affirme $\gamma = \frac{A_n}{B_n} - \frac{C_n}{B_n^2} - \ln n + O\left(\frac{1}{e^{8n}}\right)$ où cette fois les sommes pour A_n et B_n vont jusqu'à $E(\beta n)$ avec $\beta = 4,970625759\dots$ la solution de $\beta(\ln \beta - 1) = 3$. La notation $O\left(\frac{1}{e^{8n}}\right)$ indique que l'erreur est $\leq \frac{C}{e^{8n}}$ pour une certaine constante C . Mettre en œuvre cette formule. En 1999 cette formule a permis de calculer 100 millions de décimales. Combien a-t-il fallu d'itérations ?
 4. Faire une fonction qui renvoie le terme u_n de la suite définie par $u_0 = \frac{1}{3}$ et $u_{n+1} = 4u_n - 1$. Que vaut u_{100} ? Faire l'étude mathématique et commenter.

5. Arithmétique – Algorithmes récursifs

Nous allons présenter quelques algorithmes élémentaires en lien avec l'arithmétique. Nous en profitons pour présenter une façon complètement différente d'écrire des algorithmes : les fonctions récursives.

5.1. Algorithmes récursifs

Voici un algorithme très classique :

Code 24 (*recursif.py (1)*).

```
def factorielle_classique(n):
    produit = 1
    for i in range(1,n+1):
        produit = i * produit
    return produit
```

Voyons comment fonctionne cette boucle. On initialise la variable *produit* à 1, on fait varier un indice *i* de 1 à *n*. À chaque étape on multiplie *produit* par *i* et on affecte le résultat dans *produit*. Par exemple si *n* = 5 alors la variable *produit* s'initialise à 1, puis lorsque *i* varie la variable *produit* devient $1 \times 1 = 1$, $2 \times 1 = 2$, $3 \times 2 = 6$, $4 \times 6 = 24$, $5 \times 24 = 120$. Vous avez bien sûr reconnus le calcul de 5!

Étudions un autre algorithme.

Code 25 (*recursif.py (2)*).

```
def factorielle(n):
    if (n==1):
        return 1
    else:
        return n * factorielle(n-1)
```

Que fait cet algorithme ? Voyons cela pour *n* = 5. Pour *n* = 5 la condition du «si» (if) n'est pas vérifiée donc on passe directement au «sinon» (else). Donc *factorielle*(5) renvoie comme résultat : $5 * \text{factorielle}(4)$. On a plus ou moins progressé : le calcul n'est pas fini car on ne connaît pas encore *factorielle*(4) mais on s'est ramené à un calcul au rang précédent, et on itère :

factorielle(5) = $5 * \text{factorielle}(4)$ = $5 * 4 * \text{factorielle}(3)$ = $5 * 4 * 3 * \text{factorielle}(2)$ et enfin *factorielle*(5) = $5 * 4 * 3 * 2 * \text{factorielle}(1)$. Pour *factorielle*(1) la condition du if (*n==1*) est vérifiée et alors *factorielle*(1)=1. Le bilan est donc que *factorielle*(5) = $5 * 4 * 3 * 2 * 1$ c'est bien 5 !

Une fonction qui lorsque elle s'exécute s'appelle elle-même est une **fonction récursive**. Il y a une analogie très forte avec la récurrence. Par exemple on peut définir la suite des factorielles ainsi :

$$u_1 = 1 \quad \text{et} \quad u_n = n \times u_{n-1} \text{ si } n \geq 2.$$

Nous avons ici $u_n = n!$ pour tout $n \geq 1$.

Comme pour la récurrence une fonction récursive comporte une étape d'**initialisation** (ici if (*n==1*): *return 1*) correspondant à $u_1 = 1$) et une étape d'**hérité** (ici *return n * factorielle(n-1)* correspondant à $u_n = n \times u_{n-1}$).

On peut même faire deux appels à la fonction :

Code 26 (*recursif.py (3)*).

```
def fibonacci(n):
    if (n==0) or (n==1):
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Faites-le calcul de `fibonacci(5)`. Voici la version mathématique des nombres de Fibonacci.

$$F_0 = 1, F_1 = 1 \quad \text{et} \quad F_n = F_{n-1} + F_{n-2} \quad \text{si } n \geq 2.$$

On obtient un nombre en additionnant les deux nombres des rangs précédents :

1 1 2 3 5 8 13 21 34 ...

5.2. L'algorithme d'Euclide

L'algorithme d'Euclide est basé sur le principe suivant

$$\text{si } b|a \text{ alors } \text{pgcd}(a, b) = b \quad \text{sinon } \text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b)$$

Travaux pratiques 19.

1. Créer une fonction récursive `pgcd(a, b)` qui calcule le pgcd.
2. On note p_n la probabilité que deux entiers a, b tirés au hasard dans $1, 2, \dots, n$ soient premiers entre eux. Faire une fonction qui approxime p_n . Lorsque n devient grand, comparer p_n et $\frac{6}{\pi^2}$.

Voici le code pour l'algorithme d'Euclide récursif. Notez à quel point le code est succinct et épuré !

Code 27 (*arith.py (1)*).

```
def pgcd(a,b):
    if a%b == 0:
        return b
    else:
        return pgcd(b, a%b)
```

Deux entiers a, b sont premiers entre eux ssi $\text{pgcd}(a, b) = 1$, donc voici l'algorithme :

Code 28 (*arith.py (2)*).

```
def nb_premiers_entre_eux(n,nbtirages):
    i = 1
    nbpremiers = 0
    while i <= nbtirages:
        i = i+1
        a = random.randint(1,n)
        b = random.randint(1,n)
        if pgcd(a,b)==1:
            nbpremiers = nbpremiers + 1
    return nbpremiers
```

On tire au hasard deux entiers a et b entre 1 et n et on effectue cette opération `nbtirages` fois. Par exemple entre 1 et 1000 si l'on effectue 10000 tirage on trouve une probabilité mesurée par `nbpremiers/nbtirages` de 0,60... (les décimales d'après dépendent des tirages).

Lorsque n tend vers $+\infty$ alors $p_n \rightarrow \frac{6}{\pi^2} = 0.607927\dots$ et on dit souvent que : «la probabilité que deux entiers tirés au hasard soient premiers entre eux est $\frac{6}{\pi^2}$ ».

Commentaires sur les algorithmes récursifs :

- Les algorithmes récursifs ont souvent un code très court, et proche de la formulation mathématique lorsque l'on a une relation de récurrence.
- Selon le langage ou la fonction programmée il peut y avoir des problèmes de mémoire (si par exemple pour calculer $5!$ l'ordinateur a besoin de stocker $4!$ pour lequel il a besoin de stocker $3!...$).
- Il est important de bien réfléchir à la condition initiale (qui est en fait celle qui termine l'algorithme) et à la récurrence sous peine d'avoir une fonction qui boucle indéfiniment !
- Il n'existe pas des algorithmes récursifs pour tout (voir par exemple les nombres premiers) mais ils apparaissent beaucoup dans les algorithmes de tris. Autre exemple : la dichotomie se programme très bien par une fonction récursive.

5.3. Nombres premiers

Les nombres premiers offrent peu de place aux algorithmes récursifs car il n'y a pas de lien de récurrence entre les nombres premiers.

Travaux pratiques 20.

1. Écrire une fonction qui détecte si un nombre n est premier ou pas en testant s'il existe des entiers k qui divise n . (On se limitera aux entiers $2 \leq k \leq \sqrt{n}$, pourquoi?).
2. Faire un algorithme pour le crible d'Eratosthène : écrire tous les entiers de 2 à n , conserver 2 (qui est premier) et barrer tous les multiples suivants de 2. Le premier nombre non barré (c'est 3) est premier. Barrer tous les multiples suivants de 3,...
3. Dessiner la spirale d'Ulam : on place les nombres entiers en spirale, et on colorie en rouge les nombres premiers.

... ...
5 4 3 :
6 1 2 11
7 8 9 10

1. Si n n'est pas premier alors $n = a \times b$ avec $a, b \geq 2$. Il est clair que soit $a \leq \sqrt{n}$ ou bien $b \leq \sqrt{n}$ (sinon $n = a \times b > n$). Donc il suffit de tester les diviseurs $2 \leq k \leq \sqrt{n}$. D'où l'algorithme :

Code 29 (`arith.py (3)`).

```
def est_premier(n):
    if (n<=1): return False
    k = 2
    while k*k <= n:
        if n%k==0:
            return False
        else:
            k = k +1
    return True
```

Notez qu'il vaut mieux écrire la condition $k*k \leq n$ plutôt que $k \leq \sqrt{n}$: il est beaucoup plus rapide de calculer le carré d'un entier plutôt qu'extraire une racine carrée.

Nous avons utilisé un nouveau type de variable : un **booléen** est une variable qui ne peut prendre que deux états Vrai ou Faux (ici *True* or *False*, souvent codé 1 et 0). Ainsi `est_premier(13)` renvoie *True*, alors que `est_premier(14)` renvoie *False*.

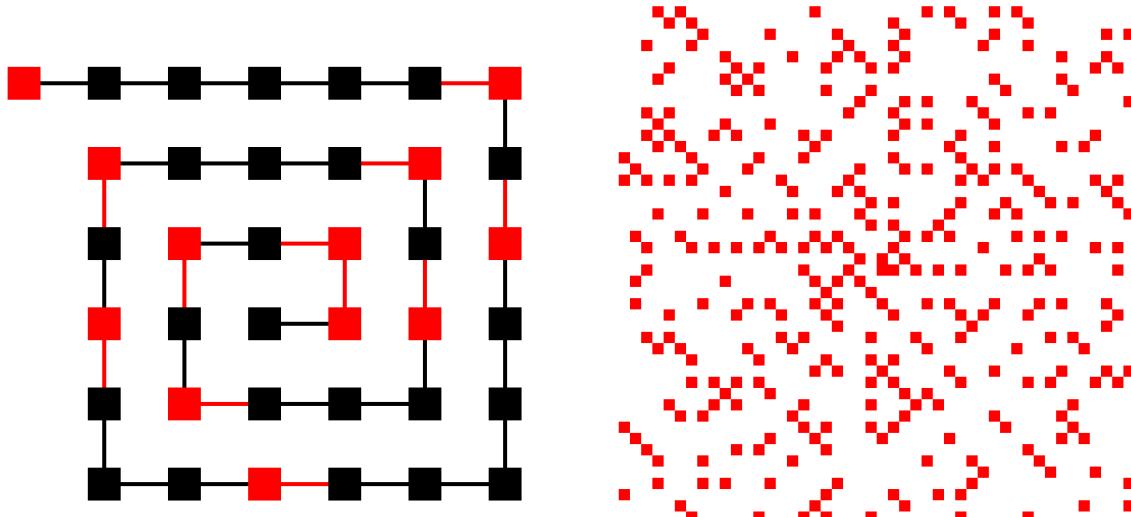
2. Pour le crible d'Eratosthène le plus dur est de trouver le bon codage de l'information.

Code 30 (*arith.py* (4)).

```
def eratosthene(n):
    liste_entiers = list(range(n+1)) # tous les entiers
    liste_entiers[1] = 0             # 1 n'est pas premier
    k = 2                           # on commence par les multiples de 2
    while k*k <= n:
        if liste_entiers[k] != 0: # si le nombre k n'est pas barré
            i = k                 # les i sont les multiples de k
            while i <= n-k:
                i = i+k
                liste_entiers[i] = 0 # multiples de k : pas premiers
        k = k +1
    liste_premiers = [k for k in liste_entiers if k !=0] # efface les 0
    return liste_premiers
```

Ici on commence par faire un tableau contenant les entiers $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, \dots]$. Pour signifier qu'un nombre n'est pas premier on remplace l'entier par 0. Comme 1 n'est pas un nombre premier : on le remplace par 0. Puis on fait une boucle, on part de 2 et on remplace tous les autres multiples de 2 par 0 : la liste est maintenant : $[0, 0, 2, 3, 0, 5, 0, 7, 0, 9, 0, 11, 0, 13, \dots]$. Le premiers nombre après 2 est 3 c'est donc un nombre premier. (car s'il n'a aucun diviseur autre que 1 et lui-même car sinon il aurait été rayé). On garde 3 et remplace tous les autres multiples de 3 par 0. La liste est maintenant : $[0, 0, 2, 3, 0, 5, 0, 7, 0, 0, 0, 11, 0, 13, \dots]$. On itère ainsi, à la fin on efface les zéros pour obtenir : $[2, 3, 5, 7, 11, 13, \dots]$.

3. Pour la spirale d'Ulam la seule difficulté est de placer les entiers sur une spirale, voici le résultat.



À gauche le début de la spirale (de $n = 1$ à 37) en rouge les nombres premiers (en noir les nombres non premiers) ; à droite le motif obtenu jusqu'à de grandes valeurs (en blanc les nombres non premiers).

- Mini-exercices.** 1. Écrire une version itérative et une version récursive pour les fonctions suivantes : (a) la somme des carrés des entiers de 1 à n ; (b) 2^n (sans utiliser d'exposant); (c) la partie entière d'un réel $x \geqslant 0$; (d) le quotient de la division euclidienne de a par b (avec $a \in \mathbb{N}$, $b \in \mathbb{N}^*$); (e) le reste de cette division euclidienne (sans utiliser les commandes % ni //).
2. Écrire une version itérative de la suite de Fibonacci.
3. Écrire une version itérative de l'algorithme d'Euclide. Faire une version qui calcule les coefficients de Bézout.
4. Écrire une fonction itérative, puis récursive, qui pour un entier n renvoie la liste de ses diviseurs. Dessiner une spirale d'Ulam, dont l'intensité de la couleur dépend du nombre de diviseurs.
5. Une suite de Syracuse est définie ainsi : partant d'un entier s si s est pair on le divise par deux, s'il est impair on le multiplie par 3 et on ajoute 1. On itère ce processus. Quelle conjecture peut-on faire sur cette suite ?
6. Dessiner le triangle de Pascal $\begin{array}{cccccc} & & & 1 \\ & & & 1 & 1 \\ & & & \dots & 2 & 1 \\ & & & & 1 & 1 \end{array}$. Ensuite effacer tous les coefficients pairs (ou mieux : remplacer les coefficients pairs par un carré blanc et les coefficients impairs par un carré rouge). Quelle figure reconnaissiez-vous ?

6. Polynômes – Complexité d'un algorithme

Nous allons étudier la complexité des algorithmes à travers l'exemple des polynômes.

6.1. Qu'est-ce qu'un algorithme ?

Qu'est ce qu'un algorithme ? Un algorithme est une succession d'instructions qui renvoie un résultat. Pour être vraiment un algorithme on doit justifier que le résultat renvoyé est **exact** (le programme fait bien ce que l'on souhaite) et ceci en un **nombre fini d'étapes** (cela renvoie le résultat en temps fini).

Maintenant certains algorithmes peuvent être plus rapides que d'autres. C'est souvent le temps de calcul qui est le principal critère, mais cela dépend du langage et de la machine utilisée. Il existe une manière plus mathématique de faire : la **complexité** d'un algorithme c'est le nombre d'opérations élémentaires à effectuer.

Ces opérations peuvent être le nombre d'opérations au niveau du processeur, mais pour nous ce sera le nombre d'additions +, le nombre de multiplications × à effectuer. Pour certains algorithmes la vitesse d'exécution n'est pas le seul paramètre mais aussi la taille de la mémoire occupée.

6.2. Polynômes

Travaux pratiques 21.

On code un polynôme $a_0 + a_1X + \cdots + a_nX^n$ sous la forme d'une liste $[a_0, a_1, \dots, a_n]$.

1. Écrire une fonction correspondant à la somme de deux polynômes. Calculer la complexité de cet algorithme (en terme du nombre d'additions sur les coefficients, en fonctions du degré des polynômes).
2. Écrire une fonction correspondant au produit de deux polynômes. Calculer la complexité de cet algorithme (en terme du nombre d'additions et de multiplications sur les coefficients).
3. Écrire une fonction correspondant au quotient et au reste de la division euclidienne de A par B où B est un polynôme unitaire (son coefficient de plus haut degré est 1). Majorer la complexité de cet algorithme (en terme du nombre d'additions et de multiplications sur les coefficients).

- La seule difficulté est de gérer les indices, en particulier on ne peut appeler un élément d'une liste en dehors des indices où elle est définie. Une bonne idée consiste à commencer par définir une fonction `degre(poly)`, qui renvoie le degré du polynôme (attention au 0 non significatifs).

Voici le code dans le cas simple où $\deg A = \deg B$:

Code 31 (*polynome.py (1)*).

```
def somme(A,B):          # si deg(A)=deg(B)
    C = []
    for i in range(0,degre(A)+1):
        s = A[i]+B[i]
        C.append(s)
```

Calculons sa complexité, on suppose $\deg A \leq n$ et $\deg B \leq n$: il faut faire l'addition des coefficients $a_i + b_i$, pour i variant de 0 à n : donc la complexité est de $n + 1$ additions (dans \mathbb{Z} ou \mathbb{R}).

- Pour le produit il faut se rappeler que si $A(X) = \sum_{i=0}^m a_i X^i$, $B(X) = \sum_{j=0}^n b_j X^j$ et $C = A \times B = \sum_{k=0}^{m+n} c_k X^k$ alors le k -ème coefficient de C est $c_k = \sum_{i+j=k} a_i \times b_j$. Dans la pratique on fait attention de ne pas accéder à des coefficients qui n'ont pas été définis.

Code 32 (*polynome.py (2)*).

```
def produit(A,B):
    C = []
    for k in range(degre(A)+degre(B)+1):
        s = 0
        for i in range(k+1):
            if (i <= degre(A)) and (k-i <= degre(B)):
                s = s + A[i]*B[k-i]
        C.append(s)
    return C
```

Pour la complexité on commence par compter le nombre de multiplications (dans \mathbb{Z} ou \mathbb{R}). Notons $m = \deg A$ et $n = \deg B$. Alors il faut multiplier les $m + 1$ coefficients de A par les $n + 1$ coefficients de B : il y a donc $(m + 1)(n + 1)$ multiplications.

Comptons maintenant les additions : les coefficients de $A \times B$ sont : $c_0 = a_0 b_0$, $c_1 = a_0 b_1 + a_1 b_0$, $c_2 = a_2 b_0 + a_1 b_1 + a_2 b_0, \dots$

Nous utilisons l'astuce suivante : nous savons que le produit $A \times B$ est de degré $m + n$ donc a (au plus) $m + n + 1$ coefficients. Partant de $(m + 1)(n + 1)$ produits, chaque addition regroupe deux termes, et nous devons arriver à $m + n + 1$ coefficients. Il y a donc $(m + 1)(n + 1) - (m + n + 1) = mn$ additions.

- Pour la division euclidienne, le principe est de poser une division de polynôme. Par exemple pour $A = 2X^4 - X^3 - 2X^2 + 3X - 1$ et $B = X^2 - X + 1$.

$$\begin{array}{r}
 & 2X^4 - X^3 - 2X^2 + 3X - 1 \\
 - & 2X^4 - 2X^3 + 2X^2 \\
 \hline
 & X^3 - 4X^2 + 3X - 1 \\
 - & X^3 - X^2 + X \\
 \hline
 & -3X^2 + 2X - 1 \\
 - & -3X^2 + 3X - 3 \\
 \hline
 & -X + 2
 \end{array}
 \quad
 \begin{array}{r}
 X^2 - X + 1 \\
 \hline
 2X^2 + X - 3
 \end{array}$$

Alors on cherche quel monôme P_1 fait diminuer le degré de $A - P_1B$, c'est $2X^2$ (le coefficient 2 est le coefficient dominant de A). On pose ensuite $R_1 = A - P_1B = X^3 - 4X^2 + 3X - 1$, $Q_1 = 2X^2$, on recommence avec R_1 divisé par B , $R_2 = R_1 - P_2B$ avec $P_2 = X$, $Q_2 = Q_1 + P_2, \dots$. On arrête lorsque $\deg R_i < \deg B$.

Code 33 (*polynome.py* (3)).

```
def division(A,B):
    Q = [0]      # Quotient
    R = A        # Reste
    while (degre(R) >= degre(B)):
        P = monome(R[degre(R)],degre(R)-degre(B))
        R = somme(R,produit(-P,B))
        Q = somme(Q,P)
    return Q,R
```

C'est une version un peu simplifiée du code : où $P = r_n X^{\deg R - \deg B}$ et où il faut remplacer $-P$ par $[-a_0, -a_1, \dots]$. Si $A, B \in \mathbb{Z}[X]$ alors le fait que B soit unitaire implique que Q et R sont aussi à coefficients entiers.

Quelle est la complexité de la division euclidienne ? À chaque étape on effectue une multiplication de polynômes ($P_i \times B$) puis une addition de polynôme ($R_i - P_iB$) ; à chaque étape le degré de R_i diminue (au moins) de 1. Donc il y a au plus $\deg A - \deg B + 1$ étapes.

Mais dans la pratique c'est plus simple que cela. La multiplication $P_i \times B$ est très simple : car P_i est un monôme $P_i = p_i X^i$. Multiplier par X^i c'est juste un décalage d'indice (comme multiplier par 10^i en écriture décimale) c'est donc une opération négligeable. Il reste donc à multiplier les coefficients de B par p_i : il y a donc $\deg B + 1$ multiplications de coefficients. La soustraction aussi est assez simple on retire à R_i un multiple de B , donc on a au plus $\deg B + 1$ coefficients à soustraire : il y a à chaque étape $\deg B + 1$ additions de coefficients.

Bilan : si $m = \deg A$ et $n = \deg B$ alors la division euclidienne s'effectue en au plus $(m - n + 1)(m + 1)$ multiplications et le même nombre d'additions (dans \mathbb{Z} ou \mathbb{R}).

6.3. Algorithme de Karatsuba

Pour diminuer la complexité de la multiplication de polynômes, on va utiliser un paradigme très classique de programmation : « diviser pour régner ». Pour cela, on va décomposer les polynômes à multiplier P et Q de degrés strictement inférieurs à $2n$ en

$$P = P_1 + P_2 \cdot X^n \quad \text{et} \quad Q = Q_1 + Q_2 \cdot X^n$$

avec les degrés de P_1, P_2, Q_1 et Q_2 strictement inférieurs à n .

Travaux pratiques 22.

1. Écrire une formule qui réduit la multiplication des polynômes P et Q de degrés strictement inférieurs à $2n$ en multiplications de polynômes de degrés strictement inférieurs à n .
2. Programmer un algorithme récursif de multiplication qui utilise la formule précédente. Quelle est sa complexité ?
3. On peut raffiner cette méthode avec la remarque suivante de Karatsuba : le terme intermédiaire de $P \cdot Q$ s'écrit

$$P_1 \cdot Q_2 + P_2 \cdot Q_1 = (P_1 + P_2) \cdot (Q_1 + Q_2) - P_1 Q_1 - P_2 Q_2$$

Comme on a déjà calculé $P_1 Q_1$ et $P_2 Q_2$, on échange deux multiplications et une addition (à gauche) contre une multiplication et quatre additions (à droite). Écrire une fonction qui réalise la multiplication de polynômes à la Karatsuba.

4. Trouver la formule de récurrence qui définit la complexité de la multiplication de Karatsuba. Quelle est sa solution ?

1. Il suffit de développer le produit $(P_1 + X^n P_2) \cdot (Q_1 + X^n Q_2)$:

$$(P_1 + X^n P_2) \cdot (Q_1 + X^n Q_2) = P_1 Q_1 + X^n \cdot (P_1 Q_2 + P_2 Q_1) + X^{2n} \cdot P_2 Q_2$$

On se ramène ainsi aux quatre multiplications $P_1 Q_1$, $P_1 Q_2$, $P_2 Q_1$ et $P_2 Q_2$ entre polynômes de degrés strictement inférieurs à n , plus deux multiplications par X^n et X^{2n} qui ne sont que des ajouts de zéros en tête de liste.

2. On sépare les deux étapes de l'algorithme : d'abord la découpe des polynômes (dans laquelle il ne faut pas oublier de donner n en argument car ce n'est pas forcément le milieu du polynôme, n doit être le même pour P et Q). Le découpage $P1, P2 = \text{decoupe}(P, n)$ correspond à l'écriture $P = P_1 + X^n P_2$.

Code 34 (*polynome.py* (4)).

```
def decoupe(P,n):
    if (degre(P)<n): return P, [0]
    else: return P[0:n], P[n:]
```

On a aussi besoin d'une fonction *produit_monomie*(P, n) qui renvoie le polynôme $X^n \cdot P$ par un décalage. Voici la multiplication proprement dite avec les appels récursifs et leur combinaison.

Code 35 (*polynome.py* (5)).

```
def produit_assez_rapide(P,Q):
    p = degre(P) ; q = degre(Q)
    if (p == 0): return [P[0]*k for k in Q] # Condition initiale: P=cst
    if (q == 0): return [Q[0]*k for k in P] # Condition initiale: Q=cst
    n = (max(p,q)+1)//2 # demi-degré
    P1,P2 = decoupe(P,n) # decoupages
    Q1,Q2 = decoupe(Q,n)
    P1Q1 = produit_assez_rapide(P1,Q1) # produits en petits degrés
    P2Q2 = produit_assez_rapide(P2,Q2)
    P1Q2 = produit_assez_rapide(P1,Q2)
    P2Q1 = produit_assez_rapide(P2,Q1)
    R1 = produit_monomie(somme(P1Q2,P2Q1),n) # décalages
    R2 = produit_monomie(P2Q2,2*n)
```

```
    return somme(P1Q1,somme(R1,R2))           # sommes
```

La relation de récurrence qui exprime la complexité de cet algorithme est $C(n) = 4C(n/2) + O(n)$ et elle se résout en $C(n) = O(n^2)$. Voir la question suivante pour une méthode de résolution.

3.

Code 36 (*polynome.py* (6)).

```
def produit_rapide(P,Q):
    p = degre(P) ; q = degre(Q)
    if (p == 0): return [P[0]*k for k in Q]      # Condition initiale: P=cst
    if (q == 0): return [Q[0]*k for k in P]      # Condition initiale: Q=cst
    n = (max(p,q)+1)//2                         # demi-degré
    P1,P2 = decoupe(P,n)                         # decoupages
    Q1,Q2 = decoupe(Q,n)
    P1Q1 = produit_rapide(P1,Q1)                  # produits en petits degrés
    P2Q2 = produit_rapide(P2,Q2)
    PQ = produit_rapide(somme(P1,P2),somme(Q1,Q2))
    R1 = somme(PQ,somme([-k for k in P1Q1],[-k for k in P2Q2]))  # décalages
    R1 = produit_monomie(R1,n)
    R2 = produit_monomie(P2Q2,2*n)
    return somme(P1Q1,somme(R1,R2))           # sommes
```

4. Notons $C(n)$ la complexité de la multiplication entre deux polynômes de degrés strictement inférieurs à n . En plus des trois appels récursifs, il y a des opérations linéaires : deux calculs de degrés, deux découpes en $n/2$ puis des additions : deux de taille $n/2$, une de taille n , une de taille $3n/2$ et une de taille $2n$. On obtient donc la relation de récurrence suivante :

$$C(n) = 3 \cdot C(n/2) + \gamma n$$

où $\gamma = \frac{15}{2}$. Une méthode de résolution est de poser $\alpha_\ell = \frac{C(2^\ell)}{3^\ell}$ qui vérifie $\alpha_\ell = \alpha_{\ell-1} + \gamma \left(\frac{2}{3}\right)^\ell$. D'où on tire, puisque $\alpha_0 = C(1) = 1$,

$$\alpha_\ell = \gamma \sum_{k=1}^{\ell} \left(\frac{2}{3}\right)^k + \alpha_0 = 3\gamma \left(1 - \left(\frac{2}{3}\right)^{\ell+1}\right) + 1 - \gamma$$

puis pour $n = 2^\ell$:

$$C(n) = C(2^\ell) = 3^\ell \alpha_\ell = \gamma(3^{\ell+1} - 2^{\ell+1}) + (1 - \gamma)3^\ell = O(3^\ell) = O(2^{\ell \frac{\ln 3}{\ln 2}}) = O(n^{\frac{\ln 3}{\ln 2}})$$

La complexité de la multiplication de Karatsuba est donc $O(n^{\frac{\ln 3}{\ln 2}}) \simeq O(n^{1.585})$.

6.4. Optimiser ses algorithmes

Voici quelques petites astuces pour accélérer l'écriture ou la vitesse des algorithmes :

- $k ** 3$ au lieu de $k * k * k$ (cela économise de la mémoire, une seule variable au lieu de 3) ;
- $k ** 2 <= n$ au lieu de $k <= \sqrt{n}$ (les calculs avec les entiers sont beaucoup plus rapides qu'avec les réels) ;
- $x += 1$ au lieu de $x = x + 1$ (gain de mémoire) ;
- $a, b = a+b, a-b$ au lieu de $newa = a+b ; newb = a-b ; a = newa ; b = newb$ (gain de mémoire, code plus court).

Cependant il ne faut pas que cela nuise à la lisibilité du code : il est important que quelqu'un puisse relire et modifier votre code. Le plus souvent c'est vous même qui modifierez les algorithmes qui vous avez écrits et vous serez ravi d'y trouver des commentaires clairs et précis !

- Mini-exercices.**
1. Faire une fonction qui renvoie le pgcd de deux polynômes.
 2. Comparer les complexités des deux méthodes suivantes pour évaluer un polynôme P en une valeur $x_0 \in \mathbb{R}$: $P(x_0) = a_0 + a_1x_0 + \dots + a_{n-1}x_0^{n-1} + a_nx_0^n$ et $P(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots + x_0(a_{n-1} + a_nx_0)))$ (méthode de Horner).
 3. Comment trouver le maximum d'une liste ? Montrer que votre méthode est de complexité minimale (en terme du nombre de comparaisons).
 4. Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction continue vérifiant $f(a) \cdot f(b) \leq 0$. Combien d'itérations de la méthode de dichotomie sont nécessaires pour obtenir une racine de $f(x) = 0$ avec une précision inférieure à ϵ ?
 5. Programmer plusieurs façons de calculer les coefficients du binôme de Newton $\binom{n}{k}$ et les comparer.
 6. Trouver une méthode de calcul de 2^n qui utilise peu de multiplications. On commencera par écrire n en base 2.

Auteurs du chapitre Rédaction : Arnaud Bodin

Relecture : Jean-François Barraud

Remerciements à Lionel Rieg pour son tp sur l'algorithme de Karatsuba

Zéros des fonctions

[Vidéo ■ partie 1. La dichotomie](#)

[Vidéo ■ partie 2. La méthode de la sécante](#)

[Vidéo ■ partie 3. La méthode de Newton](#)

Dans ce chapitre nous allons appliquer toutes les notions précédentes sur les suites et les fonctions, à la recherche des zéros des fonctions. Plus précisément, nous allons voir trois méthodes afin de trouver des approximations des solutions d'une équation du type ($f(x) = 0$).

1. La dichotomie

1.1. Principe de la dichotomie

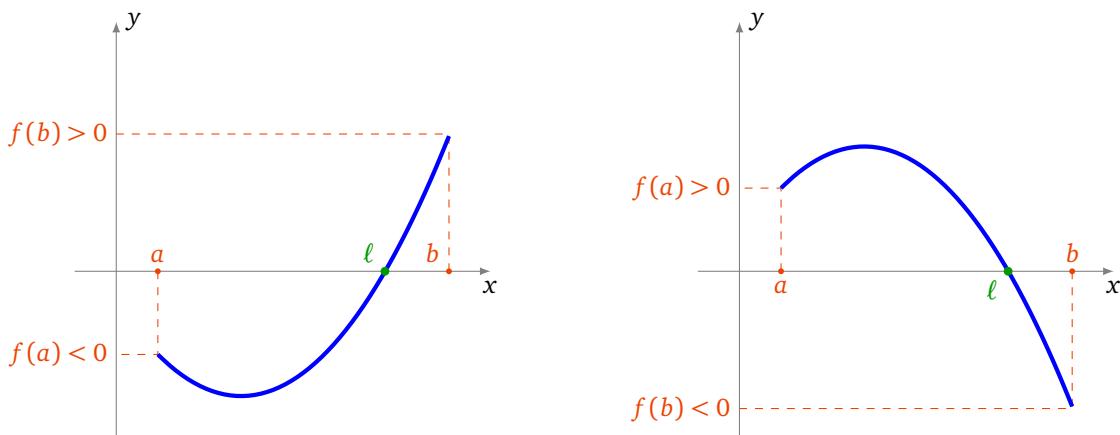
Le principe de dichotomie repose sur la version suivante du [théorème des valeurs intermédiaires](#) :

Théorème 1.

Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction continue sur un segment.

Si $f(a) \cdot f(b) \leq 0$, alors il existe $\ell \in [a, b]$ tel que $f(\ell) = 0$.

La condition $f(a) \cdot f(b) \leq 0$ signifie que $f(a)$ et $f(b)$ sont de signes opposés (ou que l'un des deux est nul). L'hypothèse de continuité est essentielle !



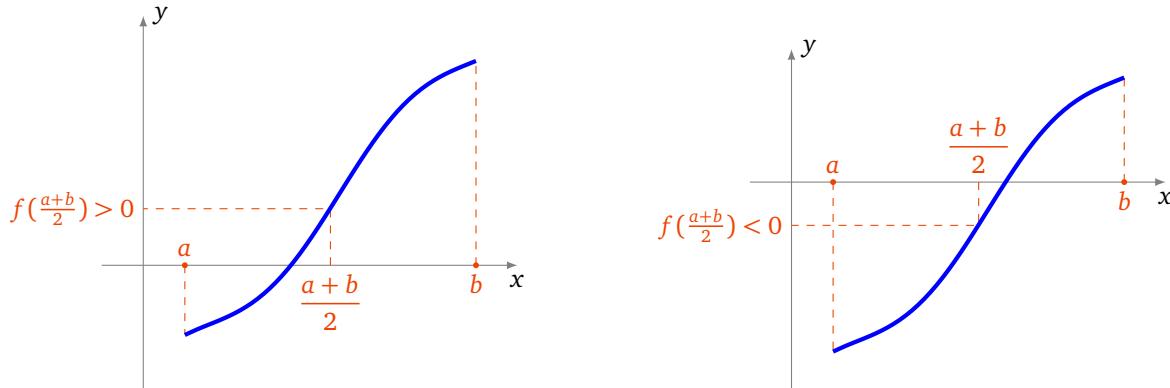
Ce théorème affirme qu'il existe au moins une solution de l'équation ($f(x) = 0$) dans l'intervalle $[a, b]$. Pour le rendre effectif, et trouver une solution (approchée) de l'équation ($f(x) = 0$), il s'agit maintenant de l'appliquer sur un intervalle suffisamment petit. On va voir que cela permet d'obtenir un ℓ solution de l'équation ($f(x) = 0$) comme la limite d'une suite.

Voici comment construire une suite d'intervalles emboîtés, dont la longueur tend vers 0, et contenant chacun une solution de l'équation ($f(x) = 0$).

On part d'une fonction $f : [a, b] \rightarrow \mathbb{R}$ continue, avec $a < b$, et $f(a) \cdot f(b) \leq 0$.

Voici la première étape de la construction : on regarde le signe de la valeur de la fonction f appliquée au point milieu $\frac{a+b}{2}$.

- Si $f(a) \cdot f(\frac{a+b}{2}) \leq 0$, alors il existe $c \in [a, \frac{a+b}{2}]$ tel que $f(c) = 0$.
- Si $f(a) \cdot f(\frac{a+b}{2}) > 0$, cela implique que $f(\frac{a+b}{2}) \cdot f(b) \leq 0$, et alors il existe $c \in [\frac{a+b}{2}, b]$ tel que $f(c) = 0$.



Nous avons obtenu un intervalle de longueur moitié dans lequel l'équation ($f(x) = 0$) admet une solution. On itère alors le procédé pour diviser de nouveau l'intervalle en deux.

Voici le processus complet :

- **Au rang 0 :**

On pose $a_0 = a$, $b_0 = b$. Il existe une solution x_0 de l'équation ($f(x) = 0$) dans l'intervalle $[a_0, b_0]$.

- **Au rang 1 :**

- Si $f(a_0) \cdot f(\frac{a_0+b_0}{2}) \leq 0$, alors on pose $a_1 = a_0$ et $b_1 = \frac{a_0+b_0}{2}$,
- sinon on pose $a_1 = \frac{a_0+b_0}{2}$ et $b_1 = b_0$.

— Dans les deux cas, il existe une solution x_1 de l'équation ($f(x) = 0$) dans l'intervalle $[a_1, b_1]$.

- ...

- **Au rang n :** supposons construit un intervalle $[a_n, b_n]$, de longueur $\frac{b-a}{2^n}$, et contenant une solution x_n de l'équation ($f(x) = 0$). Alors :

- Si $f(a_n) \cdot f(\frac{a_n+b_n}{2}) \leq 0$, alors on pose $a_{n+1} = a_n$ et $b_{n+1} = \frac{a_n+b_n}{2}$,
- sinon on pose $a_{n+1} = \frac{a_n+b_n}{2}$ et $b_{n+1} = b_n$.

— Dans les deux cas, il existe une solution x_{n+1} de l'équation ($f(x) = 0$) dans l'intervalle $[a_{n+1}, b_{n+1}]$.

À chaque étape on a

$$a_n \leq x_n \leq b_n.$$

On arrête le processus dès que $b_n - a_n = \frac{b-a}{2^n}$ est inférieur à la précision souhaitée.

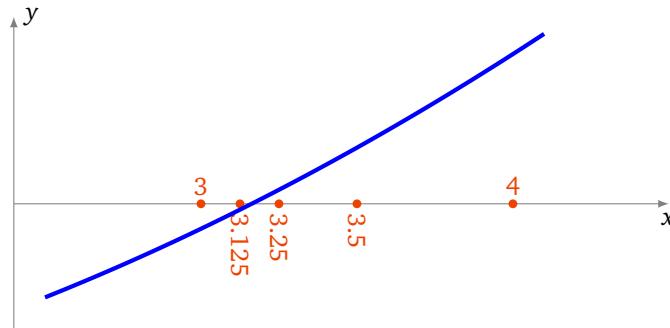
Comme (a_n) est par construction une suite croissante, (b_n) une suite décroissante, et $(b_n - a_n) \rightarrow 0$ lorsque $n \rightarrow +\infty$, les suites (a_n) et (b_n) sont adjacentes et donc elles admettent une même limite. D'après le théorème des gendarmes, c'est aussi la limite disons ℓ de la suite (x_n) . La continuité de f montre que $f(\ell) = \lim_{n \rightarrow +\infty} f(x_n) = \lim_{n \rightarrow +\infty} 0 = 0$. Donc les suites (a_n) et (b_n) tendent toutes les deux vers ℓ , qui est une solution de l'équation ($f(x) = 0$).

1.2. Résultats numériques pour $\sqrt{10}$

Nous allons calculer une approximation de $\sqrt{10}$. Soit la fonction f définie par $f(x) = x^2 - 10$, c'est une fonction continue sur \mathbb{R} qui s'annule en $\pm\sqrt{10}$. De plus $\sqrt{10}$ est l'unique solution positive de l'équation ($f(x) = 0$). Nous pouvons restreindre la fonction f à l'intervalle $[3, 4]$: en effet $3^2 = 9 \leq 10$ donc $3 \leq \sqrt{10}$ et $4^2 = 16 \geq 10$ donc $4 \geq \sqrt{10}$. En d'autre termes $f(3) \leq 0$ et $f(4) \geq 0$, donc l'équation ($f(x) = 0$) admet

une solution dans l'intervalle $[3, 4]$ d'après le théorème des valeurs intermédiaires, et par unicité c'est $\sqrt{10}$, donc $\sqrt{10} \in [3, 4]$.

Notez que l'on ne choisit pas pour f la fonction $x \mapsto x - \sqrt{10}$ car on ne connaît pas la valeur de $\sqrt{10}$. C'est ce que l'on cherche à calculer !



Voici les toutes premières étapes :

1. On pose $a_0 = 3$ et $b_0 = 4$, on a bien $f(a_0) \leq 0$ et $f(b_0) \geq 0$. On calcule $\frac{a_0+b_0}{2} = 3,5$ puis $f(\frac{a_0+b_0}{2}) : f(3,5) = 3,5^2 - 10 = 2,25 \geq 0$. Donc $\sqrt{10}$ est dans l'intervalle $[3; 3,5]$ et on pose $a_1 = a_0 = 3$ et $b_1 = \frac{a_0+b_0}{2} = 3,5$.
2. On sait donc que $f(a_1) \leq 0$ et $f(b_1) \geq 0$. On calcule $f(\frac{a_1+b_1}{2}) = f(3,25) = 0,5625 \geq 0$, on pose $a_2 = 3$ et $b_2 = 3,25$.
3. On calcule $f(\frac{a_2+b_2}{2}) = f(3,125) = -0,23\dots \leq 0$. Comme $f(b_2) \geq 0$ alors cette fois f s'annule sur le second intervalle $[\frac{a_2+b_2}{2}, b_2]$ et on pose $a_3 = \frac{a_2+b_2}{2} = 3,125$ et $b_3 = b_2 = 3,25$.

À ce stade, on a prouvé : $3,125 \leq \sqrt{10} \leq 3,25$.

Voici la suite des étapes :

$$\begin{array}{ll}
 a_0 = 3 & b_0 = 4 \\
 a_1 = 3 & b_1 = 3,5 \\
 a_2 = 3 & b_2 = 3,25 \\
 a_3 = 3,125 & b_3 = 3,25 \\
 a_4 = 3,125 & b_4 = 3,1875 \\
 a_5 = 3,15625 & b_5 = 3,1875 \\
 a_6 = 3,15625 & b_6 = 3,171875 \\
 a_7 = 3,15625 & b_7 = 3,164062\dots \\
 a_8 = 3,16015\dots & b_8 = 3,164062\dots
 \end{array}$$

Donc en 8 étapes on obtient l'encadrement :

$$3,160 \leq \sqrt{10} \leq 3,165$$

En particulier, on vient d'obtenir les deux premières décimales : $\sqrt{10} = 3,16\dots$

1.3. Résultats numériques pour $(1, 10)^{1/12}$

Nous cherchons maintenant une approximation de $(1, 10)^{1/12}$. Soit $f(x) = x^{12} - 1,10$. On pose $a_0 = 1$ et $b_0 = 1,1$. Alors $f(a_0) = -0,10 \leq 0$ et $f(b_0) = 2,038\dots \geq 0$.

$a_0 = 1$	$b_0 = 1,10$
$a_1 = 1$	$b_1 = 1,05$
$a_2 = 1$	$b_2 = 1,025$
$a_3 = 1$	$b_3 = 1,0125$
$a_4 = 1,00625$	$b_4 = 1,0125$
$a_5 = 1,00625$	$b_5 = 1,00937\dots$
$a_6 = 1,00781\dots$	$b_6 = 1,00937\dots$
$a_7 = 1,00781\dots$	$b_7 = 1,00859\dots$
$a_8 = 1,00781\dots$	$b_8 = 1,00820\dots$

Donc en 8 étapes on obtient l'encadrement :

$$1,00781 \leq (1,10)^{1/12} \leq 1,00821$$

1.4. Calcul de l'erreur

La méthode de dichotomie a l'énorme avantage de fournir un encadrement d'une solution ℓ de l'équation ($f(x) = 0$). Il est donc facile d'avoir une majoration de l'erreur. En effet, à chaque étape, la taille l'intervalle contenant ℓ est divisée par 2. Au départ, on sait que $\ell \in [a, b]$ (de longueur $b - a$) ; puis $\ell \in [a_1, b_1]$ (de longueur $\frac{b-a}{2}$) ; puis $\ell \in [a_2, b_2]$ (de longueur $\frac{b-a}{4}$) ; ... ; $[a_n, b_n]$ étant de longueur $\frac{b-a}{2^n}$.

Si, par exemple, on souhaite obtenir une approximation de ℓ à 10^{-N} près, comme on sait que $a_n \leq \ell \leq b_n$, on obtient $|\ell - a_n| \leq |b_n - a_n| = \frac{b-a}{2^n}$. Donc pour avoir $|\ell - a_n| \leq 10^{-N}$, il suffit de choisir n tel que $\frac{b-a}{2^n} \leq 10^{-N}$. Nous allons utiliser le logarithme décimal :

$$\begin{aligned} \frac{b-a}{2^n} \leq 10^{-N} &\iff (b-a)10^N \leq 2^n \\ &\iff \log(b-a) + \log(10^N) \leq \log(2^n) \\ &\iff \log(b-a) + N \leq n \log 2 \\ &\iff n \geq \frac{N + \log(b-a)}{\log 2} \end{aligned}$$

Sachant $\log 2 = 0,301\dots$, si par exemple $b - a \leq 1$, voici le nombre d'itérations suffisantes pour avoir une précision de 10^{-N} (ce qui correspond, à peu près, à N chiffres exacts après la virgule).

10^{-10} (~ 10 décimales)	34 itérations
10^{-100} (~ 100 décimales)	333 itérations
10^{-1000} (~ 1000 décimales)	3322 itérations

Il faut entre 3 et 4 itérations supplémentaires pour obtenir une nouvelle décimale.

Remarque.

En toute rigueur il ne faut pas confondre précision et nombre de décimales exactes, par exemple 0,999 est une approximation de 1,000 à 10^{-3} près, mais aucune décimale après la virgule n'est exacte. En pratique, c'est la précision qui est la plus importante, mais il est plus frappant de parler du nombre de décimales exactes.

1.5. Algorithmes

Voici comment implémenter la dichotomie dans le langage Python. Tout d'abord on définit une fonction f (ici par exemple $f(x) = x^2 - 10$) :

Code 37 (*dichotomie.py (1)*).

```
def f(x):
    return x*x - 10
```

Puis la dichotomie proprement dite : en entrée de la fonction, on a pour variables a, b et n le nombre d'étapes voulues.

Code 38 (*dichotomie.py (2)*).

```
def dicho(a,b,n):
    for i in range(n):
        c = (a+b)/2
        if f(a)*f(c) <= 0:
            b = c
        else:
            a = c
    return a,b
```

Même algorithme, mais avec cette fois en entrée la précision souhaitée :

Code 39 (*dichotomie.py (3)*).

```
def dichobis(a,b,prec):
    while b-a>prec:
        c = (a+b)/2
        if f(a)*f(c) <= 0:
            b = c
        else:
            a = c
    return a,b
```

Enfin, voici la version récursive de l'algorithme de dichotomie.

Code 40 (*dichotomie.py (4)*).

```
def dichotomie(a,b,prec):
    if b-a<=prec:
        return a,b
    else:
        c = (a+b)/2
        if f(a)*f(c) <= 0:
            return dichotomie(a,c,prec)
        else:
            return dichotomie(c,b,prec)
```

Mini-exercices. 1. À la main, calculer un encadrement à 0,1 près de $\sqrt{3}$. Idem avec $\sqrt[3]{2}$.

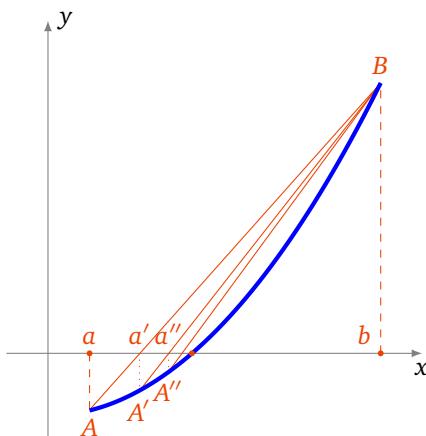
2. Calculer une approximation des solutions de l'équation $x^3 + 1 = 3x$.
3. Est-il plus efficace de diviser l'intervalle en 4 au lieu d'en 2 ? (À chaque itération, la dichotomie classique nécessite l'évaluation de f en une nouvelle valeur $\frac{a+b}{2}$ pour une précision améliorée d'un facteur 2.)
4. Écrire un algorithme pour calculer plusieurs solutions de $(f(x) = 0)$.

5. On se donne un tableau trié de taille N , rempli de nombres appartenant à $\{1, \dots, n\}$. Écrire un algorithme qui teste si une valeur k apparaît dans le tableau et en quelle position.

2. La méthode de la sécante

2.1. Principe de la sécante

L'idée de la méthode de la sécante est très simple : pour une fonction f continue sur un intervalle $[a, b]$, et vérifiant $f(a) \leq 0, f(b) > 0$, on trace le segment $[AB]$ où $A = (a, f(a))$ et $B = (b, f(b))$. Si le segment reste au-dessus du graphe de f alors la fonction s'annule sur l'intervalle $[a', b]$ où $(a', 0)$ est le point d'intersection de la droite (AB) avec l'axe des abscisses. La droite (AB) s'appelle la **sécante**. On recommence en partant maintenant de l'intervalle $[a', b]$ pour obtenir une valeur a'' .



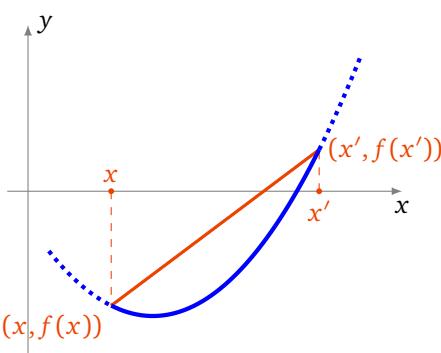
Proposition 1.

Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction continue, strictement croissante et convexe telle que $f(a) \leq 0, f(b) > 0$. Alors la suite définie par

$$a_0 = a \quad \text{et} \quad a_{n+1} = a_n - \frac{b - a_n}{f(b) - f(a_n)} f(a_n)$$

est croissante et converge vers la solution ℓ de $(f(x) = 0)$.

L'hypothèse f **convexe** signifie exactement que pour tout x, x' dans $[a, b]$ la sécante (ou corde) entre $(x, f(x))$ et $(x', f(x'))$ est au-dessus du graphe de f .



Démonstration. 1. Justifions d'abord la construction de la suite récurrente.

L'équation de la droite passant par les deux points $(a, f(a))$ et $(b, f(b))$ est

$$y = (x - a) \frac{f(b) - f(a)}{b - a} + f(a)$$

Cette droite intersecte l'axe des abscisses en $(a', 0)$ qui vérifie donc $0 = (a' - a) \frac{f(b) - f(a)}{b - a} + f(a)$, donc $a' = a - \frac{b-a}{f(b)-f(a)} f(a)$.

2. Croissance de (a_n) .

Montrons par récurrence que $f(a_n) \leq 0$. C'est vrai au rang 0 car $f(a_0) = f(a) \leq 0$ par hypothèse. Supposons vraie l'hypothèse au rang n . Si $a_{n+1} < a_n$ (un cas qui s'avérera *a posteriori* jamais réalisé), alors comme f est strictement croissante, on a $f(a_{n+1}) < f(a_n)$, et en particulier $f(a_{n+1}) \leq 0$. Sinon $a_{n+1} \geq a_n$. Comme f est convexe : la sécante entre $(a_n, f(a_n))$ et $(b, f(b))$ est au-dessus du graphe de f . En particulier le point $(a_{n+1}, 0)$ (qui est sur cette sécante par définition a_{n+1}) est au-dessus du point $(a_{n+1}, f(a_{n+1}))$, et donc $f(a_{n+1}) \leq 0$ aussi dans ce cas, ce qui conclut la récurrence.

Comme $f(a_n) \leq 0$ et f est croissante, alors par la formule $a_{n+1} = a_n - \frac{b-a_n}{f(b)-f(a_n)} f(a_n)$, on obtient que $a_{n+1} \geq a_n$.

3. Convergence de (a_n) .

La suite (a_n) est croissante et majorée par b , donc elle converge. Notons ℓ sa limite. Par continuité $f(a_n) \rightarrow f(\ell)$. Comme pour tout n , $f(a_n) \leq 0$, on en déduit que $f(\ell) \leq 0$. En particulier, comme on suppose $f(b) > 0$, on a $\ell < b$. Comme $a_n \rightarrow \ell$, $a_{n+1} \rightarrow \ell$, $f(a_n) \rightarrow f(\ell)$, l'égalité $a_{n+1} = a_n - \frac{b-a_n}{f(b)-f(a_n)} f(a_n)$ devient à la limite (lorsque $n \rightarrow +\infty$) : $\ell = \ell - \frac{b-\ell}{f(b)-f(\ell)} f(\ell)$, ce qui implique $f(\ell) = 0$.

Conclusion : (a_n) converge vers la solution de $(f(x) = 0)$.

□

2.2. Résultats numériques pour $\sqrt{10}$

Pour $a = 3$, $b = 4$, $f(x) = x^2 - 10$ voici les résultats numériques, est aussi indiquée une majoration de l'erreur $\epsilon_n = \sqrt{10} - a_n$ (voir ci-après).

$a_0 = 3$	$\epsilon_0 \leq 0,1666\dots$
$a_1 = 3,14285714285\dots$	$\epsilon_1 \leq 0,02040\dots$
$a_2 = 3,16000000000\dots$	$\epsilon_2 \leq 0,00239\dots$
$a_3 = 3,16201117318\dots$	$\epsilon_3 \leq 0,00028\dots$
$a_4 = 3,16224648985\dots$	$\epsilon_4 \leq 3,28\dots \cdot 10^{-5}$
$a_5 = 3,16227401437\dots$	$\epsilon_5 \leq 3,84\dots \cdot 10^{-6}$
$a_6 = 3,16227723374\dots$	$\epsilon_6 \leq 4,49\dots \cdot 10^{-7}$
$a_7 = 3,16227761029\dots$	$\epsilon_7 \leq 5,25\dots \cdot 10^{-8}$
$a_8 = 3,16227765433\dots$	$\epsilon_8 \leq 6,14\dots \cdot 10^{-9}$

2.3. Résultats numériques pour $(1, 10)^{1/12}$

Voici les résultats numériques avec une majoration de l'erreur $\epsilon_n = (1, 10)^{1/12} - a_n$, avec $f(x) = x^{12} - 1, 10$, $a = 1$ et $b = 1, 1$

$a_0 = 1$	$\epsilon_0 \leq 0,0083\dots$
$a_1 = 1,00467633\dots$	$\epsilon_1 \leq 0,0035\dots$
$a_2 = 1,00661950\dots$	$\epsilon_2 \leq 0,0014\dots$
$a_3 = 1,00741927\dots$	$\epsilon_3 \leq 0,00060\dots$
$a_4 = 1,00774712\dots$	$\epsilon_4 \leq 0,00024\dots$
$a_5 = 1,00788130\dots$	$\epsilon_5 \leq 0,00010\dots$
$a_6 = 1,00793618\dots$	$\epsilon_6 \leq 4,14\dots \cdot 10^{-5}$
$a_7 = 1,00795862\dots$	$\epsilon_7 \leq 1,69\dots \cdot 10^{-5}$
$a_8 = 1,00796779\dots$	$\epsilon_8 \leq 6,92\dots \cdot 10^{-6}$

2.4. Calcul de l'erreur

La méthode de la sécante fournit l'encadrement $a_n \leq l \leq b$. Mais comme b est fixe cela ne donne pas d'information exploitable pour $|l - a_n|$. Voici une façon générale d'estimer l'erreur, à l'aide du théorème des accroissements finis.

Proposition 2.

Soit $f : I \rightarrow \mathbb{R}$ une fonction dérivable et ℓ tel que $f(\ell) = 0$. S'il existe une constante $m > 0$ telle que pour tout $x \in I$, $|f'(x)| \geq m$ alors

$$|x - \ell| \leq \frac{|f(x)|}{m} \quad \text{pour tout } x \in I.$$

Démonstration. Par l'inégalité des accroissements finis entre x et ℓ : $|f(x) - f(\ell)| \geq m|x - \ell|$ mais $f(\ell) = 0$, d'où la majoration. \square

Exemple 1 (Erreur pour $\sqrt{10}$).

Soit $f(x) = x^2 - 10$ et l'intervalle $I = [3, 4]$. Alors $f'(x) = 2x$ donc $|f'(x)| \geq 6$ sur I . On pose donc $m = 6$, $\ell = \sqrt{10}$, $x = a_n$. On obtient l'estimation de l'erreur :

$$\epsilon_n = |\ell - a_n| \leq \frac{|f(a_n)|}{m} = \frac{|a_n^2 - 10|}{6}$$

Par exemple on a trouvé $a_2 = 3,16\dots \leq 3,17$ donc $\sqrt{10} - a_2 \leq \frac{|3,17^2 - 10|}{6} = 0,489$.

Pour a_8 on a trouvé $a_8 = 3,1622776543347473\dots$ donc $\sqrt{10} - a_8 \leq \frac{|a_8^2 - 10|}{6} = 6,14\dots \cdot 10^{-9}$. On a en fait 7 décimales exactes après la virgule.

Dans la pratique, voici le nombre d'itérations suffisantes pour avoir une précision de 10^{-n} pour cet exemple. Grossièrement, une itération de plus donne une décimale supplémentaire.

10^{-10} (~ 10 décimales)	10 itérations
10^{-100} (~ 100 décimales)	107 itérations
10^{-1000} (~ 1000 décimales)	1073 itérations

Exemple 2 (Erreur pour $(1, 10)^{1/12}$).

On pose $f(x) = x^{12} - 1,10$, $I = [1; 1,10]$ et $\ell = (1, 10)^{1/12}$. Comme $f'(x) = 12x^{11}$, si on pose de plus $m = 12$, on a $|f'(x)| \geq m$ pour $x \in I$. On obtient

$$\epsilon_n = |\ell - a_n| \leq \frac{|a_n^{12} - 1,10|}{12}.$$

Par exemple $a_8 = 1.0079677973185432\dots$ donc

$$|(1, 10)^{1/12} - a_8| \leq \frac{|a_8^{12} - 1,10|}{12} = 6,92\dots \cdot 10^{-6}.$$

2.5. Algorithme

Voici l'algorithme : c'est tout simplement la mise en œuvre de la suite récurrente (a_n) .

Code 41 (*secante.py*).

```
def secante(a,b,n):
    for i in range(n):
        a = a-f(a)*(b-a)/(f(b)-f(a))
    return a
```

Mini-exercices. 1. À la main, calculer un encadrement à 0,1 près de $\sqrt{3}$. Idem avec $\sqrt[3]{2}$.

2. Calculer une approximation des solutions de l'équation $x^3 + 1 = 3x$.

3. Calculer une approximation de la solution de l'équation ($\cos x = 0$) sur $[0, \pi]$. Idem avec ($\cos x = 2 \sin x$).

4. Étudier l'équation ($\exp(-x) = -\ln(x)$). Donner une approximation de la (ou des) solution(s) et une majoration de l'erreur correspondante.

3. La méthode de Newton

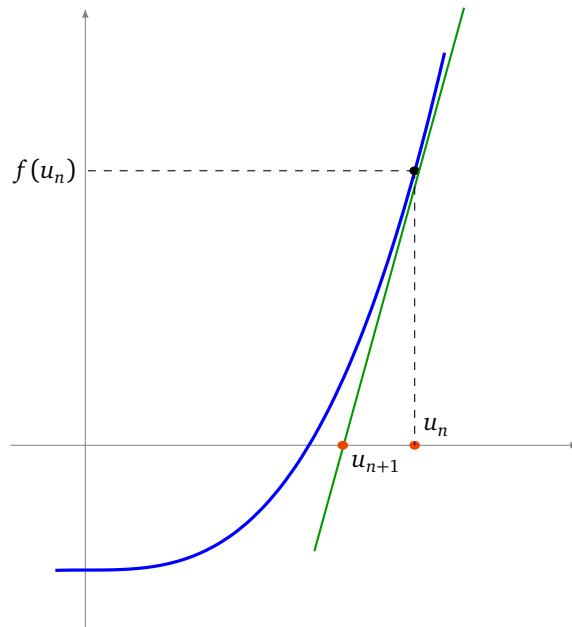
3.1. Méthode de Newton

La méthode de Newton consiste à remplacer la sécante de la méthode précédente par la tangente. Elle est d'une redoutable efficacité.

Partons d'une fonction dérivable $f : [a, b] \rightarrow \mathbb{R}$ et d'un point $u_0 \in [a, b]$. On appelle $(u_1, 0)$ l'intersection de la tangente au graphe de f en $(u_0, f(u_0))$ avec l'axe des abscisses. Si $u_1 \in [a, b]$ alors on recommence l'opération avec la tangente au point d'abscisse u_1 . Ce processus conduit à la définition d'une suite récurrente :

$$u_0 \in [a, b] \quad \text{et} \quad u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}.$$

Démonstration. En effet la tangente au point d'abscisse u_n a pour équation : $y = f'(u_n)(x - u_n) + f(u_n)$. Donc le point $(x, 0)$ appartenant à la tangente (et à l'axe des abscisses) vérifie $0 = f'(u_n)(x - u_n) + f(u_n)$. D'où $x = u_n - \frac{f(u_n)}{f'(u_n)}$. \square



3.2. Résultats pour $\sqrt{10}$

Pour calculer \sqrt{a} , on pose $f(x) = x^2 - a$, avec $f'(x) = 2x$. La suite issue de la méthode de Newton est déterminée par $u_0 > 0$ et la relation de récurrence $u_{n+1} = u_n - \frac{u_n^2 - a}{2u_n}$. Suite qui pour cet exemple s'appelle **suite de Héron** et que l'on récrit souvent

$$u_0 > 0 \quad \text{et} \quad u_{n+1} = \frac{1}{2} \left(u_n + \frac{a}{u_n} \right).$$

Proposition 3.

Cette suite (u_n) converge vers \sqrt{a} .

Pour le calcul de $\sqrt{10}$, on pose par exemple $u_0 = 4$, et on peut même commencer les calculs à la main :

$$\begin{aligned} u_0 &= 4 \\ u_1 &= \frac{1}{2} \left(u_0 + \frac{10}{u_0} \right) = \frac{1}{2} \left(4 + \frac{10}{4} \right) = \frac{13}{4} = 3,25 \\ u_2 &= \frac{1}{2} \left(u_1 + \frac{10}{u_1} \right) = \frac{1}{2} \left(\frac{13}{4} + \frac{10}{\frac{13}{4}} \right) = \frac{329}{104} = 3,1634\dots \\ u_3 &= \frac{1}{2} \left(u_2 + \frac{10}{u_2} \right) = \frac{216401}{68432} = 3,16227788\dots \\ u_4 &= 3,162277660168387\dots \end{aligned}$$

Pour u_4 on obtient $\sqrt{10} = 3,1622776601683\dots$ avec déjà 13 décimales exactes !

Voici la preuve de la convergence de la suite (u_n) vers \sqrt{a} .

Démonstration.

$$u_0 > 0 \quad \text{et} \quad u_{n+1} = \frac{1}{2} \left(u_n + \frac{a}{u_n} \right).$$

- Montrons que $u_n \geq \sqrt{a}$ pour $n \geq 1$.

Tout d'abord

$$u_{n+1}^2 - a = \frac{1}{4} \left(\frac{u_n^2 + a}{u_n} \right)^2 - a = \frac{1}{4u_n^2} (u_n^4 - 2au_n^2 + a^2) = \frac{1}{4} \frac{(u_n^2 - a)^2}{u_n^2}$$

Donc $u_{n+1}^2 - a \geq 0$. Comme il est clair que pour tout $n \geq 0$, $u_n \geq 0$, on en déduit que pour tout $n \geq 0$, $u_{n+1} \geq \sqrt{a}$. (Notez que u_0 lui est quelconque.)

- Montrons que $(u_n)_{n \geq 1}$ est une suite décroissante qui converge.

Comme $\frac{u_{n+1}}{u_n} = \frac{1}{2} \left(1 + \frac{a}{u_n^2} \right)$, et que pour $n \geq 1$ on vient de voir que $u_n^2 \geq a$ (donc $\frac{a}{u_n^2} \leq 1$), alors $\frac{u_{n+1}}{u_n} \leq 1$, pour tout $n \leq 1$.

Conséquence : la suite $(u_n)_{n \geq 1}$ est décroissante et minorée par 0 donc elle converge.

- (u_n) converge vers \sqrt{a} .

Notons ℓ la limite de (u_n) . Alors $u_n \rightarrow \ell$ et $u_{n+1} \rightarrow \ell$. Lorsque $n \rightarrow +\infty$ dans la relation $u_{n+1} = \frac{1}{2} \left(u_n + \frac{a}{u_n} \right)$, on obtient $\ell = \frac{1}{2} \left(\ell + \frac{a}{\ell} \right)$. Ce qui conduit à la relation $\ell^2 = a$ et par positivité de la suite, $\ell = \sqrt{a}$.

□

3.3. Résultats numériques pour $(1, 10)^{1/12}$

Pour calculer $(1, 10)^{1/12}$, on pose $f(x) = x^{12} - a$ avec $a = 1, 10$. On a $f'(x) = 12x^{11}$. On obtient $u_{n+1} = u_n - \frac{u_n^{12} - a}{12u_n^{11}}$. Ce que l'on reformule ainsi :

$$u_0 > 0 \quad \text{et} \quad u_{n+1} = \frac{1}{12} \left(11u_n + \frac{a}{u_n^{11}} \right).$$

Voici les résultats numériques pour $(1, 10)^{1/12}$ en partant de $u_0 = 1$.

$$\begin{aligned} u_0 &= 1 \\ u_1 &= 1,008333333333333\dots \\ u_2 &= 1,0079748433368980\dots \\ u_3 &= 1,0079741404315996\dots \\ u_4 &= 1,0079741404289038\dots \end{aligned}$$

Toutes les décimales affichées pour u_4 sont exactes : $(1,10)^{1/12} = 1,0079741404289038\dots$

3.4. Calcul de l'erreur pour $\sqrt{10}$

Proposition 4. 1. Soit k tel que $u_1 - \sqrt{a} \leq k$. Alors pour tout $n \geq 1$:

$$u_n - \sqrt{a} \leq 2\sqrt{a} \left(\frac{k}{2\sqrt{a}} \right)^{2^{n-1}}$$

2. Pour $a = 10$, $u_0 = 4$, on a :

$$u_n - \sqrt{10} \leq 8 \left(\frac{1}{24} \right)^{2^{n-1}}$$

Admirez la puissance de la méthode de Newton : 11 itérations donnent déjà 1000 décimales exactes après la virgule. Cette rapidité de convergence se justifie grâce au calcul de l'erreur : la précision est multipliée par 2 à chaque étape, donc à chaque itération le nombre de décimales exactes double !

10^{-10} (~ 10 décimales)	4 itérations
10^{-100} (~ 100 décimales)	8 itérations
10^{-1000} (~ 1000 décimales)	11 itérations

Démonstration. 1. Dans la preuve de la proposition 3, nous avons vu l'égalité :

$$u_{n+1}^2 - a = \frac{(u_n^2 - a)^2}{4u_n^2} \text{ donc } (u_{n+1} - \sqrt{a})(u_{n+1} + \sqrt{a}) = \frac{(u_n - \sqrt{a})^2(u_n + \sqrt{a})^2}{4u_n^2}$$

Ainsi comme $u_n \geq \sqrt{a}$ pour $n \geq 1$:

$$u_{n+1} - \sqrt{a} = (u_n - \sqrt{a})^2 \times \frac{1}{u_{n+1} + \sqrt{a}} \times \frac{1}{4} \left(1 + \frac{\sqrt{a}}{u_n} \right)^2 \leq (u_n - \sqrt{a})^2 \times \frac{1}{2\sqrt{a}} \times \frac{1}{4} \cdot (1+1)^2 = \frac{1}{2\sqrt{a}} (u_n - \sqrt{a})^2$$

Si k vérifie $u_1 - \sqrt{a} \leq k$, nous allons en déduire par récurrence, pour tout $n \geq 1$, la formule

$$u_n - \sqrt{a} \leq 2\sqrt{a} \left(\frac{k}{2\sqrt{a}} \right)^{2^{n-1}}$$

C'est vrai pour $n = 1$. Supposons la formule vraie au rang n , alors :

$$u_{n+1} - \sqrt{a} \leq \frac{1}{2\sqrt{a}} (u_n - \sqrt{a})^2 = \frac{1}{2\sqrt{a}} (2\sqrt{a})^2 \left(\left(\frac{k}{2\sqrt{a}} \right)^{2^{n-1}} \right)^2 = 2\sqrt{a} \left(\frac{k}{2\sqrt{a}} \right)^{2^n}$$

La formule est donc vrai au rang suivant.

2. Pour $a = 10$ avec $u_0 = 4$ on a $u_1 = 3,25$. Comme $3 \leq \sqrt{10} \leq 4$ alors $u_1 - \sqrt{10} \leq u_1 - 3 \leq \frac{1}{4}$. On fixe donc $k = \frac{1}{4}$. Toujours par l'encadrement $3 \leq \sqrt{10} \leq 4$, la formule obtenue précédemment devient

$$u_n - \sqrt{a} \leq 2 \cdot 4 \left(\frac{\frac{1}{4}}{2 \cdot 3} \right)^{2^{n-1}} = 8 \left(\frac{1}{24} \right)^{2^{n-1}}.$$

□

3.5. Algorithme

Voici l'algorithme pour le calcul de \sqrt{a} . On précise en entrée le réel $a \geq 0$ dont on veut calculer la racine et le nombre n d'itérations.

Code 42 (*newton.py*).

```
def racine_carree(a,n):
    u=4 # N'importe quelle valeur > 0
    for i in range(n):
```

```

u = 0.5*(u+a/u)
return u

```

En utilisant le module `decimal` le calcul de u_n pour $n = 11$ donne 1000 décimales de $\sqrt{10}$:

3,
16227766016837933199889354443271853371955513932521 68268575048527925944386392382213442481083793002951
87347284152840055148548856030453880014690519596700 15390334492165717925994065915015347411333948412408
53169295770904715764610443692578790620378086099418 28371711548406328552999118596824564203326961604691
31433612894979189026652954361267617878135006138818 62785804636831349524780311437693346719738195131856
78403231241795402218308045872844614600253577579702 82864402902440797789603454398916334922265261206779
26516760310484366977937569261557205003698949094694 21850007358348844643882731109289109042348054235653
40390727401978654372593964172600130699000095578446 31096267906944183361301813028945417033158077316263
86395193793704654765220632063686587197822049312426 05345411160935697982813245229700079888352375958532
85792513629646865114976752171234595592380393756251 25369855194955325099947038843990336466165470647234
99979613234340302185705218783667634578951073298287 51579452157716521396263244383990184845609357626020

- Mini-exercices.** 1. À la calculette, calculer les trois premières étapes pour une approximation de $\sqrt{3}$, sous forme de nombres rationnels. Idem avec $\sqrt[3]{2}$.
2. Implémenter la méthode de Newton, étant données une fonction f et sa dérivée f' .
3. Calculer une approximation des solutions de l'équation $x^3 + 1 = 3x$.
4. Soit $a > 0$. Comment calculer $\frac{1}{a}$ par une méthode de Newton ?
5. Calculer n de sorte que $u_n - \sqrt{10} \leq 10^{-\ell}$ (avec $u_0 = 4$, $u_{n+1} = \frac{1}{2} \left(u_n + \frac{a}{u_n} \right)$, $a = 10$).

Cryptographie

- Vidéo ■ partie 1. Le chiffrement de César
- Vidéo ■ partie 2. Le chiffrement de Vigenère
- Vidéo ■ partie 3. La machine Enigma et les clés secrètes
- Vidéo ■ partie 4. La cryptographie à clé publique
- Vidéo ■ partie 5. L'arithmétique pour RSA
- Vidéo ■ partie 6. Le chiffrement RSA

1. Le chiffrement de César

1.1. César a dit...

Jules César a-t-il vraiment prononcé la célèbre phrase :

DOHD MDFWD HVW

ou bien comme le disent deux célèbres Gaulois : « Ils sont fous ces romains ! ».

En fait César, pour ses communications importantes à son armée, cryptait ses messages. Ce que l'on appelle le chiffrement de César est un décalage des lettres : pour crypter un message, **A** devient **D**, **B** devient **E**, **C** devient **F**...

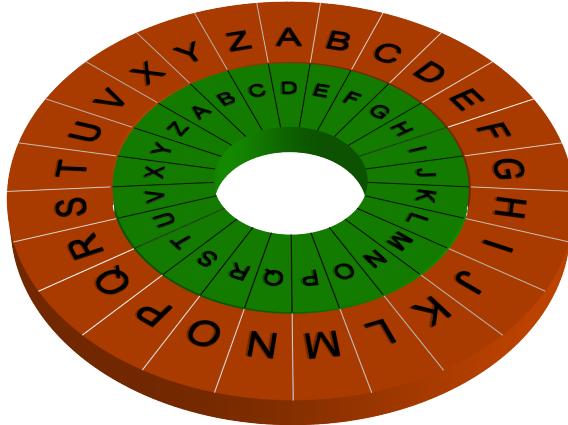
$$A \rightarrow D \quad B \rightarrow E \quad C \rightarrow F \quad \dots \quad W \rightarrow Z \quad X \rightarrow A \quad Y \rightarrow B \quad Z \rightarrow C$$

Voici une figure avec l'alphabet d'origine en haut et en **rouge**, en correspondance avec l'alphabet pour le chiffrement en-dessous et en **vert**.



Nous adopterons la convention suivante, en **vert** c'est la partie du message à laquelle tout le monde a accès (ou qui pourrait être intercepté), c'est donc le message crypté. Alors qu'en **rouge** c'est la partie du message confidentiel, c'est le message en clair.

Pour prendre en compte aussi les dernières lettres de l'alphabet, il est plus judicieux de représenter l'alphabet sur un anneau. Ce décalage est un **décalage circulaire** sur les lettres de l'alphabet.



Pour déchiffrer le message de César, il suffit de décaler les lettres dans l'autre sens, **D** se déchiffre en **A**, **E** en **B**...

Et la célèbre phrase de César est :

ALEA JACTA EST

qui traduite du latin donne « Les dés sont jetés ».

1.2. Des chiffres et des lettres

Il est plus facile de manipuler des nombres que des lettres, aussi nous passons à une formulation arithmétique. Nous associons à chacune des 26 lettres de *A* à *Z* un nombre de 0 à 25. En termes mathématiques, nous définissons une bijection :

$$f : \{A, B, C, \dots, Z\} \longrightarrow \{0, 1, 2, \dots, 25\}$$

par

$$A \longmapsto 0 \quad B \longmapsto 1 \quad C \longmapsto 2 \quad \dots \quad Z \longmapsto 25$$

Ainsi "**A L E A**" devient "**0 11 4 0**".

Le chiffrement de César est un cas particulier de **substitution mono-alphabétique**, c'est-à-dire un chiffrement lettre à lettre.

Quel est l'intérêt ? Nous allons voir que le chiffrement de César correspond à une opération mathématique très simple. Pour cela, rappelons la notion de congruence et l'ensemble $\mathbb{Z}/26\mathbb{Z}$.

1.3. Modulo

Soit $n \geq 2$ un entier fixé.

Définition 1.

On dit que **a est congru à b modulo n** , si n divise $b - a$. On note alors

$$a \equiv b \pmod{n}.$$

Pour nous $n = 26$. Ce qui fait que $28 \equiv 2 \pmod{26}$, car $28 - 2$ est bien divisible par 26. De même $85 = 3 \times 26 + 7$ donc $85 \equiv 7 \pmod{26}$.

On note $\mathbb{Z}/26\mathbb{Z}$ l'ensemble de tous les éléments de \mathbb{Z} modulo 26. Cet ensemble peut par exemple être représenté par les 26 éléments $\{0, 1, 2, \dots, 25\}$. En effet, puisqu'on compte modulo 26 :

$$0, 1, 2, \dots, 25, \quad \text{puis} \quad 26 \equiv 0, 27 \equiv 1, 28 \equiv 2, \dots, 52 \equiv 0, 53 \equiv 1, \dots$$

et de même $-1 \equiv 25, -2 \equiv 24, \dots$

Plus généralement $\mathbb{Z}/n\mathbb{Z}$ contient n éléments. Pour un entier $a \in \mathbb{Z}$ quelconque, son **représentant** dans $\{0, 1, 2, \dots, n-1\}$ s'obtient comme le reste k de la division euclidienne de a par n : $a = bn + k$. De sorte que $a \equiv k \pmod{n}$ et $0 \leq k < n$.

De façon naturelle l'addition et la multiplication d'entiers se transposent dans $\mathbb{Z}/n\mathbb{Z}$.

Pour $a, b \in \mathbb{Z}/n\mathbb{Z}$, on associe $a + b \in \mathbb{Z}/n\mathbb{Z}$.

Par exemple dans $\mathbb{Z}/26\mathbb{Z}$, $15 + 13$ égale 2. En effet $15 + 13 = 28 \equiv 2 \pmod{26}$. Autre exemple : que vaut $133 + 64$? $133 + 64 = 197 = 7 \times 26 + 15 \equiv 15 \pmod{26}$. Mais on pourrait procéder différemment : tout d'abord $133 = 5 \times 26 + 3 \equiv 3 \pmod{26}$ et $64 = 2 \times 26 + 12 \equiv 12 \pmod{26}$. Et maintenant sans calculs : $133 + 64 \equiv 3 + 12 \equiv 15 \pmod{26}$.

On fait de même pour la multiplication : pour $a, b \in \mathbb{Z}/n\mathbb{Z}$, on associe $a \times b \in \mathbb{Z}/n\mathbb{Z}$.

Par exemple 3×12 donne 10 modulo 26, car $3 \times 12 = 36 = 1 \times 26 + 10 \equiv 10 \pmod{26}$. De même : $3 \times 27 = 81 = 3 \times 26 + 3 \equiv 3 \pmod{26}$. Une autre façon de voir la même opération est d'écrire d'abord $27 = 1 \pmod{26}$ puis $3 \times 27 \equiv 3 \times 1 \equiv 3 \pmod{26}$.

1.4. Chiffrer et déchiffrer

Le chiffrement de César est simplement une addition dans $\mathbb{Z}/26\mathbb{Z}$! Fixons un entier k qui est le décalage (par exemple $k = 3$ dans l'exemple de César ci-dessus) et définissons la **fonction de chiffrement de César de décalage k** qui va de l'ensemble $\mathbb{Z}/26\mathbb{Z}$ dans lui-même :

$$C_k : \begin{cases} \mathbb{Z}/26\mathbb{Z} & \longrightarrow \mathbb{Z}/26\mathbb{Z} \\ x & \longmapsto x+k \end{cases}$$

Par exemple, pour $k = 3$: $C_3(0) = 3$, $C_3(1) = 4\dots$

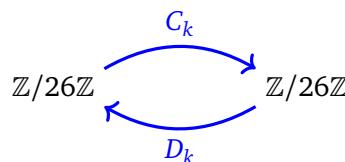
Pour déchiffrer, rien de plus simple ! Il suffit d'aller dans l'autre sens, c'est-à-dire ici de soustraire. La **fonction de déchiffrement de César de décalage k** est

$$D_k : \begin{cases} \mathbb{Z}/26\mathbb{Z} & \longrightarrow \mathbb{Z}/26\mathbb{Z} \\ x & \longmapsto x-k \end{cases}$$

En effet, si 1 a été chiffré en 4, par la fonction C_3 alors $D_3(4) = 4 - 3 = 1$. On retrouve le nombre original. Mathématiquement, D_k est la bijection réciproque de C_k , ce qui implique que pour tout $x \in \mathbb{Z}/26\mathbb{Z}$:

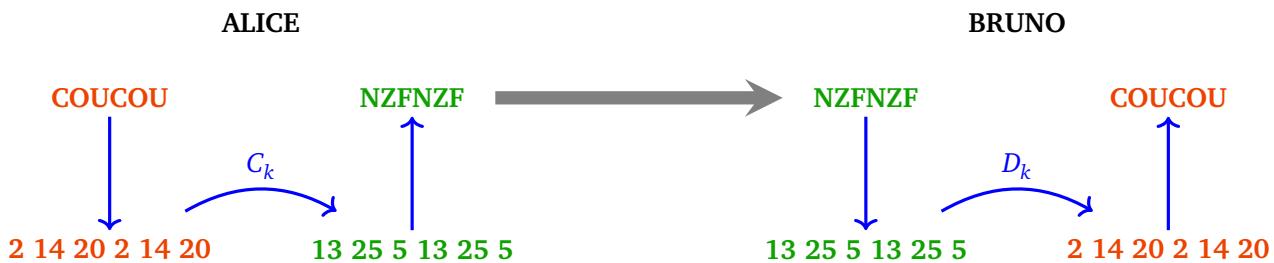
$$D_k(C_k(x)) = x$$

En d'autres termes, si x est un nombre, on applique la fonction de chiffrement pour obtenir le nombre crypté $y = C_k(x)$; ensuite la fonction de déchiffrement fait bien ce que l'on attend d'elle $D_k(y) = x$, on retrouve le nombre original x .



Une autre façon de voir la fonction de déchiffrement est de remarquer que $D_k(x) = C_{-k}(x)$. Par exemple $C_{-3}(x) = x + (-3) \equiv x + 23 \pmod{26}$.

Voici le principe du chiffrement : Alice veut envoyer des messages secrets à Bruno. Ils se sont d'abord mis d'accord sur une clé secrète k , par exemple $k = 11$. Alice veut envoyer le message "**COUCOU**" à Bruno. Elle transforme "**COUCOU**" en "**2 14 20 2 14 20**". Elle applique la fonction de chiffrement $C_{11}(x) = x + 11$ à chacun des nombres : "**13 25 5 13 25 5**" ce qui correspond au mot crypté "**NZFNZF**". Elle transmet le mot crypté à Bruno, qui selon le même principe applique la fonction de déchiffrement $D_{11}(x) = x - 11$.



Exemple 1.

Un exemple classique est le "rot13" (pour rotation par un décalage de 13) :

$$C_{13}(x) = x + 13$$

et comme $-13 \equiv 13 \pmod{26}$ alors $D_{13}(x) = x + 13$. La fonction de déchiffrement est la même que la fonction de chiffrement !

Exemple : déchiffrez le mot "**PRFNE**".

Notons ici deux points importants pour la suite : tout d'abord nous avons naturellement considéré un mot comme une succession de lettres, et chaque opération de chiffrement et déchiffrement s'effectue sur un bloc d'une seule lettre. Ensuite nous avons vu que chiffrer un message est une opération mathématique (certes sur un ensemble un peu spécial).

1.5. Espace des clés et attaque

Combien existe-t-il de possibilités de chiffrement par la méthode de César ? Il y a 26 fonctions C_k différentes, $k = 0, 1, \dots, 25$. Encore une fois, k appartient à $\mathbb{Z}/26\mathbb{Z}$, car par exemple les fonctions C_{29} et C_3 sont identiques. Le décalage k s'appelle la **clé de chiffrement**, c'est l'information nécessaire pour crypter le message. Il y a donc 26 clés différentes et l'**espace des clés** est $\mathbb{Z}/26\mathbb{Z}$.

Il est clair que ce chiffrement de César est d'une sécurité très faible. Si Alice envoie un message secret à Bruno et que Chloé intercepte ce message, il sera facile pour Chloé de le déchiffrer même si elle ne connaît pas la clé secrète k . L'attaque la plus simple pour Chloé est de tester ce que donne chacune des 26 combinaisons possibles et de reconnaître parmi ces combinaisons laquelle donne un message compréhensible.

1.6. Algorithmes

Les ordinateurs ont révolutionné la cryptographie et surtout le décryptage d'un message intercepté. Nous montrons ici, à l'aide du langage Python comment programmer et attaquer le chiffrement de César. Tout d'abord la fonction de chiffrement se programme en une seule ligne :

Code 43 (*cesar.py (1)*).

```
def cesar_chiffre_nb(x, k):
    return (x+k)%26
```

Ici x est un nombre de $\{0, 1, \dots, 25\}$ et k est le décalage. $(x+k)\%26$ a pour valeur le reste modulo 26 de la somme $(x+k)$. Pour le décryptage, c'est aussi simple :

Code 44 (*cesar.py (2)*).

```
def cesar_dechiffre_nb(x, k):
    return (x-k)%26
```

Pour chiffrer un mot ou une phrase, il n'y a pas de problèmes théoriques, mais seulement des difficultés techniques :

- Un mot ou une phrase est une chaîne de caractères, qui en fait se comporte comme une liste. Si `mot` est une chaîne alors `mot[0]` est la première lettre, `mot[1]` la deuxième lettre... et la boucle `for lettre in mot:` permet de parcourir chacune des lettres.
- Pour transformer une lettre en un nombre, on utilise le code Ascii qui à chaque caractère associe un nombre, `ord(A)` vaut 65, `ord(B)` vaut 66... Ainsi `(ord(lettre) - 65)` renvoie le rang de la lettre entre 0 et 25 comme nous l'avons fixé dès le départ.
- La transformation inverse se fait par la fonction `chr` : `chr(65)` renvoie le caractère A, `chr(66)` renvoie B...
- Pour ajouter une lettre à une liste, faites `maliste.append(lettre)`. Enfin pour transformer une liste de caractères en une chaîne, faites `" ".join(maliste)`.

Ce qui donne :

Code 45 (`cesar.py (3)`).

```
def cesar_chiffre_mot(mot,k):
    message_code = []                      # Liste vide
    for lettre in mot:                     # Pour chaque lettre
        nb = ord(lettre)-65                # Lettre devient nb de 0 à 25
        nb_crypté = cesar_chiffre_nb(nb,k)  # Chiffrement de César
        lettre_crypté = chr(nb_crypté+65)   # Retour aux lettres
        message_code.append(lettre_crypté)   # Ajoute lettre au message
    message_code = " ".join(message_code)    # Revient à chaîne caractères
    return(message_code)
```

Pour l'attaque on parcourt l'intégralité de l'espace des clés : k varie de 0 à 25. Noter que pour décrypter les messages on utilise ici simplement la fonction de César avec la clé $-k$.

Code 46 (`cesar.py (4)`).

```
def cesar_attaque(mot):
    for k in range(26):
        print(cesar_chiffre_mot(mot,-k))
    return None
```

2. Le chiffrement de Vigenère

2.1. Substitution mono-alphabétique

Principe

Nous avons vu que le chiffrement de César présente une sécurité très faible, la principale raison est que l'espace des clés est trop petit : il y a seulement 26 clés possibles, et on peut attaquer un message chiffré en testant toutes les clés à la main.

Au lieu de faire correspondre circulairement les lettres, on associe maintenant à chaque lettre une autre lettre (sans ordre fixe ou règle générale).

Par exemple :

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
F	Q	B	M	X	I	T	E	P	A	L	W	H	S	D	O	Z	K	V	G	R	C	N	Y	J	U

Pour crypter le message

ETRE OU NE PAS ETRE TELLE EST LA QUESTION

on regarde la correspondance et on remplace la lettre **E** par la lettre **X**, puis la lettre **T** par la lettre **G**, puis la lettre **R** par la lettre **K**...

Le message crypté est alors :

XGKX DR SX OFV XGKX GXWWX XVG WF ZRXVGPDS

Pour le décrypter, en connaissant les substitutions, on fait l'opération inverse.

Avantage : nous allons voir que l'espace des clés est gigantesque et qu'il n'est plus question d'énumérer toutes les possibilités.

Inconvénients : la clé à retenir est beaucoup plus longue, puisqu'il faut partager la clé constituée des 26 lettres "FQBMX...". Mais surtout, nous allons voir que finalement ce protocole de chiffrement est assez simple à « craquer ».

Espace des clés

Mathématiquement, le choix d'une clé revient au choix d'une bijection de l'ensemble $\{A, B, \dots, Z\}$ vers le même ensemble $\{A, B, \dots, Z\}$. Il y a $26!$ choix possibles. En effet pour la lettre A de l'ensemble de départ, il y a 26 choix possibles (nous avons choisi F), pour B il reste 25 choix possibles (tout sauf F qui est déjà choisi), pour C il reste 24 choix... enfin pour Z il ne reste qu'une seule possibilité, la seule lettre non encore choisie. Au final il y a : $26 \times 25 \times 24 \times \dots \times 2 \times 1$ soit $26!$ choix de clés. Ce qui fait environ 4×10^{26} clés. Il y a plus de clés différentes que de grains de sable sur Terre ! Si un ordinateur pouvait tester 1 milliard de clés par seconde, il lui faudrait alors 12 milliards d'années pour tout énumérer.

Attaque statistique

La principale faiblesse du chiffrement mono-alphabétique est qu'une même lettre est toujours chiffrée de la même façon. Par exemple, ici **E** devient **X**. Dans les textes longs, les lettres n'apparaissent pas avec la même fréquence. Ces fréquences varient suivant la langue utilisée. En français, les lettres les plus rencontrées sont dans l'ordre :

E S A I N T R U L O D C P M V Q G F H B X J Y Z K W

avec les fréquences (souvent proches et dépendant de l'échantillon utilisé) :

E	S	A	I	N	T	R	U	L	O	D
14.69%	8.01%	7.54%	7.18%	6.89%	6.88%	6.49%	6.12%	5.63%	5.29%	3.66%

Voici la méthode d'attaque : dans le texte crypté, on cherche la lettre qui apparaît le plus, et si le texte est assez long cela devrait être le chiffrement du **E**, la lettre qui apparaît ensuite dans l'étude des fréquences devrait être le chiffrement du **S**, puis le chiffrement du **A**... On obtient des morceaux de texte clair sous la forme d'une texte à trous et il faut ensuite deviner les lettres manquantes.

Par exemple, déchiffrons la phrase :

LHLZ HFQ BC HFFPZ WH YOUPFH MUPZH

On compte les apparitions des lettres :

H : 6 F : 4 P : 3 Z : 3

On suppose donc que le **H** crypte la lettre **E**, le **F** la lettre **S**, ce qui donne

E** ES* ** ESS** *E ***SE *E**

D'après les statistiques **P** et **Z** devraient se décrypter en **A** et **I** (ou **I** et **A**). Le quatrième mot "**HFFPZ**", pour l'instant décrypté en "**ESS****", se complète donc en "**ESSAI**" ou "**ESSIA**". La première solution semble correcte ! Ainsi **P** crypte **A**, et **Z** crypte **I**. La phrase est maintenant :

***E*I ES* ** ESSAI *E ***ASE **AIE**

En réfléchissant un petit peu, on décrypte le message :

CECI EST UN ESSAI DE PHRASE VRAIE

2.2. Le chiffrement de Vigenère

Blocs

L'espace des clés du chiffrement mono-alphabétique est immense, mais le fait qu'une lettre soit toujours cryptée de la même façon représente une trop grande faiblesse. Le chiffrement de Vigenère reméde à ce problème. On regroupe les lettres de notre texte par blocs, par exemple ici par blocs de longueur 4 :

CETTE PHRASE NE VEUT RIEN DIRE

devient

CETT EPHR ASEN EVEU TRIE NDIR E

(les espaces sont purement indicatifs, dans la première phrase ils séparent les mots, dans la seconde ils séparent les blocs).

Si k est la longueur d'un bloc, alors on choisit une clé constituée de k nombres de 0 à 25 : (n_1, n_2, \dots, n_k) . Le chiffrement consiste à effectuer un chiffrement de César, dont le décalage dépend du rang de la lettre dans le bloc :

- un décalage de n_1 pour la première lettre de chaque bloc,
- un décalage de n_2 pour la deuxième lettre de chaque bloc,
- ...
- un décalage de n_k pour la k -ème et dernière lettre de chaque bloc.

Pour notre exemple, si on choisit comme clé $(3, 1, 5, 2)$ alors pour le premier bloc "**CETT**" :

- un décalage de 3 pour **C** donne **F**,
- un décalage de 1 pour **E** donne **F**,
- un décalage de 5 pour le premier **T** donne **Y**,
- un décalage de 2 pour le deuxième **T** donne **V**.

Ainsi "**CETT**" de vient "**FFYV**". Vous remarquez que les deux lettres **T** ne sont pas cryptées par la même lettre et que les deux **F** ne cryptent pas la même lettre. On continue ensuite avec le deuxième bloc...

Mathématiques

L'élément de base n'est plus une lettre mais un **bloc**, c'est-à-dire un regroupement de lettres. La fonction de chiffrement associe à un bloc de longueur k , un autre bloc de longueur k , ce qui donne en mathématisant les choses :

$$C_{n_1, n_2, \dots, n_k} : \left\{ \begin{array}{ccc} \mathbb{Z}/26\mathbb{Z} \times \mathbb{Z}/26\mathbb{Z} \times \cdots \times \mathbb{Z}/26\mathbb{Z} & \longrightarrow & \mathbb{Z}/26\mathbb{Z} \times \mathbb{Z}/26\mathbb{Z} \times \cdots \times \mathbb{Z}/26\mathbb{Z} \\ (x_1, x_2, \dots, x_k) & \longmapsto & (x_1 + n_1, x_2 + n_2, \dots, x_k + n_k) \end{array} \right.$$

Chacune des composantes de cette fonction est un chiffrement de César. La fonction de déchiffrement est juste $C_{-n_1, -n_2, \dots, -n_k}$.

Espace des clés et attaque

Il y a 26^k choix possibles de clés, lorsque les blocs sont de longueur k . Pour des blocs de longueur $k = 4$ cela en donne déjà 456 976, et même si un ordinateur teste toutes les combinaisons possibles sans problème, il n'est pas aisés de parcourir cette liste pour trouver le message en clair, c'est-à-dire celui qui est compréhensible ! Il persiste tout de même une faiblesse du même ordre que celle rencontrée dans le chiffrement mono-alphabétique : la lettre **A** n'est pas toujours cryptée par la même lettre, mais si deux lettres **A** sont situées à

la même position dans deux blocs différents (comme par exemple "**ALPH ABET**") alors elles seront cryptées par la même lettre.

Une attaque possible est donc la suivante : on découpe notre message en plusieurs listes, les premières lettres de chaque bloc, les deuxièmes lettres de chaque bloc... et on fait une attaque statistique sur chacun de ces regroupements. Ce type d'attaque n'est possible que si la taille des blocs est petite devant la longueur du texte.

2.3. Algorithmes

Voici un petit algorithme qui calcule la fréquence de chaque lettre d'une phrase.

Code 47 (*statistiques.py*).

```
def statistiques(phrase):
    liste_stat = [0 for x in range(26)]      # Une liste avec des 0
    for lettre in phrase:                    # On parcourt la phrase
        i = ord(lettre)-65
        if 0 <= i < 26:                      # Si c'est une vraie lettre
            liste_stat[i] = liste_stat[i] + 1
    return(liste_stat)
```

Et voici le chiffrement de Vigenère.

Code 48 (*vigenere.py*).

```
def vigenere(mot,cle):                  # Clé est du type [n_1,...,n_k]
    message_code = []
    k = len(cle)                         # Longueur de la clé
    i = 0                                 # Rang dans le bloc
    for lettre in mot:                   # Pour chaque lettre
        nomb = ord(lettre)-65            # Lettre devient nb de 0 à 25
        nomb_code = (nomb+cle[i]) % 26   # Vigenère : on ajoute n_i
        lettre_code = chr(nomb_code+65)  # On repasse aux lettres
        i=(i+1) % k                     # On passe au rang suivant
        message_code.append(lettre_code) # Ajoute lettre au message
    message_code = "".join(message_code) # Revient à chaîne caractères
    return(message_code)
```

3. La machine Enigma et les clés secrètes

3.1. Un secret parfait

L'inconvénient des chiffrements précédents est qu'une même lettre est régulièrement chiffrée de la même façon, car la correspondance d'un alphabet à un ou plusieurs autres est fixée une fois pour toutes, ce qui fait qu'une attaque statistique est toujours possible. Nous allons voir qu'en changeant la correspondance à chaque lettre, il est possible de créer un chiffrement parfait !

Expliquons d'abord le principe à l'aide d'une analogie : j'ai choisi deux entiers m et c tels que $m + c = 100$. Que vaut m ? C'est bien sûr impossible de répondre car il y a plusieurs possibilités : $0 + 100, 1 + 99, 2 + 98, \dots$ Par contre, si je vous donne aussi c alors vous trouvez m immédiatement $m = 100 - c$.

Voici le principe du chiffrement : Alice veut envoyer à Bruno le message secret M suivant :

ATTAQUE LE CHATEAU

Alice a d'abord choisi une clé secrète C qu'elle a transmise à Bruno. Cette clé secrète est de la même longueur que le message (les espaces ne comptent pas) et composée d'entiers de 0 à 25, tirés au hasard. Par exemple C :

$$[4, 18, 2, 0, 21, 12, 18, 13, 7, 11, 23, 22, 19, 2, 16, 9]$$

Elle crypte la première lettre par un décalage de César donné par le premier entier : A est décalé de 4 lettres et devient donc E . La seconde lettre est décalée du second entier : le premier T devient L . Le second T est lui décalé de 2 lettres, il devient V . Le A suivant est décalé de 0 lettre, il reste A ... Alice obtient un message chiffré X qu'elle transmet à Bruno :

EIVALGW YL NEWMGQD

Pour le décrypter, Bruno, qui connaît la clé, n'a qu'à faire le décalage dans l'autre sens.

Notez que deux lettres identiques (par exemple les T) n'ont aucune raison d'être cryptées de la même façon. Par exemple, les T du message initial sont cryptés dans l'ordre par un L , un V et un M .

Formalisons un peu cette opération. On identifie A avec 0, B avec 1, ..., Z avec 25. Alors le message crypté X est juste la "somme" du message M avec la clé secrète C , la somme s'effectuant lettre à lettre, terme à terme, modulo 26.

Notons cette opération $M \oplus C = X$.

$$\begin{array}{cccccccccccccccccccc}
 A & T & T & A & Q & U & E & & L & E & & C & H & A & T & E & A & U \\
 0 & 19 & 19 & 0 & 16 & 20 & 4 & & 11 & 4 & & 2 & 7 & 0 & 19 & 4 & 0 & 20 \\
 \oplus & & & & & & & & & & & & & & & & & \\
 4 & 18 & 2 & 0 & 21 & 12 & 18 & & 13 & 7 & & 11 & 23 & 22 & 19 & 2 & 16 & 9 \\
 \\
 = & 4 & 11 & 21 & 0 & 11 & 6 & 22 & 24 & 11 & & 13 & 4 & 22 & 12 & 6 & 16 & 3 \\
 & E & L & V & A & L & G & W & Y & L & & N & E & W & M & G & Q & D
 \end{array}$$

Bruno reçoit X et connaît C , il effectue donc $X \ominus C = M$.

Pourquoi ce système est-il inviolable ? Pour chacune des lettres, c'est exactement le même problème que trouver m , sachant que $m + c = x$ (où $x = 100$), mais sans connaître c . Toutes les possibilités pour m pourraient être juste. Et bien sûr, dès que l'on connaît c , la solution est triviale : $m = x - c$.

Il y a trois principes à respecter pour que ce système reste inviolable :

1. La longueur de la clé est égale à la longueur du message.
2. La clé est choisie au hasard.
3. La clé ne sert qu'une seule fois.

Ce système appelé "masque jetable" ou chiffrement de Vernam est parfait en théorie, mais sa mise en œuvre n'est pas pratique du tout ! Tout d'abord il faut que la clé soit aussi longue que le message. Pour un message court cela ne pose pas de problème, mais pour envoyer une image par exemple cela devient très lourd. Ensuite, il faut trouver un moyen sûr d'envoyer la clé secrète à son interlocuteur avant de lui faire parvenir le message. Et il faut recommencer cette opération à chaque message, ou bien se mettre d'accord dès le départ sur un *carnet de clés* : une longue liste de clés secrètes.

Pour justifier que ce système est vraiment inviolable voici une expérience amusante : Alice veut envoyer le message $M = \text{"ATTAQUE LE CHATEAU"}$ à Bruno, elle choisit la clé secrète $C = [4, 18, 2, 0, \dots]$ comme ci-dessus et obtient le message chiffré $X = \text{"ELVA..."}$ qu'elle transmet à Bruno.

Alice se fait kidnapper par Chloé, qui veut l'obliger à déchiffrer son message. Heureusement, Alice a anticipé les soucis : elle a détruit le message M , la clé secrète C et a créé un faux message M' et une fausse clé secrète C' . Alice fournit cette fausse clé secrète C' à Chloé, qui déchiffre le message par l'opération $X \ominus C'$

et elle trouve le message bien inoffensif M' :

RECETTE DE CUISINE

Alice est innocentée !

Comment est-ce possible ? Alice avait au préalable préparé un message neutre M' de même longueur que M et calculé la fausse clé secrète $C' = \text{X} \ominus M'$. Chloé a obtenu (par la contrainte) X et C' , elle déchiffre le message ainsi

$$\text{X} \ominus C' = \text{X} \ominus (\text{X} \ominus M') = (\text{X} \ominus \text{X}) \oplus M' = M'$$

Chloé trouve donc le faux message.

Ici la fausse clé C' est :

$$[13, 7, 19, 22, 18, 13, 18, 21, 7, 11, 10, 14, 20, 24, 3, 25]$$

La première lettre du message chiffré est un **E**, en reculant de 13 lettres dans l'alphabet, elle se déchiffre en **R**...

3.2. La machine Enigma

Afin de s'approcher de ce protocole de chiffrement parfait, il faut trouver un moyen de générer facilement de longues clés, comme si elles avaient été générées au hasard. Nous allons étudier deux exemples utilisés en pratique à la fin du siècle dernier, une méthode électro-mécanique : la machine Enigma et une méthode numérique : le DES.

La machine Enigma est une machine électro-mécanique qui ressemble à une machine à écrire. Lorsque qu'une touche est enfoncée, des disques internes sont actionnés et le caractère crypté s'allume. Cette machine, qui sert aussi au déchiffrement, était utilisée pour les communications de l'armée allemande durant la seconde guerre mondiale. Ce que les Allemands ne savaient pas, c'est que les services secrets polonais et britanniques avaient réussi à percer les secrets de cette machine et étaient capables de déchiffrer les messages transmis par les allemands. Ce long travail d'études et de recherches a nécessité tout le génie d'Alan Turing et l'invention de l'ancêtre de l'ordinateur.



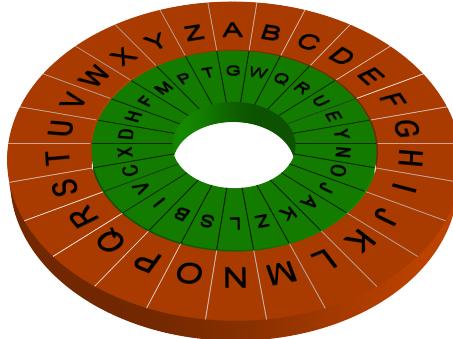
Nous symbolisons l'élément de base de la machine Enigma par deux anneaux :

- Un anneau extérieur contenant l'alphabet "ABCDE..." symbolisant le clavier de saisie des messages. Cet anneau est fixe.

- Un anneau intérieur contenant un alphabet dans le désordre (sur la figure "GWQRU..."). Cet anneau est mobile et effectue une rotation à chaque touche tapée au clavier. Il représente la clé secrète.

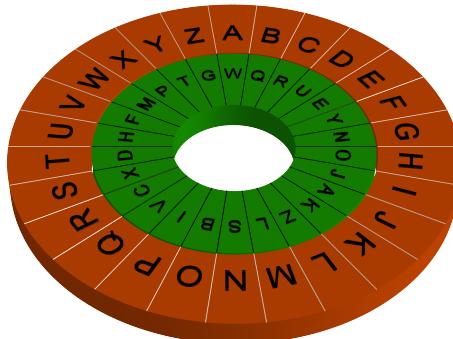
Voici, dans ce cas, le processus de chiffrement du mot "**BAC**", avec la clé de chiffrement "**G**" :

- Position initiale.** L'opérateur tourne l'anneau intérieur de sorte que le **A** extérieur et fixe soit en face du **G** intérieur (et donc **B** en face de **W**).



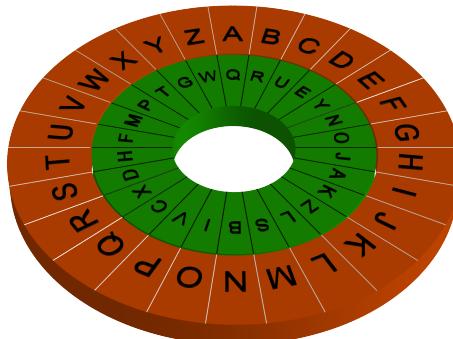
- Première lettre.** L'opérateur tape la première lettre du message : **B**, la machine affiche la correspondance **W**.

- Rotation.** L'anneau intérieur tourne de 1/26ème de tour, maintenant le **A** extérieur et fixe est en face du **W**, le **B** en face du **Q**,...



- Deuxième lettre.** L'opérateur tape la deuxième lettre du message **A**, la machine affiche la correspondance, c'est de nouveau **W**.

- Rotation.** L'anneau intérieur tourne de 1/26ème de tour, maintenant le **A** extérieur et fixe est en face du **Q**, le **B** en face du **R**, le **C** en face du **U**,...



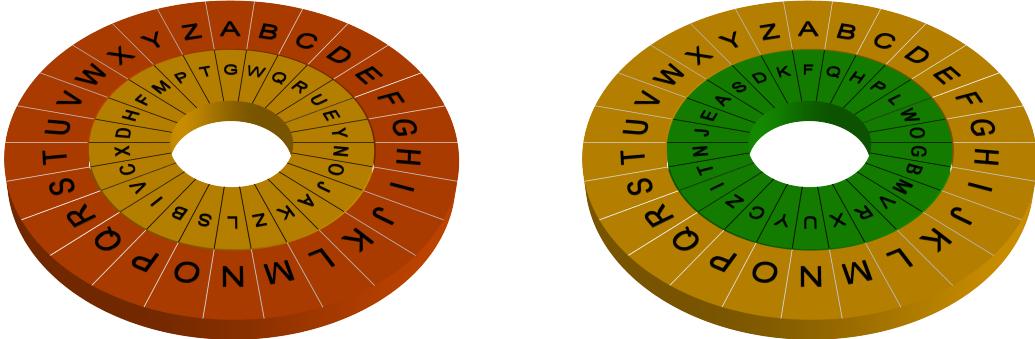
- Troisième lettre.** L'opérateur tape la troisième lettre du message **C**, la machine affiche la correspondance **U**.

- Rotation.** L'anneau intérieur effectue sa rotation.

- Message chiffré.** Le message crypté est donc "**WWU**"

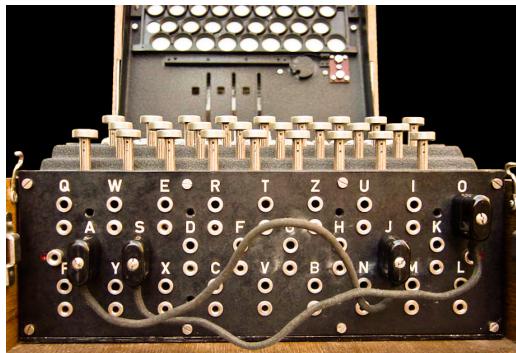
Cette méthode de chiffrement est identique à un chiffrement de type Vigenère pour une clé de longueur 26. Il y a 26 clés différents à disposition avec un seul anneau intérieur et identifiées par lettre de la position initiale : **G, W, Q... T** correspondant aux alphabets : "**GWQ...PT**", "**WQR...TG**", "**QRU...GW**"...

En fait, la machine Enigma était beaucoup plus sophistiquée, il n'y avait pas un mais plusieurs anneaux intérieurs. Par exemple pour deux anneaux intérieurs comme sur la figure : **B** s'envoie sur **W**, qui s'envoie sur **A**; la lettre **B** est cryptée en **A**. Ensuite l'anneau intérieur numéro 1 effectue 1/26ème de tour. La lettre **A** s'envoie sur **W**, qui s'envoie sur **A**; la lettre **A** est cryptée en **A**. Lorsque l'anneau intérieur numéro 1 a fait une rotation complète (26 lettres ont été tapées) alors l'anneau intérieur numéro 2 effectue 1/26ème de tour. C'est comme sur un compteur kilométrique, lorsque le chiffre des kilomètres parcourt 0, 1, 2, 3, ..., 9, alors au kilomètre suivant, le chiffre des dizaines de kilomètres est 0 et celui des unités est augmenté d'une unité.



S'il y a trois anneaux, lorsque l'anneau intérieur 2 a fait une rotation complète, l'anneau intérieur 3 tourne de 1/26ème de tour. Il y a alors 26^3 clés différentes facilement identifiables par les trois lettres des positions initiales des anneaux.

Il fallait donc pour utiliser cette machine, d'abord choisir les disques (nos anneaux intérieurs) les placer dans un certain ordre, fixer la position initiale de chaque disque. Ce système était rendu largement plus complexe avec l'ajout de correspondances par fichage entre les lettres du clavier (voir photo). Le nombre de clés possibles dépassait plusieurs milliards de milliards !



3.3. La ronde des chiffres : DES

La machine Enigma génère mécaniquement un alphabet différent à chaque caractère crypté, tentant de se rapprocher d'un chiffrement parfait. Nous allons voir une autre méthode, cette fois numérique : le DES. Le DES (*Data Encryption Standard*) est un protocole de chiffrement par blocs. Il a été, entre 1977 et 2001, le standard de chiffrement pour les organisations du gouvernement des États-Unis et par extension pour un grand nombre de pays dans le monde.

Commençons par rappeler que l'objectif est de générer une clé aléatoire de grande longueur. Pour ne pas avoir à retenir l'intégralité de cette longue clé, on va la générer de façon pseudo-aléatoire à partir d'une petite clé.

Voyons un exemple élémentaire de suite pseudo-aléatoire.

Soit (u_n) la suite définie par la donnée de (a, b) et de u_0 et la relation de récurrence

$$u_{n+1} \equiv a \times u_n + b \pmod{26}.$$

Par exemple pour $a = 2$, $b = 5$ et $u_0 = 6$, alors les premiers termes de la suites sont :

$$6 \quad 17 \quad 13 \quad 5 \quad 15 \quad 9 \quad 23 \quad 25 \quad 3 \quad 11 \quad 1 \quad 7 \quad 19 \quad 17 \quad 13 \quad 5$$

Les trois nombres (a, b, u_0) représentent la clé principale et la suite des $(u_n)_{n \in \mathbb{N}}$ les clés secondaires.

Avantages : à partir d'une clé principale courte (ici trois nombres) on a générée une longue liste de clés secondaires. Inconvénients : la liste n'est pas si aléatoire que cela, elle se répète ici avec une période de longueur 12 : 17, 13, 5, ..., 17, 13, 5, ...

Le système DES est une version sophistiquée de ce processus : à partir d'une clé courte et d'opérations élémentaires on crypte un message. Comme lors de l'étude de la machine Enigma, nous allons présenter une version très simplifiée de ce protocole afin d'en expliquer les étapes élémentaires.

Pour changer, nous allons travailler modulo 10. Lorsque l'on travaille par blocs, les additions se font *bit par bit*. Par exemple : $[1 \ 2 \ 3 \ 4] \oplus [7 \ 8 \ 9 \ 0] = [8 \ 0 \ 2 \ 4]$ car $(1 + 7 \equiv 8 \pmod{10})$, $2 + 8 \equiv 0 \pmod{10}$, ...)

Notre message est coupé en blocs, pour nos explications ce seront des blocs de longueur 8. La clé est de longueur 4.

Voici le message (un seul bloc) : $M = [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8]$ et voici la clé : $C = [3 \ 1 \ 3 \ 2]$.

Étape 0. Initialisation. On note $M_0 = M$ et on découpe M en une partie gauche et une partie droite

$$M_0 = [G_0 \parallel D_0] = [1 \ 2 \ 3 \ 4 \parallel 5 \ 6 \ 7 \ 8]$$

Étape 1. Premier tour. On pose

$$M_1 = [D_0 \parallel C \oplus \sigma(G_0)]$$

où σ est une permutation circulaire.

On effectue donc trois opérations pour passer de M_0 à M_1 :

1. On échange la partie droite et la partie gauche de M_0 :

$$M_0 \longmapsto [5 \ 6 \ 7 \ 8 \parallel 1 \ 2 \ 3 \ 4]$$

2. Sur la nouvelle partie droite, on permute circulairement les nombres :

$$\longmapsto [5 \ 6 \ 7 \ 8 \parallel 2 \ 3 \ 4 \ 1]$$

3. Puis on ajoute la clé secrète C à droite (ici $C = [3 \ 1 \ 3 \ 2]$) :

$$\longmapsto [5 \ 6 \ 7 \ 8 \parallel 5 \ 4 \ 7 \ 3] = M_1$$

On va recommencer le même processus. Cela revient à appliquer la formule de récurrence, qui partant de $M_i = [G_i \parallel D_i]$, définit

$$M_{i+1} = [D_i \parallel C \oplus \sigma(G_i)]$$

Étape 2. Deuxième tour. On part de $M_1 = [5 \ 6 \ 7 \ 8 \parallel 5 \ 4 \ 7 \ 3]$.

1. On échange la partie droite et la partie gauche de M_0 :

$$M_0 \longmapsto [5 \ 4 \ 7 \ 3 \parallel 5 \ 6 \ 7 \ 8]$$

2. Sur la nouvelle partie droite, on permute circulairement les nombres.

$$\longmapsto [5 \ 4 \ 7 \ 3 \parallel 6 \ 7 \ 8 \ 5]$$

3. Puis on ajoute la clé secrète C à droite.

$$\longmapsto [5 \ 4 \ 7 \ 3 \parallel 9 \ 8 \ 1 \ 7] = M_2$$

On peut décider de s'arrêter après ce tour et renvoyer le message crypté $X = M_2 = [5 \ 4 \ 7 \ 3 \ 9 \ 8 \ 1 \ 7]$. Comme chaque opération élémentaire est inversible, on applique un protocole inverse pour déchiffrer. Dans le vrai protocole du DES, les blocs sont de taille 64 bits, il y a plus de manipulations sur le message et les étapes mentionnées ci-dessus sont effectuées 16 fois (on parle de tours). À chaque tour, une clé différente est utilisée. Il existe donc un préambule à ce protocole : générer 16 clés secondaires (de longueur 48 bits) à partir de la clé principale, ce qui se fait selon le principe de la suite pseudo-aléatoire (u_n) expliquée plus haut.

4. La cryptographie à clé publique

Les Grecs pour envoyer des messages secrets rasaient la tête du messager, tatouaient le message sur son crâne et attendaient que les cheveux repoussent avant d'envoyer le messager effectuer sa mission ! Il est clair que ce principe repose uniquement sur le secret de la méthode.

4.1. Le principe de Kerckhoffs

Cette méthode rudimentaire va à l'encontre du principe de Kerckhoffs. Le principe de Kerckhoffs s'énonce ainsi :

«La sécurité d'un système de chiffrement ne doit reposer que sur le secret de la clé.»

Cela se résume aussi par :

«L'ennemi peut avoir connaissance du système de chiffrement.»

Voici le texte original d'Auguste Kerckhoffs de 1883 «La cryptographie militaire» paru dans le *Journal des sciences militaires*.

Il traite notamment des enjeux de sécurité lors des correspondances :

«Il faut distinguer entre un système d'écriture chiffré, imaginé pour un échange momentané de lettres entre quelques personnes isolées, et une méthode de cryptographie destinée à régler pour un temps illimité la correspondance des différents chefs d'armée entre eux.»

Le principe fondamental est le suivant :

«Dans le second cas, [...] il faut que le système n'exige pas le secret, et qu'il puisse sans inconveniente tomber entre les mains de l'ennemi.»

Ce principe est novateur dans la mesure où intuitivement il semble opportun de dissimuler le maximum de choses possibles : clé et système de chiffrement utilisés. Mais l'objectif visé par Kerckhoffs est plus académique, il pense qu'un système dépendant d'un secret mais dont le mécanisme est connu de tous sera testé, attaqué, étudié, et finalement utilisé s'il s'avère intéressant et robuste.

4.2. Factorisations des entiers

Quels outils mathématiques répondent au principe de Kerckhoffs ?

Un premier exemple est la toute simple multiplication ! En effet si je vous demande combien font 5×7 , vous répondez 35. Si je vous demande de factoriser 35 vous répondez 5×7 . Cependant ces deux questions ne sont pas du même ordre de difficulté. Si je vous demande de factoriser 1591, vous aller devoir faire plusieurs tentatives, alors que si je vous avais directement demandé de calculer 37×43 cela ne pose pas de problème.

Pour des entiers de plusieurs centaines de chiffres le problème de factorisation ne peut être résolu en un temps raisonnable, même pour un ordinateur. C'est ce problème asymétrique qui est à la base de la

cryptographie RSA (que nous détaillerons plus tard) : connaître p et q apporte plus d'information utilisable que $p \times q$. Même si en théorie à partir de $p \times q$ on peut retrouver p et q , en pratique ce sera impossible.

Formalisons ceci avec la notion de complexité. La **complexité** est le temps de calculs (ou le nombre d'opérations élémentaires) nécessaire pour effectuer une opération.

Commençons par la complexité de l'addition : disons que calculer la somme de deux chiffres (par exemple $6 + 8$) soit de complexité 1 (par exemple 1 seconde pour un humain, 1 milliseconde pour un ordinateur). Pour calculer la somme de deux entiers à n chiffres, la complexité est d'ordre n (exemple : $1234 + 2323$, il faut faire 4 additions de chiffres, donc environ 4 secondes pour un humain).

La multiplication de deux entiers à n chiffres est de complexité d'ordre n^2 . Par exemple pour multiplier 1234 par 2323 il faut faire 16 multiplications de chiffres (chaque chiffre de 1234 est à multiplier par chaque chiffre de 2323).

Par contre la meilleure méthode de factorisation connue est de complexité d'ordre $\exp(4n^{1/3})$ (c'est moins que $\exp(n)$, mais plus que n^d pour tout d , lorsque n tend vers $+\infty$).

Voici un tableau pour avoir une idée de la difficulté croissante pour multiplier et factoriser des nombres à n chiffres :

n	multiplication	factorisation
3	9	320
4	16	572
5	25	934
10	100	5 528
50	2 500	2 510 835
100	10 000	115 681 968
200	40 000	14 423 748 780

4.3. Fonctions à sens unique

Il existe bien d'autres situations mathématiques asymétriques : les **fonctions à sens unique**. En d'autres termes, étant donnée une fonction f , il est possible connaissant x de calculer «facilement» $f(x)$; mais connaissant un élément de l'ensemble image de f , il est «difficile» ou impossible de trouver son antécédent. Dans le cadre de la cryptographie, posséder une fonction à sens unique qui joue le rôle de chiffrement n'a que peu de sens. En effet, il est indispensable de trouver un moyen efficace afin de pouvoir déchiffrer les messages chiffrés. On parle alors de **fonction à sens unique avec trappe secrète**.

Prenons par exemple le cas de la fonction f suivante :

$$f : x \mapsto x^3 \pmod{100}.$$

- Connaissant x , trouver $y = f(x)$ est facile, cela nécessite deux multiplications et deux divisions.
- Connaissant y image par f d'un élément x ($y = f(x)$), retrouver x est difficile.

Tentons de résoudre le problème suivant : trouver x tel que $x^3 \equiv 11 \pmod{100}$.

On peut pour cela :

- soit faire une recherche exhaustive, c'est-à-dire essayer successivement 1, 2, 3, ..., 99, on trouve alors :

$$71^3 = 357\,911 \equiv 11 \pmod{100},$$

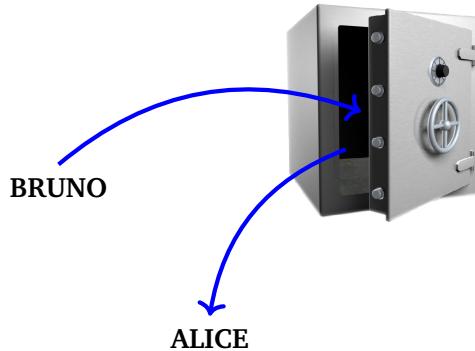
- soit utiliser la trappe secrète : $y \mapsto y^7 \pmod{100}$ qui fournit directement le résultat !

$$11^7 = 19\,487\,171 \equiv 71 \pmod{100}.$$

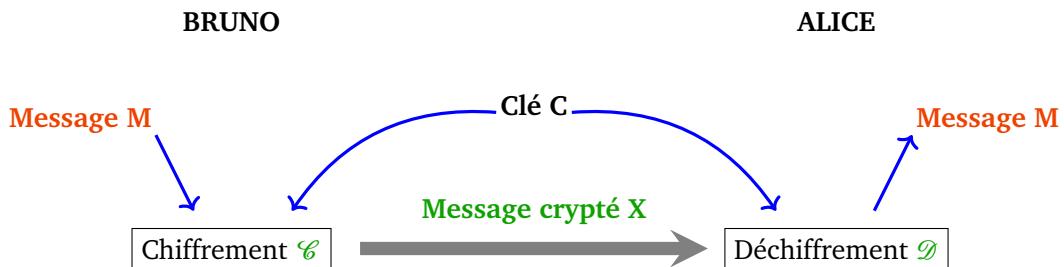
La morale est la suivante : le problème est dur à résoudre, sauf pour ceux qui connaissent la trappe secrète. (Attention, dans le cas de cet exemple, la fonction f n'est pas bijective.)

4.4. Chiffrement à clé secrète

Petit retour en arrière. Les protocoles étudiés dans les chapitres précédents étaient des **chiffrements à clé secrète**. De façon imagée, tout se passe comme si Bruno pouvaient déposer son message dans un coffre fort pour Alice, Alice et Bruno étant les seuls à posséder la clé du coffre.

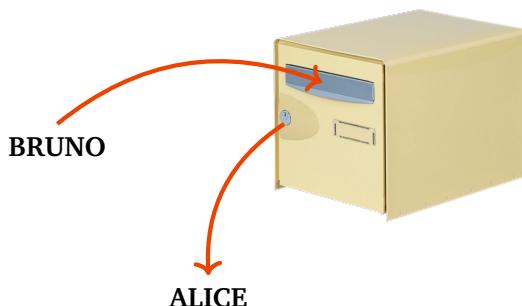


En effet, jusqu'ici, les deux interlocuteurs se partageaient une même clé qui servait à chiffrer (et déchiffrer) les messages. Cela pose bien sûr un problème majeur : Alice et Bruno doivent d'abord se communiquer la clé.

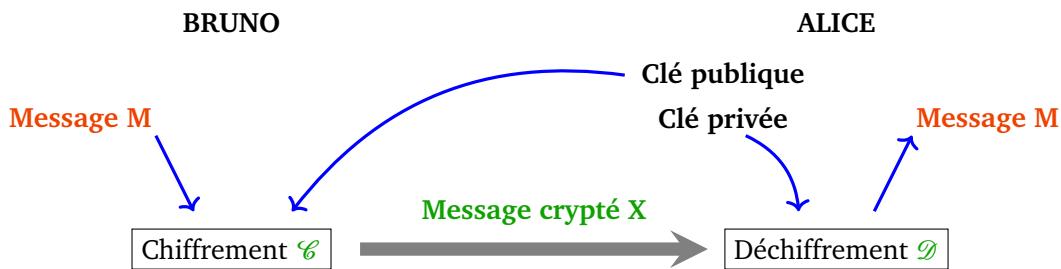


4.5. Chiffrement à clé publique

Les fonctions à sens unique à trappe donnent naissance à des protocoles de chiffrement à clé publique. L'association «clé» et «publique» peut paraître incongrue, mais il signifie que le principe de chiffrement est accessible à tous mais que le déchiffrement nécessite une clé qu'il faut bien sûr garder secrète.



De façon imagée, si Bruno veut envoyer un message à Alice, il dépose son message dans la boîte aux lettres d'Alice, seule Alice pourra ouvrir sa boîte et consulter le message. Ici la clé publique est symbolisée par la boîte aux lettres, tout le monde peut y déposer un message, la clé qui ouvre la boîte aux lettres est la clé privée d'Alice, que Alice doit conserver à l'abri.



En prenant appui sur l'exemple précédent, si le message initial est 71 et que la fonction f de chiffrement est connue de tous, le message transmis est 11 et le déchiffrement sera rapide si la trappe secrète 7 est connue du destinataire.

Les paramètres d'un protocole de **chiffrement à clé publique** sont donc :

- les fonctions de chiffrement et de déchiffrement : C et D ,
- la clé publique du destinataire qui va permettre de paramétriser la fonction C ,
- la clé privée du destinataire qui va permettre de paramétriser la fonction D .

Dans le cadre de notre exemple Bruno souhaite envoyer un message à Alice, ces éléments sont :

- $C : x \mapsto x^3 \pmod{100}$ et $D : x \mapsto x^7 \pmod{100}$,
- **3** : la clé publique d'Alice qui permet de définir complètement la fonction de chiffrement :

$$C : x \mapsto x^3 \pmod{100},$$

- **7** : la clé privée d'Alice qui permet de définir complètement la fonction de déchiffrement :

$$D : x \mapsto x^7 \pmod{100}.$$

Dans la pratique, un chiffrement à clé publique nécessite plus de calculs et est donc assez lent, plus lent qu'un chiffrement à clé privée. Afin de gagner en rapidité, un protocole hybride peut être mis en place de la façon suivante :

- à l'aide d'un protocole de chiffrement à clé publique, Alice et Bruno échangent une clé,
- Alice et Bruno utilisent cette clé dans un protocole de chiffrement à clé secrète.

5. L'arithmétique pour RSA

Pour un entier n , sachant qu'il est le produit de deux nombres premiers, il est difficile de retrouver les facteurs p et q tels que $n = pq$. Le principe du chiffrement RSA, chiffrement à clé publique, repose sur cette difficulté.

Dans cette partie nous mettons en place les outils mathématiques nécessaires pour le calcul des clés publique et privée ainsi que les procédés de chiffrement et déchiffrement RSA.

5.1. Le petit théorème de Fermat amélioré

Nous connaissons le petit théorème de Fermat

Théorème 1 (Petit théorème de Fermat).

Si p est un nombre premier et $a \in \mathbb{Z}$ alors

$$a^p \equiv a \pmod{p}$$

et sa variante :

Corollaire 1.

Si p ne divise pas a alors

$$a^{p-1} \equiv 1 \pmod{p}$$

Nous allons voir une version améliorée de ce théorème dans le cas qui nous intéresse :

Théorème 2 (Petit théorème de Fermat amélioré).

Soient p et q deux nombres premiers distincts et soit $n = pq$. Pour tout $a \in \mathbb{Z}$ tel que $\text{pgcd}(a, n) = 1$ alors :

$$a^{(p-1)(q-1)} \equiv 1 \pmod{n}$$

On note $\varphi(n) = (p-1)(q-1)$, la **fonction d'Euler**. L'hypothèse $\text{pgcd}(a, n) = 1$ équivaut ici à ce que a ne soit divisible ni par p , ni par q . Par exemple pour $p = 5, q = 7, n = 35$ et $\varphi(n) = 4 \cdot 6 = 24$. Alors pour $a = 1, 2, 3, 4, 6, 8, 9, 11, 12, 13, 16, 17, 18, \dots$ on a bien $a^{24} \equiv 1 \pmod{35}$.

Démonstration. Notons $c = a^{(p-1)(q-1)}$. Calculons c modulo p :

$$c \equiv a^{(p-1)(q-1)} \equiv (a^{p-1})^{q-1} \equiv 1^{q-1} \equiv 1 \pmod{p}$$

où l'on applique le petit théorème de Fermat : $a^{p-1} \equiv 1 \pmod{p}$, car p ne divise pas a .

Calculons ce même c mais cette fois modulo q :

$$c \equiv a^{(p-1)(q-1)} \equiv (a^{q-1})^{p-1} \equiv 1^{p-1} \equiv 1 \pmod{q}$$

où l'on applique le petit théorème de Fermat : $a^{q-1} \equiv 1 \pmod{q}$, car q ne divise pas a .

Conclusion partielle : $c \equiv 1 \pmod{p}$ et $c \equiv 1 \pmod{q}$.

Nous allons en déduire que $c \equiv 1 \pmod{pq}$.

Comme $c \equiv 1 \pmod{p}$ alors il existe $\alpha \in \mathbb{Z}$ tel que $c = 1 + \alpha p$; comme $c \equiv 1 \pmod{q}$ alors il existe $\beta \in \mathbb{Z}$ tel que $c = 1 + \beta q$. Donc $c - 1 = \alpha p = \beta q$. De l'égalité $\alpha p = \beta q$, on tire que $p \mid \beta q$.

Comme p et q sont premiers entre eux (car ce sont des nombres premiers distincts) alors par le lemme de Gauss on en déduit que $p \mid \beta$. Il existe donc $\beta' \in \mathbb{Z}$ tel que $\beta = \beta' p$.

Ainsi $c = 1 + \beta q = 1 + \beta' p q$. Ce qui fait que $c \equiv 1 \pmod{pq}$, c'est exactement dire $a^{(p-1)(q-1)} \equiv 1 \pmod{n}$. \square

5.2. L'algorithme d'Euclide étendu

Nous avons déjà étudié l'algorithme d'Euclide qui repose sur le principe que $\text{pgcd}(a, b) = \text{pgcd}(b, a \pmod{b})$. Voici sa mise en œuvre informatique.

Code 49 (*euclide.py* (1)).

```
def euclide(a,b):
    while b !=0 :
        a , b = b , a % b
    return a
```

On profite que Python assure les affectations simultanées, ce qui pour nous correspond aux suites

$$\begin{cases} a_{i+1} = b_i \\ b_{i+1} \equiv a_i \pmod{b_i} \end{cases}$$

initialisée par $a_0 = a, b_0 = b$.

Nous avons vu aussi comment « remonter » l'algorithme d'Euclide à la main pour obtenir les coefficients de Bézout u, v tels que $au + bv = \text{pgcd}(a, b)$. Cependant il nous faut une méthode plus automatique pour obtenir ces coefficients, c'est l'**algorithme d'Euclide étendu**.

On définit deux suites $(x_i), (y_i)$ qui vont aboutir aux coefficients de Bézout.

L'initialisation est :

$$x_0 = 1 \quad x_1 = 0 \quad y_0 = 0 \quad y_1 = 1$$

et la formule de récurrence pour $i \geq 1$:

$$x_{i+1} = x_{i-1} - q_i x_i \quad y_{i+1} = y_{i-1} - q_i y_i$$

où q_i est le quotient de la division euclidienne de a_i par b_i .

Code 50 (*euclide.py (2)*).

```
def euclide_etendu(a,b):
    x = 1 ; xx = 0
    y = 0 ; yy = 1
    while b != 0 :
        q = a // b
        a , b = b , a % b
        xx , x = x - q*xx , xx
        yy , y = y - q*yy , yy
    return (a,x,y)
```

Cet algorithme renvoie d'abord le pgcd, puis les coefficients u, v tels que $au + bv = \text{pgcd}(a, b)$.

5.3. Inverse modulo n

Soit $a \in \mathbb{Z}$, on dit que $x \in \mathbb{Z}$ est un **inverse de a modulo n** si $ax \equiv 1 \pmod{n}$.

Trouver un inverse de a modulo n est donc un cas particulier de l'équation $ax \equiv b \pmod{n}$.

Proposition 1.

- a admet un inverse modulo n si et seulement si a et n sont premiers entre eux.
- Si $au + nv = 1$ alors u est un inverse de a modulo n .

En d'autres termes, trouver un inverse de a modulo n revient à calculer les coefficients de Bézout associés à la paire (a, n) .

Démonstration. La preuve est essentiellement une reformulation du théorème de Bézout :

$$\begin{aligned} \text{pgcd}(a, n) = 1 &\iff \exists u, v \in \mathbb{Z} \quad au + nv = 1 \\ &\iff \exists u \in \mathbb{Z} \quad au \equiv 1 \pmod{n} \end{aligned}$$

□

Voici le code :

Code 51 (*euclide.py (3)*).

```
def inverse(a,n):
    c,u,v = euclide_etendu(a,n)      # pgcd et coeff. de Bézout
    if c != 1 :                      # Si pgcd différent de 1 renvoie 0
        return 0
    else :
        return u % n                # Renvoie l'inverse
```

5.4. L'exponentiation rapide

Nous aurons besoin de calculer rapidement des puissances modulo n . Pour cela il existe une méthode beaucoup plus efficace que de calculer d'abord a^k puis de le réduire modulo n . Il faut garder à l'esprit que les entiers que l'on va manipuler ont des dizaines voire des centaines de chiffres.

Voyons la technique sur l'exemple de $5^{11} \pmod{14}$. L'idée est de seulement calculer $5, 5^2, 5^4, 5^8 \dots$ et de réduire modulo n à chaque fois. Pour cela on remarque que $11 = 8 + 2 + 1$ donc

$$5^{11} = 5^8 \times 5^2 \times 5^1.$$

Calculons donc les $5^{2^i} \pmod{14}$:

$$\begin{aligned} 5 &\equiv 5 \pmod{14} \\ 5^2 &\equiv 25 \equiv 11 \pmod{14} \\ 5^4 &\equiv 5^2 \times 5^2 \equiv 11 \times 11 \equiv 121 \equiv 9 \pmod{14} \\ 5^8 &\equiv 5^4 \times 5^4 \equiv 9 \times 9 \equiv 81 \equiv 11 \pmod{14} \end{aligned}$$

à chaque étape est effectuée une multiplication modulaire. Conséquence :

$$5^{11} \equiv 5^8 \times 5^2 \times 5^1 \equiv 11 \times 11 \times 5 \equiv 11 \times 55 \equiv 11 \times 13 \equiv 143 \equiv 3 \pmod{14}.$$

Nous obtenons donc un calcul de $5^{11} \pmod{14}$ en 5 opérations au lieu de 10 si on avait fait $5 \times 5 \times 5 \dots$. Voici une formulation générale de la méthode. On écrit le développement de l'exposant k en base 2 : $(k_\ell, \dots, k_2, k_1, k_0)$ avec $k_i \in \{0, 1\}$ de sorte que

$$k = \sum_{i=0}^{\ell} k_i 2^i.$$

On obtient alors

$$x^k = x^{\sum_{i=0}^{\ell} k_i 2^i} = \prod_{i=0}^{\ell} (x^{2^i})^{k_i}.$$

Par exemple 11 en base 2 s'écrit (1, 0, 1, 1), donc, comme on l'a vu :

$$5^{11} = (5^{2^3})^1 \times (5^{2^2})^0 \times (5^{2^1})^1 \times (5^{2^0})^1.$$

Voici un autre exemple : calculons $17^{154} \pmod{100}$. Tout d'abord on décompose l'exposant $k = 154$ en base 2 : $154 = 128 + 16 + 8 + 2 = 2^7 + 2^4 + 2^3 + 2^1$, il s'écrit donc en base 2 : (1, 0, 0, 1, 1, 0, 1, 0).

Ensuite on calcule $17, 17^2, 17^4, 17^8, \dots, 17^{128}$ modulo 100.

$$\begin{aligned} 17 &\equiv 17 \pmod{100} \\ 17^2 &\equiv 17 \times 17 \equiv 289 \equiv 89 \pmod{100} \\ 17^4 &\equiv 17^2 \times 17^2 \equiv 89 \times 89 \equiv 7921 \equiv 21 \pmod{100} \\ 17^8 &\equiv 17^4 \times 17^4 \equiv 21 \times 21 \equiv 441 \equiv 41 \pmod{100} \\ 17^{16} &\equiv 17^8 \times 17^8 \equiv 41 \times 41 \equiv 1681 \equiv 81 \pmod{100} \\ 17^{32} &\equiv 17^{16} \times 17^{16} \equiv 81 \times 81 \equiv 6561 \equiv 61 \pmod{100} \\ 17^{64} &\equiv 17^{32} \times 17^{32} \equiv 61 \times 61 \equiv 3721 \equiv 21 \pmod{100} \\ 17^{128} &\equiv 17^{64} \times 17^{64} \equiv 21 \times 21 \equiv 441 \equiv 41 \pmod{100} \end{aligned}$$

Il ne reste qu'à rassembler :

$$17^{154} \equiv 17^{128} \times 17^{16} \times 17^8 \times 17^2 \equiv 41 \times 81 \times 41 \times 89 \equiv 3321 \times 3649 \equiv 21 \times 49 \equiv 1029 \equiv 29 \pmod{100}$$

On en déduit un algorithme pour le calcul rapide des puissances.

Code 52 (*puissance.py*).

```
def puissance(x,k,n):
    puiss = 1 # Résultat
    while (k>0):
        if k % 2 != 0 : # Si k est impair (i.e. k_i=1)
            puiss = (puiss*x) % n
        x = x*x % n # Vaut x, x^2, x^4, ...
        k = k // 2
    return(puiss)
```

En fait Python sait faire l'exponentiation rapide : `pow(x, k, n)` pour le calcul de a^k modulo n , il faut donc éviter `(x ** k) % n` qui n'est pas adapté.

6. Le chiffrement RSA

Voici le but ultime de ce cours : la chiffrement RSA. Il est temps de relire l'introduction du chapitre « Arithmétique » pour s'apercevoir que nous sommes prêts !

Pour crypter un message on commence par le transformer en un –ou plusieurs– nombres.

Les processus de chiffrement et déchiffrement font appel à plusieurs notions :

- *On choisit deux **nombres premiers** p et q que l'on garde secrets et on pose $n = p \times q$. Le principe étant que même connaissant n il est très difficile de retrouver p et q (qui sont des nombres ayant des centaines de chiffres).*
- *La clé secrète et la clé publique se calculent à l'aide de l'**algorithme d'Euclide** et des **coefficients de Bézout**.*
- *Les calculs de cryptage se feront **modulo** n .*
- *Le déchiffrement fonctionne grâce à une variante du **petit théorème de Fermat**.*

Dans cette section, c'est Bruno qui veut envoyer un message secret à Alice. La processus se décompose ainsi :

1. Alice prépare une clé publique et une clé privée,
2. Bruno utilise la clé publique d'Alice pour crypter son message,
3. Alice reçoit le message crypté et le déchiffre grâce à sa clé privée.

6.1. Calcul de la clé publique et de la clé privée

Choix de deux nombres premiers

Alice effectue, une fois pour toute, les opérations suivantes (en secret) :

- elle choisit deux nombres premiers distincts p et q (dans la pratique ce sont de très grand nombres, jusqu'à des centaines de chiffres),
- Elle calcule $n = p \times q$,
- Elle calcule $\varphi(n) = (p - 1) \times (q - 1)$.

Exemple 1.

- $p = 5$ et $q = 17$
- $n = p \times q = 85$
- $\varphi(n) = (p - 1) \times (q - 1) = 64$

Vous noterez que le calcul de $\varphi(n)$ n'est possible que si la décomposition de n sous la forme $p \times q$ est connue. D'où le caractère secret de $\varphi(n)$ même si n est connu de tous.

Exemple 2.

- $p = 101$ et $q = 103$
- $n = p \times q = 10\,403$
- $\varphi(n) = (p - 1) \times (q - 1) = 10\,200$

Choix d'un exposant et calcul de son inverse

Alice continue :

- elle choisit un exposant e tel que $\text{pgcd}(e, \varphi(n)) = 1$,
- elle calcule l'inverse d de e module $\varphi(n)$: $d \times e \equiv 1 \pmod{\varphi(n)}$. Ce calcul se fait par l'algorithme d'Euclide étendu.

Exemple 1.

- Alice choisit par exemple $e = 5$ et on a bien $\text{pgcd}(e, \varphi(n)) = \text{pgcd}(5, 64) = 1$,
- Alice applique l'algorithme d'Euclide étendu pour calculer les coefficients de Bézout correspondant à $\text{pgcd}(e, \varphi(n)) = 1$. Elle trouve $5 \times 13 + 64 \times (-1) = 1$. Donc $5 \times 13 \equiv 1 \pmod{64}$ et l'inverse de e modulo $\varphi(n)$ est $d = 13$.

Exemple 2.

- Alice choisit par exemple $e = 7$ et on a bien $\text{pgcd}(e, \varphi(n)) = \text{pgcd}(7, 10\,200) = 1$,
- L'algorithme d'Euclide étendu pour $\text{pgcd}(e, \varphi(n)) = 1$ donne $7 \times (-1457) + 10\,200 \times 1 = 1$. Mais $-1457 \equiv 8743 \pmod{\varphi(n)}$, donc pour $d = 8743$ on a $d \times e \equiv 1 \pmod{\varphi(n)}$.

Clé publique

La **clé publique** d'Alice est constituée des deux nombres :

n et e

Et comme son nom l'indique Alice communique sa clé publique au monde entier.

Exemple 1. $n = 85$ et $e = 5$

Exemple 2. $n = 10\,403$ et $e = 7$

Clé privée

Alice garde pour elle sa **clé privée** :

d

Alice détruit en secret p , q et $\varphi(n)$ qui ne sont plus utiles. Elle conserve secrètement sa clé privée.

Exemple 1. $d = 13$

Exemple 2. $d = 8743$

6.2. Chiffrement du message

Bruno veut envoyer un message secret à Alice. Il se débrouille pour que son message soit un entier (quitte à découper son texte en bloc et à transformer chaque bloc en un entier).

Message

Le message est un entier m , tel que $0 \leq m < n$.

Exemple 1. Bruno veut envoyer le message $m = 10$.

Exemple 2. Bruno veut envoyer le message $m = 1234$.

Message chiffré

Bruno récupère la clé publique d'Alice : n et e avec laquelle il calcule, à l'aide de l'algorithme d'exponentiation rapide, le message chiffré :

$$x \equiv m^e \pmod{n}$$

Il transmet ce message x à Alice

Exemple 1. $m = 10$, $n = 85$ et $e = 5$ donc

$$x \equiv m^e \pmod{n} \equiv 10^5 \pmod{85}$$

On peut ici faire les calculs à la main :

$$10^2 \equiv 100 \equiv 15 \pmod{85}$$

$$10^4 \equiv (10^2)^2 \equiv 15^2 \equiv 225 \equiv 55 \pmod{85}$$

$$x \equiv 10^5 \equiv 10^4 \times 10 \equiv 55 \times 10 \equiv 550 \equiv 40 \pmod{85}$$

Le message chiffré est donc $x = 40$.

Exemple 2. $m = 1234$, $n = 10\,403$ et $e = 7$ donc

$$x \equiv m^e \pmod{n} \equiv 1234^7 \pmod{10\,403}$$

On utilise l'ordinateur pour obtenir que $x = 10\,378$.

6.3. Déchiffrement du message

Alice reçoit le message x chiffré par Bruno, elle le décrypte à l'aide de sa clé privée d , par l'opération :

$$m \equiv x^d \pmod{n}$$

qui utilise également l'algorithme d'exponentiation rapide.

Nous allons prouver dans le lemme 1, que par cette opération Alice retrouve bien le message original m de Bruno.

Exemple 1. $c = 40$, $d = 13$, $n = 85$ donc

$$x^d \equiv (40)^{13} \pmod{85}.$$

Calculons à la main $40^{13} \pmod{85}$ on note que $13 = 8 + 4 + 1$, donc $40^{13} = 40^8 \times 40^4 \times 40$.

$$40^2 \equiv 1600 \equiv 70 \pmod{85}$$

$$40^4 \equiv (40^2)^2 \equiv 70^2 \equiv 4900 \equiv 55 \pmod{85}$$

$$40^8 \equiv (40^4)^2 \equiv 55^2 \equiv 3025 \equiv 50 \pmod{85}$$

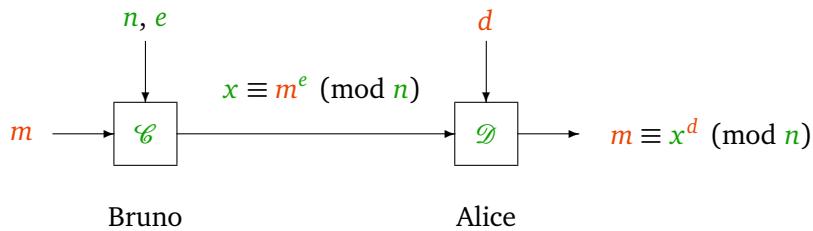
Donc

$$x^d \equiv 40^{13} \equiv 40^8 \times 40^4 \times 40 \equiv 50 \times 55 \times 40 \equiv 10 \pmod{85}$$

qui est bien le message m de Bruno.

Exemple 2. $c = 10\,378$, $d = 8743$, $n = 10\,403$. On calcule par ordinateur $x^d \equiv (10\,378)^{8743} \pmod{10\,403}$ qui vaut exactement le message original de Bruno $m = 1234$.

6.4. Schéma



Clés d'Alice :

- publique : n, e
- privée : d

6.5. Lemme de déchiffrement

Le principe de déchiffrement repose sur le petit théorème de Fermat amélioré.

Lemme 1.

Soit d l'inverse de e modulo $\varphi(n)$.

$$\boxed{\text{Si } x \equiv m^e \pmod{n} \text{ alors } m \equiv x^d \pmod{n}.}$$

Ce lemme prouve bien que le message original m de Bruno, chiffré par clé publique d'Alice (e, n) en le message x , peut-être retrouvé par Alice à l'aide de sa clé secrète d .

Démonstration. • Que d soit l'inverse de e modulo $\varphi(n)$ signifie $d \cdot e \equiv 1 \pmod{\varphi(n)}$. Autrement dit, il existe $k \in \mathbb{Z}$ tel que $d \cdot e = 1 + k \cdot \varphi(n)$.

- On rappelle que par le petit théorème de Fermat généralisé : lorsque m et n sont premiers entre eux

$$m^{\varphi(n)} \equiv m^{(p-1)(q-1)} \equiv 1 \pmod{n}$$

- **Premier cas** $\text{pgcd}(m, n) = 1$.

Notons $c \equiv m^e \pmod{n}$ et calculons x^d :

$$x^d \equiv (m^e)^d \equiv m^{e \cdot d} \equiv m^{1+k \cdot \varphi(n)} \equiv m \cdot m^{k \cdot \varphi(n)} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \cdot (1)^k \equiv m \pmod{n}$$

- **Deuxième cas** $\text{pgcd}(m, n) \neq 1$.

Comme n est le produit des deux nombres premiers p et q et que m est strictement plus petit que n alors si m et n ne sont pas premiers entre eux cela implique que p divise m ou bien q divise m (mais pas les deux en même temps). Faisons l'hypothèse $\text{pgcd}(m, n) = p$ et $\text{pgcd}(m, q) = 1$, le cas $\text{pgcd}(m, n) = q$ et $\text{pgcd}(m, p) = 1$ se traiterait de la même manière.

Étudions $(m^e)^d$ à la fois modulo p et modulo q à l'image de ce que nous avons fait dans la preuve du théorème de Fermat amélioré.

- modulo p : $m \equiv 0 \pmod{p}$ et $(m^e)^d \equiv 0 \pmod{p}$ donc $(m^e)^d \equiv m \pmod{p}$,
- modulo q : $(m^e)^d \equiv m \times (m^{\varphi(n)})^k \equiv m \times (m^{q-1})^{(p-1)k} \equiv m \pmod{q}$.

Comme p et q sont deux nombres premiers distincts, ils sont premiers entre eux et on peut écrire comme dans la preuve du petit théorème de Fermat amélioré que

$$(m^e)^d \equiv m \pmod{n}$$

□

6.6. Algorithmes

La mise en œuvre est maintenant très simple. Alice choisit deux nombres premiers p et q et un exposant e . Voici le calcul de la clé secrète :

Code 53 (*rsa.py (1)*).

```
def cle_privee(p,q,e) :
    n = p * q
    phi = (p-1)*(q-1)
    c,d,dd = euclide_etendu(e,phi)           # Pgcd et coeff de Bézout
    return(d % phi)                          # Bon représentant
```

Le chiffrement d'un message m est possible par tout le monde, connaissant la clé publique (n, e).

Code 54 (*rsa.py (2)*).

```
def codage_rsa(m,n,e):
    return pow(m,e,n)
```

Seule Alice peut déchiffrer le message crypté x , à l'aide de sa clé privée d .

Code 55 (*rsa.py (3)*).

```
def decodage_rsa(x,n,d):
    return pow(x,d,n)
```

Pour continuer...

Bibliographie commentée :

1. **Histoire des codes secrets** de Simon Singh, Le livre de Poche.

Les codes secrets racontés comme un roman policier. Passionnant. Idéal pour les plus littéraires.

2. **Comprendre les codes secrets** de Pierre Vigoureux, édition Ellipses.

Un petit livre très clair et très bien écrit, qui présente un panorama complet de la cryptographie sans rentrer dans les détails mathématiques. Idéal pour les esprits logiques.

3. **Codage et cryptographie** de Joan Gómez, édition Le Monde – Images des mathématiques. Un autre petit livre très clair et très bien, un peu de maths, des explications claires et des encarts historiques intéressants.

4. **Introduction à la cryptographie** de Johannes Buchmann, édition Dunod.

Un livre d'un niveau avancé (troisième année de licence) pour comprendre les méthodes mathématiques de la cryptographie moderne. Idéal pour unifier les points de vue des mathématiques avec l'informatique.

5. **Algèbre - Première année** de Liret et Martinais, édition Dunod.

Livre qui recouvre tout le programme d'algèbre de la première année, très bien adapté aux étudiants des universités. Pas de cryptographie.

Calcul formel

- Vidéo ■ partie 1. Premiers pas avec Sage
- Vidéo ■ partie 2. Structures de contrôle avec Sage
- Vidéo ■ partie 3. Suites récurrentes et preuves formelles
- Vidéo ■ partie 4. Suites récurrentes et visualisation
- Vidéo ■ partie 5. Algèbre linéaire
- Vidéo ■ partie 6a. Courbes et surfaces
- Vidéo ■ partie 6b. Courbes et surfaces
- Vidéo ■ partie 7. Calculs d'intégrales
- Vidéo ■ partie 8. Polynômes
- Vidéo ■ partie 9. Équations différentielles

1. Premiers pas avec Sage

Le calcul formel est un domaine charnière entre mathématiques et informatique. L'objectif de ce cours est d'obtenir des algorithmes efficaces afin de manipuler des objets mathématiques abstraits tels que les fonctions, les polynômes, les matrices, etc. À notre niveau, le calcul formel ou symbolique sera synonyme de «mathématiques effectives et efficaces».

1.1. Hello world !

Servons-nous d'abord de Sage comme d'une calculatrice :

Travaux pratiques 1.

1. Calculer $1 + 2 \times 3^4$.
2. Calculer $\frac{22}{33}$.
3. Calculer $\cos(\frac{\pi}{6})$.

Voilà les résultats :

Code 56 (*hello-world.sage*).

```
sage: 1+2*3^4
163
sage: 22/33
2/3
sage: cos(pi/6)
1/2*sqrt(3)
```

On retient que :

- Sage connaît les opérations classiques $+$, $-$, \times , $/$. La puissance a^b s'écrit a^b ou $a**b$.
- Sage fait des calculs exacts, il simplifie $\frac{22}{33}$ en $\frac{2}{3}$, contrairement à une banale calculatrice qui afficherait $0.6666\dots$
- Sage manipule les fonctions et les constantes usuelles : par exemple $\cos\left(\frac{\pi}{6}\right) = \frac{\sqrt{3}}{2}$.

Dans toute la suite, on omettra l'invite de commande « sage : ». En fin de section vous trouverez davantage d'informations sur la mise en place de Sage.

1.2. Calcul formel

L'utilisation d'un système de calcul symbolique conduit le mathématicien à un type de démarche auquel il n'est traditionnellement pas habitué : l'expérimentation !

1. Je me pose des questions.
2. J'écris un algorithme pour tenter d'y répondre.
3. Je trouve une conjecture sur la base des résultats expérimentaux.
4. Je prouve la conjecture.

Travaux pratiques 2.

1. Que vaut le nombre complexe $(1 - i)^k$, pour $k \geq 0$?
2. Les nombres de la forme $F_n = 2^{(2^n)} + 1$ sont-ils tous premiers ?

Nous pouvons faire des calculs avec les nombres complexes. En posant $z = 1 - i$, on calcule successivement la partie réelle (par la commande `real_part`), la partie imaginaire (`imag_part`), le module (`abs`) et l'argument (`arg`) de z^0, z^1, z^2, \dots

Code 57 (*motiv-calcul-formel.sage* (1)).

```
z = 1-I
for k in range(10):
    print k, z^k, real_part(z^k), imag_part(z^k), abs(z^k), arg(z^k)

0 1      5 4*I - 4      10 -32*I      z4ℓ = (-1)ℓ22ℓ
1 -I + 1  6 8*I       11 -32*I - 32  z4ℓ+1 = (-1)ℓ22ℓ(1 - i)
2 -2*I    7 8*I + 8   12 -64        z4ℓ+2 = (-1)ℓ22ℓ+1(-i)
3 -2*I - 2 8 16      13 64*I - 64   z4ℓ+3 = (-1)ℓ22ℓ+1(-1 - i)
4 -4      9 -16*I + 16 14 128*I
```

On remarque expérimentalement une structure assez simple. Par exemple pour passer de $z_k = (1 + i)^k$ à $z_{k+1} = (1 + i)^{k+1}$ le module est multiplié par $\sqrt{2}$ alors que l'argument change de $-\frac{\pi}{4}$. On en conjecture $z_{k+1} = \sqrt{2}e^{-\frac{i\pi}{4}}z_k$. Comme $z_0 = (1 - i)^0 = 1$ alors on conjecture $z_k = \sqrt{2}^k e^{-\frac{k i\pi}{4}}$.

Passons à la preuve : en écrivant sous la forme module-argument $z = \sqrt{2}e^{-\frac{i\pi}{4}}$, on obtient bien que $z^k = \sqrt{2}^k e^{-\frac{k i\pi}{4}}$. On pourrait aussi obtenir une expression simple en discutant selon les valeurs de k modulo 8.

Le calcul formel permet aussi d'éviter de passer des années à essayer de montrer un résultat qui s'avère faux au final. Pierre de Fermat pensait que tous les nombres de la forme $F_n = 2^{2^n} + 1$ étaient premiers. Cent ans plus tard, Euler calcule que $F_5 = 4\ 294\ 967\ 297 = 641 \times 6\ 700\ 417$ n'est pas un nombre premier.

Code 58 (*motiv-calcul-formel.sage* (2)).

```
for n in range(8):
    print n, factor(2^(2^n)+1)
```

1.3. Calcul formel vs calcul numérique

Même si Sage est conçu pour le calcul symbolique, il sait aussi effectuer des calculs numériques. Bien que non exact, le calcul numérique possède plusieurs avantages : il est plus rapide, souvent suffisant pour les applications pratiques et peut aussi être plus lisible.

Travaux pratiques 3.

Quelle est la limite de la suite définie par récurrence :

$$u_0 = 1 \quad u_{n+1} = \sqrt{1 + u_n} \text{ pour } n \geq 0 \quad ?$$

Si on calcule formellement les premiers termes on trouve :

$$u_0 = 1 \quad u_1 = \sqrt{2} \quad u_2 = \sqrt{1 + \sqrt{2}} \quad u_3 = \sqrt{1 + \sqrt{1 + \sqrt{2}}} \quad u_4 = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{2}}}}$$

Ce qui n'éclaire pas vraiment sur le comportement de la suite.

Code 59 (*calcul-numerique.sage*).

```
u = 1
for i in range(10):
    u = sqrt(1 + u)
    print(u)
    print(numerical_approx(u))
```

Par contre au vu des approximations :

$$\begin{aligned} u_0 &= 1 \\ u_1 &= 1.4142... \\ u_2 &= 1.5537... \\ u_3 &= 1.5980... \\ u_4 &= 1.6118... \\ u_5 &= 1.6161... \end{aligned}$$

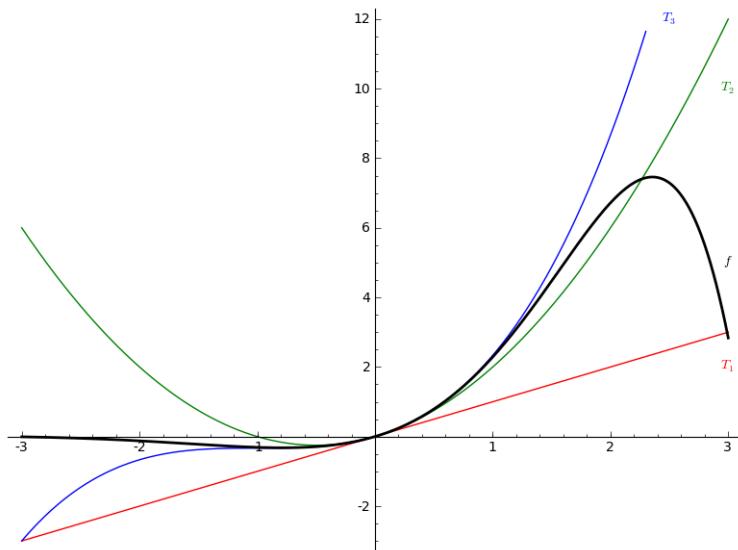
on peut émettre plusieurs conjectures : la suite est croissante, elle est majorée par 2 et converge. En poussant les calculs, une approximation de la limite est 1.618033... Les plus observateurs auront reconnu une approximation du nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$, solution positive de l'équation $x^2 - x - 1$. On renvoie à un cours d'analyse pour la preuve que la suite (u_n) converge effectivement vers ϕ .

1.4. Graphiques

La production de graphiques est un formidable outil qui permet de mettre en lumière des phénomènes complexes.

Travaux pratiques 4.

Soit la fonction $f(x) = \sin(x) \cdot \exp(x)$. Calculer les premiers polynômes de Taylor associés aux développements limités de f en 0. Tracer leurs graphes. Quelles propriétés des développements limités cela met-il en évidence ?



- Après avoir défini la fonction f par $f = \sin(x)*\exp(x)$, la commande `taylor(f,x,0,n)` renvoie le DL de f en $x = 0$ à l'ordre n , par exemple ici $f(x) = x + x^2 + \frac{1}{3}x^3 + \epsilon(x)x^3$, donc $T_1(x) = x$, $T_2(x) = x + x^2$, $T_3(x) = x + x^2 + \frac{1}{3}x^3$.
- La commande `plot(f,(a,b))` trace le graphe de f sur l'intervalle $[a, b]$.

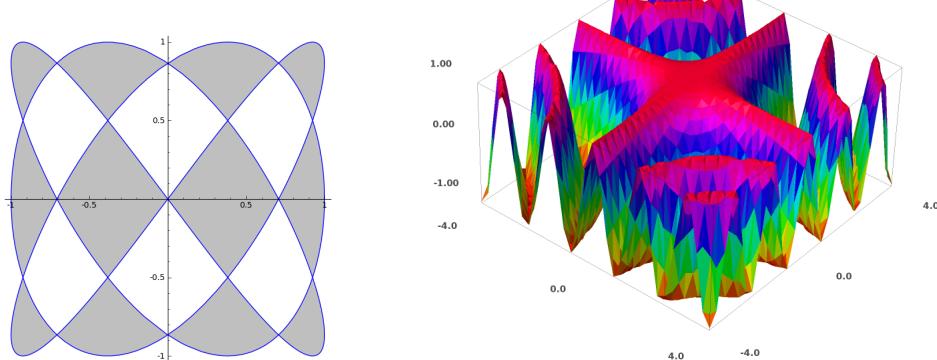
Il est perceptible que :

- Les polynômes de Taylor sont une bonne approximation de la fonction au voisinage d'un point.
- Plus l'ordre du DL est élevé, meilleure est l'approximation.
- L'approximation est seulement *locale* : loin du point considéré (ici l'origine) les polynômes de Taylor n'approchent plus du tout la fonction.

Il est possible de tracer une grande variété de graphiques. Voici par exemple la courbe de Lissajous d'équation $t \mapsto (\cos(3t), \sin(4t))$ et le graphe de la fonction de deux variables définie par $f(x, y) = \cos(xy)$. Les commandes sont :

- `parametric_plot((cos(3*t), sin(4*t)), (t, 0, 2*pi))`
- `plot3d(cos(x*y), (x, -4, 4), (y, -4, 4))`

Attention ! Il faut avoir au préalable définir les variables utilisées : `var('t')` et `var('x,y')`. (En fait, seule la variable x est définie par défaut dans Sage.)



1.5. Le calcul formel peut-il tout faire ?

Le calcul formel ne résout malheureusement pas tous les problèmes de mathématique d'un coup de baguette magique !

Travaux pratiques 5.

1. Pouvez-vous calculer les solutions réelles de $x^k - x - 1 = 0$ pour les entiers $k \geq 2$?
2. Est-ce que Sage sait que toutes ces expressions sont nulles ?

$$2^{10} - 1024 \quad (x + 1)^2 - x^2 - 2x - 1 \quad \sin^2(x) + \cos^2(x) - 1$$

1. La première limitation est propre aux mathématiques : on ne peut pas trouver une écriture explicite des solutions de toutes les équations. Pour $x^2 - x - 1 = 0$ à l'aide de la commande `solve(x^2-x-1==0, x)` on trouve bien les deux solutions $\frac{1+\sqrt{5}}{2}$ et $\frac{1-\sqrt{5}}{2}$. Par contre `solve(x^5-x-1==0, x)` ne renvoie pas les solutions, mais il renvoie l'équation qui définit les solutions (ce qui ne nous avance guère). Ce n'est pas ici un problème de Sage. En effet, il n'est mathématiquement pas possible d'exprimer la solution réelle de $x^5 - x - 1 = 0$ à l'aide de racines ($\sqrt{}$, $\sqrt[3]{}$, $\sqrt[4]{}$, ...). C'est seulement possible jusqu'au degré 4. Par contre on obtient une approximation de la solution réelle par la commande `find_root(x^5-x-1==0, -1, 2)` en précisant que l'on cherche la solution sur l'intervalle $[-1, 2]$.

2. (a) Sans problème $2^{10}-1024$ renvoie 0.

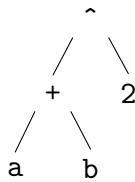
(b) Il est nécessaire de développer pour trouver 0 : `expand((x+1)^2-x^2-2*x-1)`.

(c) Il faut explicitement préciser de simplifier l'expression trigonométrique : d'abord `f = sin(x)^2+cos(x)^2 - 1` puis on demande de simplifier `f.simplify_trig()` pour obtenir 0.

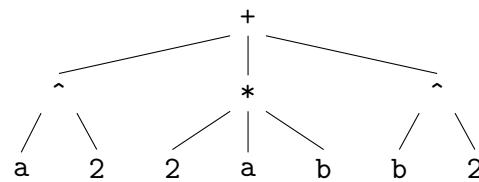
Il n'est pas du tout évident pour un ordinateur de reconnaître les identités comme $(a + b)^2 = a^2 + 2ab + b^2$ ou bien $\cos^2(x) + \sin^2(x) = 1$. Souvenez-vous d'ailleurs qu'il faut plusieurs années d'apprentissage pour les assimiler. Lorsqu'il y a plusieurs écritures d'une même expression, il n'est pas non plus évident pour l'ordinateur de savoir quelle forme est la plus adaptée à l'utilisateur. Par exemple, selon le contexte, les trois écritures sont utiles : $(a - b)^3 = (a - b)(a^2 - 2ab + b^2) = a^3 - 3a^2b + 3a^2b - b^3$. Il faudra donc « guider » le logiciel de calcul formel avec les fonctions `expand`, `factor`, `simplify`...

Remarque.

Pour avoir une idée de la difficulté à identifier deux expressions, voici une représentation de la façon dont les expressions $(a + b)^2$ et $a^2 + 2ab + b^2$ sont stockées dans le logiciel.



$$(a + b)^2$$



$$a^2 + 2ab + b^2$$

1.6. Un peu plus sur Sage

Ce cours n'a pas pour but d'être un manuel d'utilisation du logiciel Sage. Vous trouverez sur internet des tutoriels pour démarrer et pour un usage avancé :

[Site officiel de Sage](#)

Il existe aussi un livre gratuit :

[Calcul mathématique avec Sage](#)

Une façon simple d'obtenir de l'aide pour une commande Sage est d'utiliser le point d'interrogation : `ln?` (ou bien `help(ln)`). Vous y apprendrez que la fonction `ln` est le logarithme naturel.

Il y a deux façons d'utiliser Sage :

- *En ligne de commande* : vous obtenez une fenêtre avec l'invite `sage` : puis vous tapez vos commandes (en n'oubliant pas de faire une copie de votre travail dans un fichier texte du type `mon_programme.sage`).
- *Dans votre navigateur* : à partir de l'invite `sage` : vous tapez `notebook()` pour obtenir une interface complète et conviviale.

Voici une liste de fonctions usuelles :

<code>abs(x)</code>	$ x $
<code>x^n</code> ou <code>x**n</code>	x^n
<code>sqrt(x)</code>	\sqrt{x}
<code>exp(x)</code>	e^x
<code>ln(x)</code> ou <code>log(x)</code>	$\ln x$ logarithme népérien
<code>log(x,10)</code>	$\log x$ logarithme décimal
<code>cos(x)</code> , <code>sin(x)</code> , <code>tan(x)</code>	$\cos x$, $\sin x$, $\tan x$ en radians
<code>arccos(x)</code> , <code>arcsin(x)</code> , <code>arctan(x)</code>	$\arccos x$, $\arcsin x$, $\arctan x$ en radians
<code>floor(x)</code>	partie entière $E(x)$: plus grand entier $n \leq x$ (<code>floor</code> = plancher)
<code>ceil(x)</code>	plus petit entier $n \geq x$ (<code>ceil</code> = plafond)

Il existe des fonctions spécifiques qui manipulent les entiers, les vecteurs, les matrices, les polynômes, les fonctions mathématiques... Nous les découvrions au fil des chapitres.

La syntaxe de Sage est celle du langage Python. Il n'est pas nécessaire de connaître ce langage, la syntaxe sera introduite au fur et à mesure. Cependant vous pourrez étudier avec profit le chapitre «Algorithmes et mathématiques».

Pour l'instant voici ce que l'on retient de l'exemple

Code 60 (*motiv-calcul-formel.sage (2)*).

```
for n in range(8):
    print n, factor(2^(2^n)+1)
```

- Une boucle `for n in range(N)` : l'indice n parcourt les entiers de 0 à $N - 1$.
- Le bloc d'instructions suivant `print n, factor(2^(2^n)+1)` est donc exécuté successivement pour $n = 0, 1, \dots, N - 1$.
- Les espaces en début de ligne (*l'indentation*) sont essentielles car elles délimitent le début et la fin d'un bloc d'instructions.

2. Structures de contrôle avec Sage

2.1. Boucles

Boucle for (pour)

Pour faire varier un élément dans un ensemble on utilise l'instruction "for x in ensemble:". Le bloc d'instructions suivant sera successivement exécuté pour toutes les valeurs de x.

Code 61 (*structures.sage (1)*).

```
for x in ensemble:
    première ligne de la boucle
    deuxième ligne de la boucle
    ...
    dernière ligne de la boucle
instructions suivantes
```

Notez encore une fois que le bloc d'instructions exécuté est délimité par les espaces en début de ligne. Un exemple fréquent est "for k in range(n) :" qui fait varier k de 0 à $n - 1$.

Liste range (intervalle)

En fait `range(n)` renvoie la liste des n premiers entiers : $[0, 1, 2, \dots, n-1]$

Plus généralement `range(a, b)` renvoie la liste $[a, a+1, \dots, b-1]$. Alors que `range(a, b, c)` effectue une saut de c termes, par exemple `range(0, 101, 5)` renvoie la liste $[0, 5, 10, 15, \dots, 100]$.

Une autre façon légèrement différente pour définir des listes d'entiers est l'instruction `[a..b]` qui renvoie la liste des entiers k tels que $a \leq k \leq b$. Par exemple après l'instruction `for k in [-7..12]` : l'entier k va prendre successivement les valeurs $-7, -6, -5, \dots, -1, 0, +1, \dots$ jusqu'à $+12$.

La boucle `for` permet plus généralement de parcourir n'importe quelle liste, par exemple `for x in [0.13, 0.31, 0.53, 0.98]`, fait prendre à x les 4 valeurs de $\{0.13, 0.31, 0.53, 0.98\}$.

Boucle while (tant que)

Code 62 (*structures.sage (2)*).

```
while condition:
    première ligne de la boucle
    deuxième ligne de la boucle
    ...
    dernière ligne de la boucle
instructions suivantes
```

La boucle `while` exécute le bloc d'instructions, tant que la condition est vérifiée. Lorsque la condition n'est plus vérifiée, on passe aux instructions suivantes.

Voici le calcul de la racine carrée entière d'un entier n :

Code 63 (*structures.sage (3)*).

```
n = 123456789      # l'entier dont on cherche la racine carrée
k = 1                # le premier candidat
while k*k <= n:     # tant que le carré de k ne dépasse pas n
```

```

k = k+1      # on passe au candidat suivant
print(k-1)    # la racine cherchée

```

Lorsque la recherche est terminée, k est le plus petit entier dont le carré dépasse n , par conséquent la racine carrée entière de n est $k - 1$.

Notez qu'utiliser une boucle « tant que » comporte des risques, en effet il faut toujours s'assurer que la boucle se termine. Voici quelques instructions utiles pour les boucles : `break` termine immédiatement la boucle alors que `continue` passe directement à l'itération suivante (sans exécuter le reste du bloc).

Test if... else (si... sinon)

Code 64 (*structures.sage (4)*).

```

if condition:
    première ligne d'instruction
    ...
dernière ligne d'instruction
else:
    autres instructions

```

Si la condition est vérifiée, c'est le premier bloc d'instructions qui est exécuté, si la condition n'est pas vérifiée, c'est le second bloc. On passe ensuite aux instructions suivantes.

2.2. Booléens et conditions

Booléens

Une **expression booléenne** est une expression qui peut seulement prendre les valeurs « Vrai » ou « Faux » et qui sont codées par `True` ou `False`. Les expressions booléennes s'obtiennent principalement par des conditions.

Quelques conditions

Voici une liste de conditions :

- `a < b` : teste l'inégalité stricte $a < b$,
- `a > b` : teste l'inégalité stricte $a > b$,
- `a <= b` : teste l'inégalité large $a \leq b$,
- `a >= b` : teste l'inégalité large $a \geq b$,
- `a == b` : teste l'égalité $a = b$,
- `a <> b` (ou `a != b`) : teste la non égalité $a \neq b$.
- `a in B` : teste si l'élément a appartient à B .

Remarque. 1. Une condition prend la valeur `True` si elle est vraie et `False` sinon. Par exemple `x == 2` renvoie `True` si x vaut 2 et `False` sinon. La valeur de x n'est pas modifiée pas le test.

2. Il ne faut surtout pas confondre le test d'égalité `x == 2` avec l'affectation `x = 2` (après cette dernière instruction x vaut 2).
3. On peut combiner des conditions avec les opérateurs `and`, `or`, `not`. Par exemple : `(n>0) and (not (is_prime(n))` est vraie si et seulement si n est strictement positif et non premier.

Travaux pratiques 6.

1. Pour deux assertions logiques P, Q écrire l'assertion $P \implies Q$.
2. Une **tautologie** est une assertion vraie quelles que soient les valeurs des paramètres, par exemple $(P \text{ ou } (\text{non } P))$ est vraie que l'assertion P soit vraie ou fausse. Montrer que l'assertion suivante est une tautologie :

$$\text{non} \left(\left[\text{non} (P \text{ et } Q) \text{ et } (Q \text{ ou } R) \right] \text{ et } \left[P \text{ et } (\text{non } R) \right] \right)$$

Il faut se souvenir du cours de logique qui définit l'assertion « $P \implies Q$ » comme étant « $\text{non} (P) \text{ ou } Q$ ». Il suffit donc de renvoyer : `not(P) or Q`.

Pour l'examen de la tautologie, on teste toutes les possibilités et on vérifie que le résultat est vrai dans chaque cas !

Code 65 (*structures.sage* (10)).

```
for P in {True, False}:
    for Q in {True, False}:
        for R in {True, False}:
            print(not( (not(P and Q) and (Q or R)) and ( P and (not R) ) ))
```

2.3. Fonctions informatiques

Une fonction informatique prend en entrée un ou plusieurs paramètres et renvoie un ou plusieurs résultats.

Code 66 (*structures.sage* (5)).

```
def mafonction (mesvariables):
    première ligne d'instruction
    ...
    dernière ligne d'instruction
    return monresultat
```

Par exemple voici comment définir une fonction valeur absolue.

Code 67 (*structures.sage* (6)).

```
def valeur_absolue(x):
    if x >= 0:
        return x
    else:
        return -x
```

Par exemple `valeur_absolue(-3)` renvoie +3.

Il est possible d'utiliser cette fonction dans le reste du programme ou dans une autre fonction. Noter aussi qu'une fonction peut prendre plusieurs paramètres.

Par exemple que fait la fonction suivante ? Quel nom aurait été plus approprié ?

Code 68 (*structures.sage* (7)).

```
def ma_fonction(x,y):
    resultat = (x+y+valeur_absolue(x-y))/2
    return resultat
```

Travaux pratiques 7.

On considère trois réels distincts a, b, c et on définit le polynôme

$$P(X) = \frac{(X-a)(X-b)}{(c-a)(c-b)} + \frac{(X-b)(X-c)}{(a-b)(a-c)} + \frac{(X-a)(X-c)}{(b-a)(b-c)} - 1$$

1. Définir trois variables par `var('a, b, c')`. Définir une fonction `polynome(x)` qui renvoie $P(x)$.
2. Calculer $P(a), P(b), P(c)$.
3. Comment un polynôme de degré 2 pourrait avoir 3 racines distinctes ? Expliquez ! (Vous pourrez appliquer la méthode `full_simplify()` à votre fonction.)

2.4. Variable locale/variable globale

Il faut bien faire la distinction entre les variables locales et les variables globales.

Par exemple, qu'affiche l'instruction `print(x)` dans ce petit programme ?

Code 69 (*structures.sage (9)*).

```
x = 2
def incremente(x):
    x = x+1
    return x
incremente(x)
print(x)
```

Il faut bien comprendre que tous les x de la fonction `incremente` représentent une variable locale qui n'existe que dans cette fonction et disparaît en dehors. On aurait tout aussi bien pu définir `def incremente(y): y = y+1 return y` ou bien utiliser la variable `truc` ou `zut`. Par contre les x des lignes `x=2`, `incremente(x)`, `print(x)` correspondent à une variable globale. Une variable globale existe partout (sauf dans les fonctions où une variable locale porte le même nom !).

En mathématique, on parle plutôt de variable muette au lieu de variable locale, par exemple dans

$$\sum_{k=0}^n k^2$$

k est une variable muette, on aurait pu tout aussi bien la nommer i ou x . Par contre il faut au préalable avoir défini n (comme une variable globale), par exemple « $n = 7$ » ou « fixons un entier n »...

2.5. Conjecture de Syracuse

Mettez immédiatement en pratique les structures de contrôle avec le travail suivant.

Travaux pratiques 8.

La **suite de Syracuse** de terme initial $u_0 \in \mathbb{N}^*$ est définie par récurrence pour $n \geq 0$ par :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

1. Écrire une fonction qui, à partir de u_0 et de n , calcule le terme u_n de la suite de Syracuse.
2. Vérifier pour différentes valeurs du terme initial u_0 , que la suite de Syracuse atteint la valeur 1 au bout d'un certain rang (puis devient périodique : ..., 1, 4, 2, 1, 4, 2, ...). Écrire une fonction qui pour un certain choix de u_0 renvoie le premier rang n tel que $u_n = 1$.

Personne ne sait prouver que pour toutes les valeurs du terme initial u_0 , la suite de Syracuse atteint toujours la valeur 1.

Voici le code pour calculer le n -ème terme de la suite.

Code 70 (*suite-syracuse.sage (1)*).

```
def syracuse(u0,n):
    u = u0
    for k in range(n):
        if u%2 == 0:
            u = u//2
        else:
            u = 3*u+1
    return u
```

Quelques remarques : on a utilisé une boucle `for` pour calculer les n premiers termes de la suite et aussi un test `if...else`.

L'instruction `u%2` renvoie le reste de la division de u par 2. Le booléen `u%2 == 0` teste donc la parité de u . L'instruction `u//2` renvoie le quotient de la division euclidienne de u par 2.

Remarque.

Si $a \in \mathbb{Z}$ et $n \in \mathbb{N}^*$ alors

- $a \% n$ est le **reste** de la division euclidienne de a par n . C'est donc l'entier r tel que $r \equiv a \pmod{n}$ et $0 \leq r < n$.
- $a // n$ est le **quotient** de la division euclidienne de a par n . C'est donc l'entier q tel que $a = nq + r$ et $0 \leq r < n$.

Il ne faut pas confondre la division de nombres réels : a/b et le quotient de la division euclidienne de deux entiers : $a//b$. Par exemple $7/2$ est la fraction $\frac{7}{2}$ dont la valeur numérique est 3,5 alors que $7//2$ renvoie 3. On a $7\%2$ qui vaut 1. On a bien $7 = 2 \times 3 + 1$.

Pour notre cas `u%2` renvoie le reste de la division euclidienne de u par 2. Donc cela vaut 0 si u est pair et 1 si u est impair. Ainsi `u%2 == 0` teste si u est pair ou impair. Ensuite `u = u//2` divise u par 2 (en fait cette opération n'est effectuée que pour u pair).

Pour calculer à partir de quel rang on obtient 1, on utilise une boucle `while` dont les instructions sont exécutées tant que $u \neq 1$. Donc par construction quand la boucle se termine c'est que $u = 1$.

Code 71 (*suite-syracuse.sage (2)*).

```
def vol_syracuse(u0):
    u = u0
    n = 0
    while u <> 1:
        n = n+1
        if u%2 == 0:
            u = u//2
        else:
            u = 3*u+1
    return n
```

3. Suites récurrentes et preuves formelles

Commençons par un exemple. Après avoir déclaré les deux variables a et b par `var('a,b')` alors la commande

```
expand((a+b)^2-a^2-2*a*b-b^2)
```

renvoie 0. Ce simple calcul doit être considéré comme une preuve par ordinateur de l'identité $(a + b)^2 = a^2 + 2ab + b^2$, sans qu'il y ait obligation de vérification. Cette « preuve » est fondée sur la confiance en Sage qui sait manipuler les expressions algébriques. Nous allons utiliser la puissance du calcul formel, nous seulement pour faire des expérimentations, mais aussi pour effectuer des preuves (presque) à notre place.

Rappelons la démarche que l'on adopte :

1. Je me pose des questions.
2. J'écris un algorithme pour tenter d'y répondre.
3. Je trouve une conjecture sur la base des résultats expérimentaux.
4. Je prouve la conjecture.

3.1. La suite de Fibonacci

La suite de Fibonacci est définie par les relations suivantes :

$$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geq 2.$$

Travaux pratiques 9.

1. Calculer les 10 premiers termes de la suite de Fibonacci.
2. **Recherche d'une conjecture.** On cherche s'il peut exister des constantes $\phi, \psi, c \in \mathbb{R}$ telles que

$$F_n = c(\phi^n - \psi^n)$$

Nous allons calculer les constantes qui conviendraient.

- (a) On pose $G_n = \phi^n$. Quelle équation doit satisfaire ϕ afin que $G_n = G_{n-1} + G_{n-2}$?
- (b) Résoudre cette équation.
- (c) Calculer c, ϕ, ψ .
3. **Preuve.** On note $H_n = \frac{1}{\sqrt{5}}(\phi^n - \psi^n)$ où $\phi = \frac{1+\sqrt{5}}{2}$ et $\psi = \frac{1-\sqrt{5}}{2}$.
 - (a) Vérifier que $F_n = H_n$ pour les premières valeurs de n .
 - (b) Montrer à l'aide du calcul formel que $H_n = H_{n-1} + H_{n-2}$. Conclure.
1. Voici une fonction qui calcule le terme de rang n de la suite de Fibonacci en appliquant la formule de récurrence.

Code 72 (*fibonacci.sage* (1)).

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    F_n_2 = 0
    F_n_1 = 1
    for k in range(n-1):
        F_n = F_n_1 + F_n_2
        F_n_2 = F_n_1
        F_n_1 = F_n
```

```

F_n_2 = F_n_1
F_n_1 = F_n
return F_n

```

On affiche les valeurs de F_0 à F_9 :

Code 73 (*fibonacci.sage (2)*).

```

for n in range(10):
    print fibonacci(n)

```

Ce qui donne :

0	1	1	2	3	5	8	13	21	34
---	---	---	---	---	---	---	----	----	----

2. (a) On pose $G_n = \phi^n$ et on suppose que $G_n = G_{n-1} + G_{n-2}$. C'est-à-dire que l'on veut résoudre $\phi^n = \phi^{n-1} + \phi^{n-2}$. Notons (E) cette équation $X^n = X^{n-1} + X^{n-2}$, ce qui équivaut à l'équation $X^{n-2}(X^2 - X - 1) = 0$. On écarte la solution triviale $X = 0$, pour obtenir l'équation $X^2 - X - 1 = 0$.
- (b) On profite de Sage pour calculer les deux racines de cette dernière équation.

Code 74 (*fibonacci.sage (3)*).

```

solutions = solve(x^2-x-1==0,x)
print(solutions)

var('phi,psi')
phi = solutions[1].rhs()
psi = solutions[0].rhs()
print 'phi:',phi,'psi:',psi

```

Les solutions sont fournies sous la forme d'une liste que l'on appelle `solutions`. On récupère chaque solution avec `solutions[0]` et `solutions[1]`. Par exemple `solutions[1]` renvoie $x == 1/2*\sqrt{5} + 1/2$. Pour récupérer la partie droite de cette solution et la nommer ϕ , on utilise la fonction `rhs()` (pour *right hand side*) qui renvoie la partie droite d'une équation (de même `lhs()`, pour *left hand side* renverrait la partie gauche d'une équation).

On obtient donc les deux racines

$$\phi = \frac{1 + \sqrt{5}}{2} \quad \text{et} \quad \psi = \frac{1 - \sqrt{5}}{2}.$$

ϕ^n et ψ^n vérifient la même relation de récurrence que la suite de Fibonacci mais bien sûr les valeurs prises ne sont pas entières.

- (c) Comme $F_1 = 1$ alors $c(\phi - \psi) = 1$, ce qui donne $c = \frac{1}{\sqrt{5}}$.

La conséquence de notre analyse est que si une formule $F_n = c(\phi^n - \psi^n)$ existe alors les constantes sont nécessairement $c = \frac{1}{\sqrt{5}}$, $\phi = \frac{1+\sqrt{5}}{2}$, $\psi = \frac{1-\sqrt{5}}{2}$.

3. Dans cette question, nous allons maintenant montrer qu'avec ces constantes on a effectivement

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right).$$

- (a) On définit une nouvelle fonction `conjec` qui correspond à H_n , la conjecture que l'on souhaite montrer.
On vérifie que l'on a l'égalité souhaitée pour les premières valeurs de n .

```
Code 75 (fibonacci.sage (4)).
```

```
def conjec(n):
    return 1/sqrt(5)*(phi^n-psi^n)

for n in range(10):
    valeur = fibonacci(n)-conjec(n)
    print expand(valeur)
```

(b) Nous allons montrer que $F_n = H_n$ pour tout $n \geq 0$ en s'aidant du calcul formel.

- **Initialisation.** Il est clair que $F_0 = H_0 = 0$ et $F_1 = H_1 = 1$. On peut même demander à l'ordinateur de le faire, en effet `fibonacci(0)-conjec(0)` et `fibonacci(1)-conjec(1)` renvoient tous les deux 0.
- **Relation de récurrence.** Nous allons montrer que H_n vérifie exactement la même relation de récurrence que les nombres de Fibonacci. Encore une fois on peut le faire à la main, ou bien demander à l'ordinateur de le faire pour nous :

```
Code 76 (fibonacci.sage (5)).
```

```
var('n')
test = conjec(n)-conjec(n-1)-conjec(n-2)
print(test.simplify_radical())
```

Le résultat est 0. Comme le calcul est valable pour la variable formelle n , il est vrai quel que soit $n \geq 0$. Ainsi $H_n = H_{n-1} + H_{n-2}$. Il a tout de même fallu préciser à Sage de simplifier les racines carrées avec la commande `simplify_radical()`.

- **Conclusion.** Les suites (F_n) et (H_n) ont les mêmes termes initiaux et vérifient la même relation de récurrence. Elles ont donc les mêmes termes pour chaque rang : pour tout $n \geq 0$, $F_n = H_n$.

3.2. L'identité de Cassini

A votre tour maintenant d'expérimenter, conjecturer et prouver *l'identité de Cassini*.

Travaux pratiques 10.

Calculer, pour différentes valeurs de n , la quantité

$$F_{n+1}F_{n-1} - F_n^2.$$

Que constatez-vous ? Proposez une formule générale. Prouvez cette formule qui s'appelle l'identité de Cassini.

Notons pour $n \geq 1$:

$$C_n = F_{n+1}F_{n-1} - F_n^2.$$

Après quelques tests, on conjecture la formule très simple $C_n = (-1)^n$.

Voici deux preuves : une preuve utilisant le calcul formel et une autre à la main.

Première preuve – À l'aide du calcul formel.

On utilise la formule

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

et on demande simplement à l'ordinateur de vérifier l'identité !

```
Code 77 (cassini.sage (1)).
```

```
def fibonacci_bis(n):
```

```
return 1/sqrt(5)*((1+sqrt(5))/2)^n - ((1-sqrt(5))/2)^n )
```

Code 78 (*cassini.sage* (2)).

```
var('n')
cassini = fibonacci_bis(n+1)*fibonacci_bis(n-1)-fibonacci_bis(n)^2
print(cassini.simplify_radical())
```

L'ordinateur effectue (presque) tous les calculs pour nous. Le seul problème est que Sage échoue de peu à simplifier l'expression contenant des racines carrées (peut-être les versions futures feront mieux ?). En effet, après simplification on obtient :

$$C_n = (-1)^n \frac{(\sqrt{5}-1)^n (\sqrt{5}+1)^n}{2^{2n}}.$$

Il ne reste qu'à conclure à la main. En utilisant l'identité remarquable $(a-b)(a+b) = a^2 - b^2$, on obtient $(\sqrt{5}-1)(\sqrt{5}+1) = (\sqrt{5})^2 - 1 = 4$ et donc $C_n = (-1)^n \frac{4^n}{2^{2n}} = (-1)^n$.

Seconde preuve – Par récurrence.

On souhaite montrer que l'assertion

$$(\mathcal{H}_n) \quad F_n^2 = F_{n+1}F_{n-1} - (-1)^n$$

est vrai pour tout $n \geq 1$.

- Pour $n = 1$, $F_1^2 = 1^2 = 1 \times 0 - (-1)^1 = F_2 \times F_0 - (-1)^1$. \mathcal{H}_1 est donc vraie.
- Fixons $n \geq 1$ et supposons \mathcal{H}_n vraie. Montrons \mathcal{H}_{n+1} est vraie. On a :

$$F_{n+1}^2 = F_{n+1} \times (F_n + F_{n-1}) = F_{n+1}F_n + F_{n+1}F_{n-1}.$$

Par l'hypothèse \mathcal{H}_n , $F_{n+1}F_{n-1} = F_n^2 + (-1)^n$. Donc

$$F_{n+1}^2 = F_{n+1}F_n + F_n^2 + (-1)^n = (F_{n+1} + F_n)F_n - (-1)^{n+1} = F_{n+2}F_n - (-1)^{n+1}.$$

Donc \mathcal{H}_{n+1} est vraie.

- Conclusion, par le principe de récurrence, pour tout $n \geq 1$, l'identité de Cassini est vérifiée.

3.3. Une autre suite récurrente double

Travaux pratiques 11.

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

$$u_0 = \frac{3}{2}, \quad u_1 = \frac{5}{3} \quad \text{et} \quad u_n = 1 + 4 \frac{u_{n-2} - 1}{u_{n-1}u_{n-2}} \quad \text{pour } n \geq 2.$$

Calculer les premiers termes de cette suite. Émettre une conjecture. La prouver par récurrence.

Indication. Les puissances de 2 seront utiles...

Commençons par la partie expérimentale, le calcul des premières valeurs de la suite (u_n) :

Code 79 (*suite-recurrente.sage* (1)).

```
def masuite(n):
    if n == 0:
        return 3/2
    if n == 1:
        return 5/3
    u_n_2, u_n_1 = 3/2, 5/3
    for k in range(n-1):
        u_n = 1+4*(u_n_2-1)/(u_n_1*u_n_2)
        u_n_2 = u_n_1
        u_n_1 = u_n
```

```

u_n_2, u_n_1 = u_n_1, u_n
return u_n

```

(Notez l'utilisation de la double affectation du type `x, y = newx, newy`. C'est très pratique comme par exemple dans `a, b = a+b, a-b` et évite l'introduction d'une variable auxiliaire.)

Cela permet de calculer les premiers termes de la suite (u_n) :

$$\frac{3}{2}, \frac{5}{3}, \frac{9}{5}, \frac{17}{9}, \frac{33}{17}, \frac{65}{33}, \dots$$

Au vu des premiers termes, on a envie de dire que notre suite (u_n) coïncide avec la suite (v_n) définie par :

$$v_n = \frac{2^{n+1} + 1}{2^n + 1}.$$

On définit donc une fonction qui renvoie la valeur de v_n en fonction de n .

Code 80 (*suite-recurrente.sage (2)*).

```

def conjec(n):
    return (1+2^(n+1))/(1+2^n)

```

Pour renforcer notre conjecture, on vérifie que (u_n) et (v_n) sont égales pour les 20 premiers termes :

Code 81 (*suite-recurrente.sage (3)*).

```

for n in range(20):
    print n,' ',masuite(n),' ',conjec(n)

```

Il est à noter que l'on peut calculer u_n pour des valeurs aussi grandes que l'on veut de n mais que l'on n'a pas une formule directe pour calculer u_n en fonction de n . L'intérêt de (v_n) est de fournir une telle formule.

Voici maintenant une preuve assistée par l'ordinateur. Notre conjecture à démontrer est : pour tout $n \geq 0$, $v_n = u_n$. Pour cela, on va prouver que v_n et u_n coïncident sur les deux premiers termes et vérifient la même relation de récurrence. Ce seront donc deux suites égales.

- **Initialisation.** Vérifions que $v_0 = u_0$ et $v_1 = u_1$. Il suffit de vérifier que `conjec(0)-3/2` et `conjec(1)-5/3` renvoient tous les deux 0.
- **Relation de récurrence.** Par définition (u_n) vérifie la relation de récurrence $u_n = 1 + 4 \frac{u_{n-2}-1}{u_{n-1}u_{n-2}}$. Définissons une fonction correspondant à cette relation : à partir d'une valeur x (correspond à u_{n-1}) et y (correspondant à u_{n-2}) on renvoie $1 + 4 \frac{y-1}{xy}$.

Code 82 (*suite-recurrente.sage (4)*).

```

def recur(u_n_1,u_n_2):
    return 1+4*(u_n_2-1)/(u_n_1*u_n_2)

```

On demande maintenant à l'ordinateur de vérifier que la suite (v_n) vérifie aussi cette relation de récurrence :

Code 83 (*suite-recurrente.sage (5)*).

```

var('n')
test = conjec(n)-recur(conjec(n-1),conjec(n-2))
print(test.simplify_rational())

```

Le résultat obtenu est 0. Donc v_n vérifie pour tout n la relation $v_n = 1 + 4 \frac{v_{n-2}-1}{v_{n-1}v_{n-2}}$.

- **Conclusion.** Les suites (v_n) et (u_n) sont égales aux rangs 0 et 1 et vérifient la même relation de récurrence pour $n \geq 2$. Ainsi $u_n = v_n$, pour tout $n \in \mathbb{N}$. Les suites (u_n) et (v_n) sont donc égales.

Notez qu'il a fallu guider Sage pour simplifier notre résultat à l'aide de la fonction `simplify_rational` qui simplifie les fractions rationnelles.

Remarque.

Quelques commentaires sur les dernières lignes de code.

- La commande `var('n')` est essentielle. Elle permet de réinitialiser la variable n en une variable formelle. Souvenez-vous qu'auparavant n valait 19. Après l'instruction `var('n')` la variable n redevient une indéterminée 'n' sans valeur spécifique.
- Pour bien comprendre ce point, supposons que vous souhaitez vérifier que $(n + 2)^3 = n^3 + 6n^2 + 12n + 8$. Vous pouvez bien sûr le vérifier par exemple pour les vingt premiers rang $n = 0, 1, 2, \dots, 19$. Mais en définissant la variable formelle `var('n')` et à l'aide de la commande `expand((n+2)^3-n^3-6*n^2-12*n-8)` le logiciel de calcul formel «prouve» directement que l'égalité est vérifiée pour tout n . Si vous oubliez de réinitialiser la variable n en une variable formelle, vous ne l'aurez vérifiée que pour l'unique valeur $n = 19$!

4. Suites récurrentes et visualisation

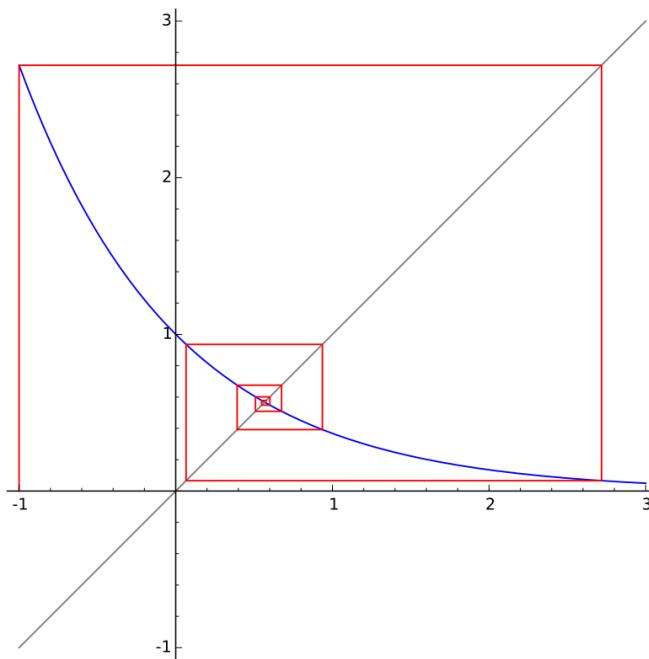
4.1. Visualiser une suite récurrente

Travaux pratiques 12.

Fixons $a \in \mathbb{R}$. Définissons une suite $(u_n)_{n \in \mathbb{N}}$ par récurrence :

$$u_0 = a \quad \text{et} \quad u_{n+1} = \exp(-u_n) \quad \text{pour } n \geq 0.$$

1. Calculer les premiers valeurs de la suite pour $a = -1$. Écrire une fonction qui renvoie ces premières valeurs sous la forme d'une liste.
2. Sur un même graphique tracer le graphe de la fonction de récurrence $f(x) = \exp(-x)$, la bissectrice ($y = x$) et la trace de la suite récurrente, c'est-à-dire la ligne brisée joignant les points $(u_k, f(u_k))$, (u_{k+1}, u_{k+1}) et $(u_{k+1}, f(u_{k+1}))$.
3. Émettre plusieurs conjectures : la suite (ou certaines sous-suites) sont-elles croissantes ou décroissantes ? Majorées, minorées ? Convergentes ?
4. Prouver vos conjectures.



1. On définit la fonction $f(x) = \exp(-x)$ par la commande `f(x) = exp(-x)`.

Code 84 (*suites-visual.sage (1)*).

```
def liste_suite(f,terme_init,n):
    maliste = []
    x = terme_init
    for k in range(n):
        maliste.append(x)
        x = f(x)
    return maliste
```

Par exemple la commande `liste_suite(f, -1, 4)` calcule, pour $a = -1$, les 4 premiers termes de la suite (u_n) ; on obtient la liste :

`[-1, e, e^(-e), e^(-e^(-e))]`

correspondant aux termes : $u_0 = -1$, $u_1 = e$, $u_2 = e^{-e}$ et $u_3 = e^{-e^{-e}}$, où l'on note $e = \exp(1)$.

2. Nous allons maintenant calculer une liste de points. On démarre du point initial $(u_0, 0)$, puis pour chaque rang on calcule deux points : $(u_k, f(u_k))$ et (u_{k+1}, u_{k+1}) . Notez que chaque élément de la liste est ici un couple (x, y) de coordonnées.

Code 85 (*suites-visual.sage (2)*).

```
def liste_points(f,terme_init,n):
    u = liste_suite(f,terme_init,n)
    mespoints = [ (u[0],0) ]
    for k in range(n-1):
        mespoints.append( (u[k],u[k+1]) )
        mespoints.append( (u[k+1],u[k+1]) )
    return mespoints
```

Par exemple la commande `liste_points(f, -1, 3)` calcule, pour $a = -1$, le point initial $(-1, 0)$ et les 3 premiers points de la visualisation de la suite (u_n) ; on obtient la liste :

$[-1, 0], (-1, e), (e, e), (e, e^{-e}), (e^{-e}, e^{-e})]$

Il ne reste plus qu'à tracer le graphe de la fonction et construire la suite en traçant les segments reliant les points.

Code 86 (*suites-visual.sage (3)*).

```
def dessine_suite(f,terme_init,n):
    mespoints = liste_points(f,terme_init,n)
    G = plot(f,(x,-1,3))      # La fonction
    G = G + plot(x,(x,-1,3))  # La droite (y=x)
    G = G + line(mespoints)    # La suite
    G.show()
```

Par exemple la figure illustrant ce tp est construite par la commande `dessine_suite(f,-1,10)`.

3. (et 4.) Passons à la partie mathématique. Pour simplifier l'étude on va supposer $a = -1$. Donc $u_0 = -1$ et $u_1 = f(u_0) = \exp(1) = e$.

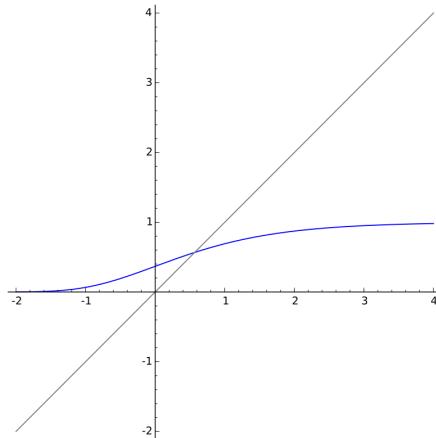
- (a) La fonction f définie par $f(x) = \exp(-x)$ est décroissante. Donc la fonction g définie par $g(x) = f(f(x))$ est croissante.
- (b) La suite $(u_{2n})_{n \in \mathbb{N}}$ des termes de rangs pairs est croissante. En effet : $u_2 = e^{-e} \geq -1 = u_0$. Alors $u_4 = f \circ f(u_2) \geq f \circ f(u_0) = u_2$ car $f \circ f = g$ est croissante. Puis par récurrence $u_6 = f \circ f(u_4) \geq f \circ f(u_2) = u_4 \dots$ La suite (u_{2n}) est croissante.
- (c) Comme $u_2 \geq u_0$ et que f est décroissante alors $u_3 \leq u_1$. On démontre alors de la même façon que la suite $(u_{2n+1})_{n \in \mathbb{N}}$ des termes de rangs impairs est décroissante.
- (d) Enfin, toujours par la même méthode et en partant de $u_0 \leq u_1$, on prouve que $u_{2n} \leq u_{2n+1}$.
- (e) La situation est donc la suivante :

$$u_0 \leq u_2 \leq \dots \leq u_{2n} \leq \dots \leq u_{2n+1} \leq \dots \leq u_3 \leq u_1$$

La suite (u_{2n}) est croissante et majorée par u_1 , donc elle converge. Notons ℓ sa limite.

La suite (u_{2n+1}) est décroissante et minorée par u_0 , donc elle converge. Notons ℓ' sa limite.

- (f) Par les théorèmes usuels d'analyse, la suite (u_{2n}) converge vers un point fixe de la fonction $g = f \circ f$, c'est-à-dire une valeur x_0 vérifiant $f \circ f(x_0) = x_0$. Une étude de la fonction g (ou plus précisément de $g(x) - x$) montrerait que g n'a qu'un seul point fixe. Voici le graphe de g .



Cela prouve en particulier que $\ell = \ell'$.

- (g) Par ailleurs la fonction f admet un unique point fixe x_0 . Mais comme $f(x_0) = x_0$ alors l'égalité $f(f(x_0)) = f(x_0) = x_0$ prouve que x_0 est aussi le point fixe de g .

- (h) Conclusion : la suite (u_{2n}) et la suite (u_{2n+1}) convergent vers $x_0 = \ell = \ell'$. Ainsi la suite (u_n) converge vers x_0 le point fixe de f .
- (i) Il n'y a pas d'expression simple de la solution x_0 de l'équation $f(x) = x$. La commande `solve(f(x)==x,x)` renvoie seulement l'équation. Par contre, on obtient une valeur numérique approchée par `find_root(f(x)==x,0,1)`. On trouve $x_0 = 0,56714329041\dots$

4.2. Listes

Une *liste* est ce qui ressemble le plus à une suite mathématique. Une liste est une suite de nombres, de points... ou même de listes !

- Une liste est présentée entre crochets : `mesprems = [2,3,5,7,11,13]`. On accède aux valeurs par `mesprems[i]` ; `mesprems[0]` vaut 2, `mesprems[1]` vaut 3...
- On parcourt les éléments d'une liste avec `for p in mesprems:`, `p` vaut alors successivement 2, 3, 5, 7...
- Une première façon de créer une liste est de partir de la liste vide, qui s'écrit `[]`, puis d'ajouter un à un des éléments à l'aide de la méthode `append`. Par exemple `mespoints=[]`, puis `mespoints.append((2,3))`, `mespoints.append((7,-1))`. La liste `mespoints` contient alors deux éléments (ici deux points) `[(2,3), (7,-1)]`.

Travaux pratiques 13.

Pour n un entier fixé. Composer les listes suivantes :

1. la liste des entiers de 0 à $n - 1$,
2. la liste des entiers premiers strictement inférieurs à n ,
3. la liste des $2p + 1$, pour les premiers p strictement inférieurs à n ,
4. les 10 premiers éléments de la liste précédente,
5. la liste de $p_i + i$, où $(p_i)_{i \geq 0}$ sont les nombres premiers strictement inférieurs à n ($p_0 = 2, p_1 = 3, \dots$),
6. la liste de 1 et 0 selon que le rang $0 \leq k < n$ soit premier ou pas.

Une utilisation intelligente permet un code très court et très lisible. Il faut potasser le manuel afin de manipuler les listes avec aisance.

1. `entiers = range(n)`
2. `premiers = [k for k in range(n) if is_prime(k)]`
3. `doubles = [2*p+1 for p in premiers]`
4. `debut = doubles[0:10]`
5. `premiersplusrang = [premiers[i]+i for i in range(len(premiers))]`
6. `binaire = [zeroun(k) for k in range(n)]` où `zeroun(k)` est une fonction à définir qui renvoie 1 ou 0 selon que k est premier ou pas.

On peut aussi trier les listes, les fusionner...

4.3. Suites et chaos

Travaux pratiques 14.

On considère la fonction f définie sur $x \in [0, 1]$ par

$$f(x) = rx(1-x) \quad \text{où} \quad 0 \leq r \leq 4.$$

Pour r fixé et $u_0 \in [0, 1]$ fixé, on définit une suite (u_n) par

$$u_{n+1} = f(u_n).$$

1. Pour différentes valeurs de r et u_0 fixées, tracer sur un même graphique le graphe de f , la droite d'équation ($y = x$) et la suite (u_n) . Essayez de conjecturer le comportement de la suite pour $0 < r \leq 3$, $3 < r \leq 1 + \sqrt{6}$, $1 + \sqrt{6} < r < 4$ et $r = 4$.
2. Pour u_0 fixé et $M < N$ grands (par exemple $M = 100$, $N = 200$) tracer le **diagramme de bifurcation** de f , c'est-à-dire l'ensemble des points

$$(u_i, r) \quad \text{pour } M \leq i \leq N \text{ et } 0 \leq r \leq 4$$

3. (a) Montrer que les points fixes de f sont 0 et $\frac{r-1}{r}$.
- (b) Montrer que le point fixe 0 est répulsif (c'est-à-dire $|f'(0)| > 1$) dès que $r > 1$.
- (c) Montrer que le point fixe $\frac{r-1}{r}$ est attractif (c'est-à-dire $|f'(\frac{r-1}{r})| < 1$) si et seulement si $1 < r < 3$.

4. **Cas $1 < r \leq 3$.** Pour $0 < u_0 < 1$ la suite (u_n) converge vers $\frac{r-1}{r}$.

On va prouver ce résultat seulement dans le cas particulier $r = 2$:

- (a) Montrer que $u_{n+1} - \frac{1}{2} = -2(u_n - \frac{1}{2})^2$.
- (b) Montrer que si $|u_0 - \frac{1}{2}| < k < \frac{1}{2}$ alors $|u_n - \frac{1}{2}| < \frac{1}{2}(2k)^{2^n}$.
- (c) Conclure.

5. **Cas $3 < r < 1 + \sqrt{6}$.** La suite (u_n) possède un cycle attracteur de période 2.

- (a) Déterminer les points fixes ℓ_1 et ℓ_2 de $f \circ f$ qui ne sont pas des points fixes de f .
- (b) Montrer que $f(\ell_1) = \ell_2$ et $f(\ell_2) = \ell_1$.
- (c) À l'aide du graphe de $f \circ f$, vérifier graphiquement sur un exemple, que les suites (u_{2n}) et (u_{2n+1}) sont croissantes ou décroissantes à partir d'un certain rang et convergent, l'une vers ℓ_1 , l'autre vers ℓ_2 .

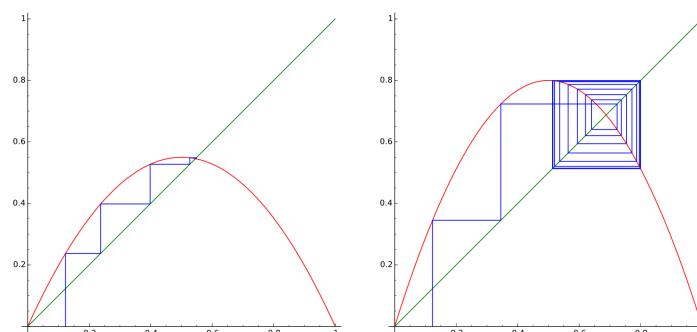
6. **Cas $1 + \sqrt{6} < r < 3,5699456\dots$** La suite (u_n) possède un cycle attracteur de période 4,8,16...

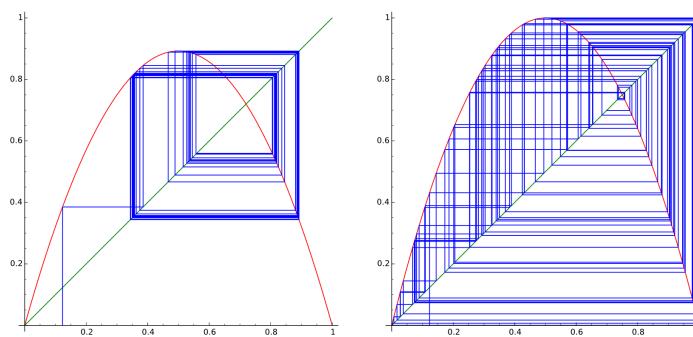
Trouver de tels exemples.

7. À partir de $r > 3,5699456\dots$ la suite (u_n) devient chaotique.

Le tp suivant sera consacré au cas $r = 4$.

1. Voici le comportement de différentes suites définies par le même $u_0 = 0,123$ et pour r valant successivement $r = 2,2$, $r = 3,2$, $r = 3,57$ et $r = 4$.



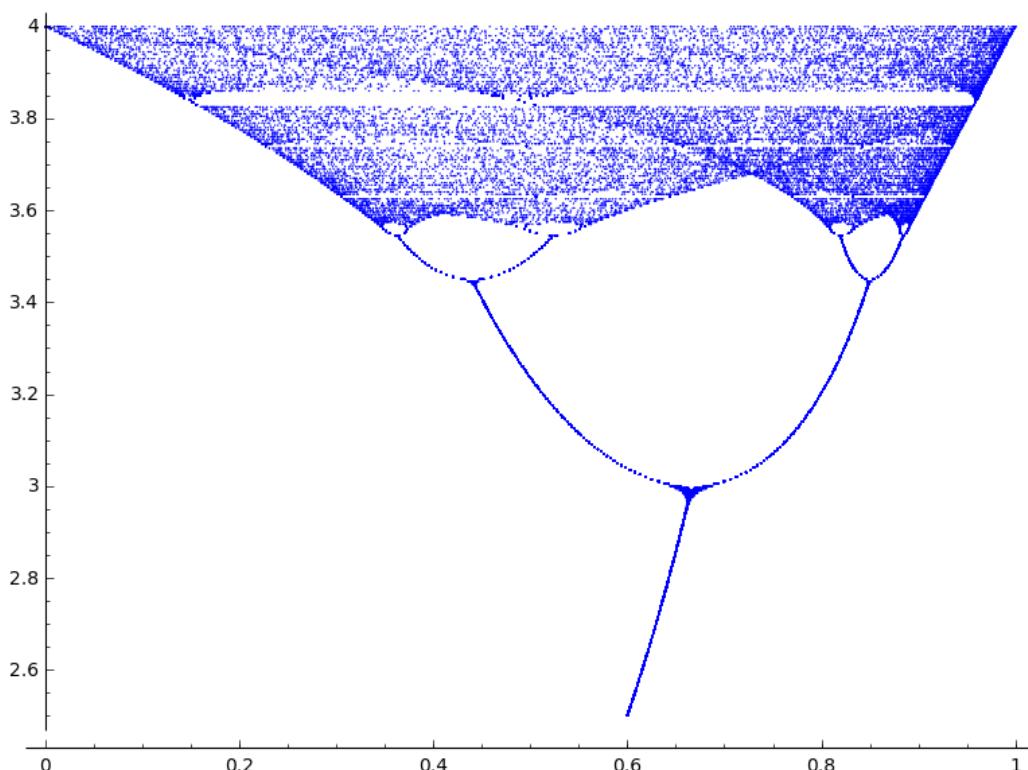


On voit d'abord une limite, puis la suite semble avoir « deux limites » c'est-à-dire deux valeurs d'adhérence, puis 4 valeurs d'adhérence. Pour $r = 4$ la situation semble chaotique.

2. Voici le code et le résultat de l'exécution `bifurcation(f, 0.102)`, où $f(x, r) = r*x*(1-x)$.

Code 87 (*suites-chaos.sage* (1)).

```
def bifurcation(F,terme_init):
    Nmin = 100           # On oublie Nmin premiers termes
    Nmax = 200           # On conserve les termes entre Nmin et Nmax
    epsilon = 0.005       # On fait varier r de epsilon à chaque pas
    r = 2.5              # r initial
    mespoints = []
    while r <= 4.0:
        u = liste_suite(F(r=r),terme_init,Nmax) # On calcule la suite
        for k in range(Nmin,Nmax):
            mespoints = mespoints + [(u[k],r)]
        r = r + epsilon
    G = points(mespoints)
    G.show()
```



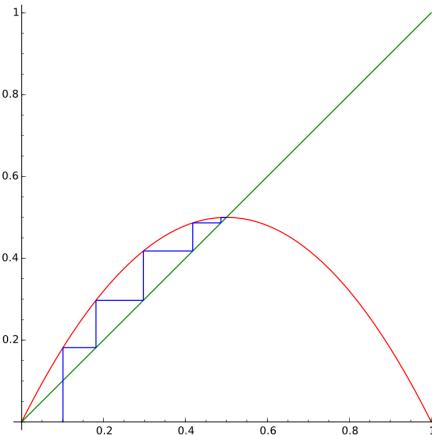
3. Voici le code pour les trois questions.

Code 88 (*suites-chaos.sage* (2)).

```
var('x,r')
f(x,r) = r*x*(1-x)
pts_fixes = solve(f==x,x)      # (a) Points fixes
ff = diff(f,x)                 # Dérivée
ff(x=0)                         # (b)  $f'(0)$ 
solve(abs(ff(x=(r-1)/r))<1,r) # (c) Condition point attractif
```

- (a) Les deux points fixes fournis dans la liste `pts_fixes` sont 0 et $\frac{r-1}{r}$.
- (b) On calcule la dérivée `ff`, on l'évalue en 0, on trouve $f'(0) = r$. Ainsi si $r > 1$ alors $|f'(0)| > 1$ et le point 0 est répulsif.
- (c) Par contre lorsque l'on résout l'inéquation $|f'(\frac{r-1}{r})| < 1$, la machine renvoie les conditions $r > 1$ et $r < 3$. Ainsi pour $1 < r < 3$, le point fixe $\frac{r-1}{r}$ est attractif.
4. On pose d'abord $r = 2$, alors le point fixe est $\frac{r-1}{r} = \frac{1}{2}$.
- (a) Après simplification $(r*u*(1-u) - 1/2) + 2*(u-1/2)^2$ vaut 0. Autrement dit $u_{n+1} - \frac{1}{2} = -2(u_n - \frac{1}{2})^2$, quelque soit u_n .
- (b) C'est une simple récurrence :
- $$\left| u_{n+1} - \frac{1}{2} \right| = 2 \left(u_n - \frac{1}{2} \right)^2 < 2 \left(\frac{1}{2}(2k)^{2^n} \right)^2 = \frac{1}{2}(2k)^{2^{n+1}}.$$
- (c) Ainsi pour $u_0 \neq 0, 1$, il existe k tel que $|u_0 - \frac{1}{2}| < k < \frac{1}{2}$ et la suite (u_n) tend (très très) vite vers le point fixe $\frac{1}{2}$.

Avec $r = 2$, en quelques itérations le terme de la suite n'est plus discernable de la limite $\frac{1}{2}$:



5. Voici le code pour les deux premières questions :

Code 89 (*suites-chaos.sage* (3)).

```
var('x,r')
f(x,r) = r*x*(1-x)          # f
g = f(x=f(x,r))              # g(x) = f(f(x))
pts_doubles = solve(g==x,x)    # (a) Points fixes de g
# Les deux nouveaux points fixes de g :
ell1 = pts_doubles[0].rhs()
ell2 = pts_doubles[1].rhs()
```

```

eq = f(x=ell1)-ell2          # (b) l'image de ell1 est-elle ell2 ?
eq.full_simplify()           # Oui !

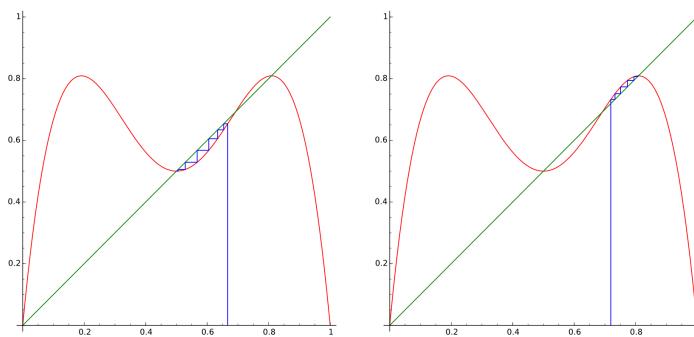
```

- (a) On calcule les points fixes de $g = f \circ f$. Il y en a quatre, mais deux d'entre eux sont les points fixes de f . Les deux nouveaux sont :

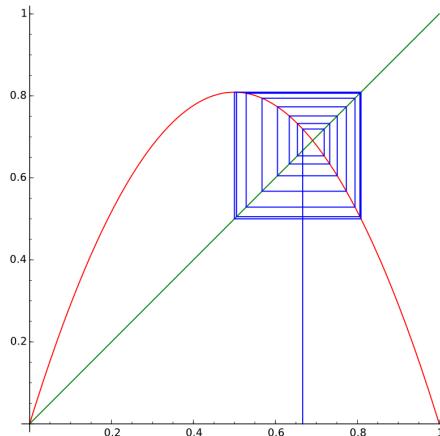
$$\ell_1 = \frac{1}{2} \frac{r+1-\sqrt{r^2-2r-3}}{r} \quad \ell_2 = \frac{1}{2} \frac{r+1+\sqrt{r^2-2r-3}}{r}$$

(b) Bien sûr ℓ_1 et ℓ_2 ne sont pas des points fixes pour f . Par contre, on vérifie que $f(\ell_1) = \ell_2$ et $f(\ell_2) = \ell_1$.

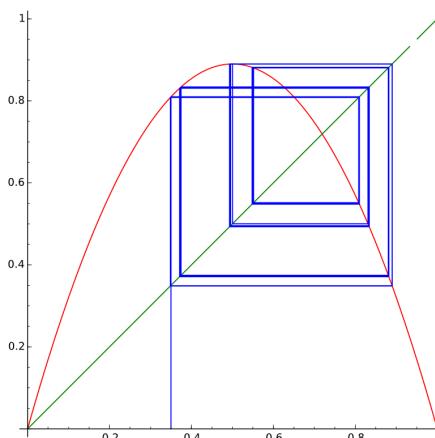
(c) Voici les dessins pour $r = 1 + \sqrt{5}$ et $u_0 = \frac{2}{3}$. La suite (u_{2n}) , qui est vue comme suite récurrente de fonction g , est décroissante vers ℓ_1 (figure de gauche). La suite (u_{2n+1}) , vue comme suite récurrente de fonction g , est croissante vers ℓ_2 (figure de droite).



La suite (u_n) , comme suite récurrente de fonction f , possède le cycle (ℓ_1, ℓ_2) comme cycle attracteur :



6. Voici un exemple de cycle de longueur 8 ($r = 3,56$, $u_0 = 0,35$) :



7. Le cas $r = 4$ sera étudié dans le tp suivant.

Travaux pratiques 15.

On s'intéresse au cas $r = 4$. On rappelle que

$$f(x) = rx(1-x) \quad u_0 \in [0, 1] \quad \text{et} \quad u_{n+1} = f(u_n) \quad (n \geq 0).$$

1. (a) Montrer que $\sin^2(2t) = 4\sin^2 t \cdot (1 - \sin^2 t)$.
- (b) Montrer que pour $u_0 \in [0, 1]$, il existe un unique $t \in [0, \frac{\pi}{2}]$ tel que $u_0 = \sin^2 t$.
- (c) En déduire $u_n = \sin^2(2^n t)$.
2. Pour $t_k = \frac{2\pi}{2^k+1}$ et $u_0 = \sin^2 t_k$, montrer que la suite (u_n) est périodique de période k .
3. **La suite est instable.**
 - (a) Pour $k = 3$, on pose $t_3 = \frac{2\pi}{9}$, $u_0 = \sin^2 t_3$. Calculer une valeur exacte (puis approchée) de u_n , pour tout n .
 - (b) Partant d'une valeur approchée \tilde{u}_0 de u_0 , que donne la suite des approximations (\tilde{u}_n) définie par la relation de récurrence ?
4. **La suite est partout dense.**

On va construire $u_0 \in [0, 1]$ tel que, tout $x \in [0, 1]$ peut être approché d'aussi près que l'on veut par un terme u_n de notre suite :

$$\forall x \in [0, 1] \quad \forall \epsilon > 0 \quad \exists n \in \mathbb{N} \quad |x - u_n| < \epsilon.$$

- Soit σ la constante binaire de Champernowne, formée de la juxtaposition des entiers en écriture binaire à 1, 2, 3, ... chiffres 0, 1, 00, 01, 10, ... dont l'écriture binaire est $\sigma = 0,0100011011\dots$
- Soit $u_0 = \sin^2(\pi\sigma)$.
- Soit $x \in [0, 1[$. Ce réel peut s'écrire $x = \sin^2(\pi t)$ avec $t \in [0, 1[$ qui admet une écriture binaire $t = 0, a_1 a_2 \dots a_p \dots$

Pour $\epsilon > 0$ fixé, montrer qu'il existe n tel que $|x - u_n| < \epsilon$.

1. (a) On pose `eq = sin(2*t)^2 - 4*sin(t)^2*(1-sin(t)^2)` et `eq.full_simplify()` renvoie 0. Donc $\sin^2(2t) = 4\sin^2 t \cdot (1 - \sin^2 t)$, pour tout $t \in \mathbb{R}$.
- (b) Soit $h(t) = \sin^2 t$. On définit la fonction `h = sin(t)^2`, sa dérivée `hh = diff(h, t)` dont on cherche les zéros par `solve(hh==0, t)`. La dérivée ne s'annulant qu'en $t = 0$ et $t = \frac{\pi}{2}$. La fonction h' est strictement monotone sur $[0, \frac{\pi}{2}]$. Comme $h(0) = 0$ et $h(\frac{\pi}{2}) = 1$, pour chaque $u_0 \in [0, 1]$, il existe un unique $t \in [0, \frac{\pi}{2}]$ tel que $u_0 = h(t)$.

- (c) Pour $u_0 = \sin^2 t$, on a

$$u_1 = f(u_0) = ru_0(1-u_0) = 4\sin^2 t \cdot (1 - \sin^2 t) = \sin^2(2t).$$

On montre de même par récurrence que $u_n = \sin^2(2^n t)$.

2. Le code suivant calcule $u_k - u_0$. Sage sait calculer que cela vaut 0, pour k prenant la valeur 0, 1, 2, ... mais pas lorsque k est une variable. Le calcul à la main est pourtant très simple !

Code 90 (*suites-chaos.sage (4)*).

```
t = 2*pi/(2^(k+1))
u_0 = sin(t)^2
u_k = sin(2^k*t-2*pi)^2
zero = u_k-u_0
zero.simplify_trig()
```

3. **La suite est instable.**

- (a) Pour $k = 3$ et $t_3 = \frac{2\pi}{9}$, la suite (u_k) est périodique de période 3. Donc il suffit de calculer u_0, u_1, u_2 .
Par exemple :

$$u_{3p} = u_0 = \sin^2\left(\frac{2\pi}{9}\right) \simeq 0,413\,175\,911\,1\dots$$

- (b) Partons de la valeur approchée de u_0 avec 10 décimales exactes : $\tilde{u}_0 = 0,413\,175\,911\,1$ et calculons les premiers termes de la suite. On en extrait :

$$\begin{aligned}\tilde{u}_0 &\simeq 0,413\,175\,911\,100 \\ \tilde{u}_3 &\simeq 0,413\,175\,911\,698 \\ \tilde{u}_6 &\simeq 0,413\,175\,906\,908 \\ \tilde{u}_9 &\simeq 0,413\,175\,945\,232 \\ \tilde{u}_{12} &\simeq 0,413\,175\,638\,640 \\ \tilde{u}_{15} &\simeq 0,413\,178\,091\,373 \\ \tilde{u}_{18} &\simeq 0,413\,158\,469\,568 \\ \tilde{u}_{21} &\simeq 0,413\,315\,447\,870 \\ \tilde{u}_{24} &\simeq 0,412\,059\,869\,464 \\ \tilde{u}_{27} &\simeq 0,422\,119\,825\,829 \\ \tilde{u}_{30} &\simeq 0,342\,897\,499\,745 \\ \tilde{u}_{33} &\simeq 0,916\,955\,513\,784 \\ \tilde{u}_{36} &\simeq 0,517\,632\,311\,613 \\ \tilde{u}_{39} &\simeq 0,019\,774\,022\,431\end{aligned}$$

Si l'approximation de départ et tous les calculs étaient exacts, on devrait obtenir u_0 à chaque ligne. On constate que l'erreur augmente terme après terme, et après \tilde{u}_{30} , le comportement de \tilde{u}_n n'a plus rien à voir avec u_n . L'explication vient de la formule $u_n = \sin^2(2^n t)$, une erreur sur t , même infime au départ, devient grande par multiplication par 2^n .

4. La suite est partout dense.

La construction est assez jolie mais délicate. (Il ne faut pas avoir peur de l'écriture en base 2 qui n'est pas ici le point clé. Vous pouvez penser en base 10 pour mieux comprendre.)

Fixons $\epsilon > 0$. Soit un entier p tel que $\frac{\pi}{2^p} < \epsilon$. L'écriture binaire de t étant $t = 0, a_1 a_2 \dots a_p \dots$ la séquence $a_1 a_2 \dots a_p$ apparaît quelque part dans la constante de Champernowne. Plus précisément il existe un entier n tel que $2^n \sigma = \dots b_2 b_1 b_0, a_1 a_2 \dots a_p \dots$ (on a fait apparaître la séquence juste après la virgule par décalage de la virgule).

On va pouvoir oublier la partie entière $m = \dots b_2 b_1 b_0 \in \mathbb{N}$ et on note $\tilde{\sigma} = 0, a_1 a_2 \dots a_p \dots$ la partie fractionnaire de σ , de sorte que $2^n \sigma = m + \tilde{\sigma}$.

$$\begin{aligned}|x - u_n| &= |\sin^2(\pi t) - \sin^2(2^n \pi \sigma)| \\ &= |\sin^2(\pi t) - \sin^2(2^n \pi m + \pi \tilde{\sigma})| \\ &= |\sin^2(\pi t) - \sin^2(\pi \tilde{\sigma})| \quad \text{car } \sin^2 \text{ est } \pi\text{-périodique} \\ &\leq |\pi t - \pi \tilde{\sigma}| \quad \text{par le théorème des accroissements finis} \\ &\leq \pi \frac{1}{2^p} \quad \text{car } t \text{ et } \tilde{\sigma} \text{ ont les mêmes } p \text{ premières décimales} \\ &< \epsilon\end{aligned}$$

Conclusion : n'importe quel x est approché d'aussi près que l'on veut par la suite.

Pour en savoir plus :

- *La suite logistique et le chaos*, Daniel Perrin.
- *Étude d'une suite récurrente*, Jean-Michel Ferrard.

5. Algèbre linéaire

L'algèbre linéaire est la partie des mathématiques qui s'occupe des matrices et des structures vectorielles. Beaucoup de problèmes se ramènent ou s'approchent par des problèmes linéaires, pour lesquels il existe souvent des solutions efficaces.

5.1. Opérations de base

Travaux pratiques 16.

Soient les matrices :

$$A = \begin{pmatrix} 1 & 2 & 3 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \quad u = \begin{pmatrix} 1 \\ x \\ x^2 \end{pmatrix} \quad v = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

1. Calculer tous les produits possibles à partir de A, B, u, v .
 2. Calculer $(A - I)^7$ et en extraire le coefficient en position (2, 3).
 3. Calculer A^{-1} . Calculer la trace de A^{-1} (c'est-à-dire la somme des coefficients sur la diagonale).
1. On définit une matrice n lignes et p colonnes par la commande
`matrix(n,p,[[ligne1], [ligne2], ...])`

Pour nous cela donne :

Code 91 (*alglin.sage* (1)).

```
A = matrix(3,3,[[1,2,3],[-1,0,1],[0,1,0]])
B = matrix(2,3,[[2,-1,0],[-1,0,1]])
I = identity_matrix(3)
u = matrix(3,1,[1,x,x^2])
v = matrix(3,1,[1,0,1])
```

On multiplie deux matrices par $A * B$. Cette opération est possible seulement si le nombre de colonnes de A est égal au nombre de lignes de B . Les produits possibles sont donc ici : $B \times A, A \times u, A \times v, B \times u, B \times v$.

2. Ayant défini la matrice identité I (par `I = identity_matrix(3)`), on calcule $M = (A - I)^7$ par `(A-I)^7`, ce qui renvoie

$$(A - I)^7 = \begin{pmatrix} -16 & -46 & 11 \\ -25 & -73 & 153 \\ 32 & 57 & -137 \end{pmatrix}.$$

Le coefficient en position (2, 3) d'une matrice M est celui sur la deuxième ligne et la troisième colonne. Donc ici c'est 153. Une matrice A se comporte en Sage comme une liste à double entrée et on accède aux coefficients par la commande `A[i, j]` (i pour les lignes, j pour les colonnes). Attention ! Les listes étant indexées à partir du rang 0, les indices i, j partent de 0. Le coefficient en position (2, 3) s'obtient donc par la commande :

```
((A-I)^7)[1,2]
```

Faites bien attention au décalage des deux indices.

3. L'inverse d'une matrice A se calcule par `A^-1` ou `A.inverse()`. Ce qui donne

$$A^{-1} = \begin{pmatrix} \frac{1}{4} & -\frac{3}{4} & -\frac{1}{2} \\ 0 & 0 & 1 \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{2} \end{pmatrix}.$$

La trace se calcule par une simple somme des éléments sur la diagonale principale :

```
sum( (A^-1)[i,i] for i in range(3) )
```

et vaut ici $-\frac{1}{4}$.

Remarque. • Noter encore une fois que pour une matrice de taille n et pour Sage les indices varient de 0 à $n - 1$.

- On peut préciser le corps sur lequel on travaille, pour nous ce sera le plus souvent $K = \mathbb{Q}$. Par exemple : `A = matrix(QQ, [[1,2],[3,4]])`.
- Avec Sage on peut aussi définir des vecteurs par `vector([x1,x2,...,xn])`. Un vecteur peut représenter soit un vecteur ligne, soit un vecteur colonne. On peut multiplier une matrice par un vecteur (à gauche ou à droite).
- Si on définit un vecteur, par exemple `v = vector([1,2,3,4])` alors `L = matrix(v)` renvoie la matrice ligne correspondante. `C = L.transpose()` renvoie la matrice colonne correspondante.

Travaux pratiques 17.

Pour une matrice $A \in M_n(\mathbb{K})$ inversible et des vecteurs colonnes u, v à n lignes on définit $\alpha \in \mathbb{K}$ par :

$$\alpha = 1 + v^T A^{-1} u.$$

La formule de Sherman-Morrison affirme que si $\alpha \neq 0$ alors

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{\alpha}$$

Prouver par le calcul formel cette formule pour les matrices de taille 2×2 et 3×3 .

Connaissant l'inverse de A , cette formule permet de calculer à moindre coût l'inverse d'une déformation de A par une matrice de rang 1. Le code ne pose pas de problème.

Code 92 (`alglin.sage` (2)).

```
var('a,b,c,d,x,y,xx,yy')
A = matrix(2,2,[[a,b],[c,d]])
u = matrix(2,1,[x,y])
v = matrix(2,1,[xx,yy])

alpha_matrice = 1 + v.transpose() * (A^-1) * u
alpha_reel = alpha_matrice[0,0]
B = ( A + u * v.transpose() )^-1
BB = A^-1 - 1/alpha_reel*( A^-1 * u * v.transpose() * A^-1 )
C = B - BB

for i in range(A.nrows()):
    for j in range(A.ncols()):
        print("indices:", i, j, "coeff:", (C[i,j]).full_simplify())
```

- On commence par définir les coefficients qui seront nos variables (ici pour $n = 2$).
- On calcule α par la formule $\alpha = 1 + v^T A^{-1} u$, qui est une matrice de taille 1×1 (`alpha_matrice`), identifiée à un réel (`alpha_reel`).
- On calcule ensuite les termes de gauche (B) et droite (BB) de la formule à prouver. Pour la matrice `C = B - BB`, on vérifie (après simplification) que tous ses coefficients sont nuls. Ce qui prouve l'égalité cherchée.
- On pourrait également obtenir directement le résultat en demandant à Sage :
`C.simplify_rational()==matrix(2,2,0)`.

Pour une preuve à la main et en toute dimension, multiplier $(A + uv^T)$ par $A^{-1} - \frac{A^{-1}uv^TA^{-1}}{\alpha}$.

5.2. Réduction de Gauss, calcul de l'inverse

Le but de cette section est de mettre en œuvre la méthode de Gauss. Cette méthode est valable pour des systèmes linéaires (ou des matrices) de taille quelconque, mais ici on se limite au calcul de l'inverse d'une matrice (qui est obligatoirement carrée). N'hésitez pas à d'abord relire votre cours sur la méthode de Gauss.

Travaux pratiques 18.

1. Réaliser trois fonctions qui transforment une matrice en fonction des opérations élémentaires sur les lignes :
 - $L_i \leftarrow cL_i$ avec $c \neq 0$: on multiplie une ligne par un réel ;
 - $L_i \leftarrow L_i + cL_j$: on ajoute à la ligne L_i un multiple d'une autre ligne L_j ;
 - $L_i \leftrightarrow L_j$: on échange deux lignes.
2. À l'aide de ces transformations élémentaires, transformer par la méthode de Gauss une matrice inversible en une matrice échelonnée (c'est-à-dire ici, en partant d'une matrice carrée, obtenir une matrice triangulaire supérieure).
3. Toujours à l'aide de ces transformations et de la méthode de Gauss, transformer cette matrice échelonnée en une matrice échelonnée et réduite (c'est-à-dire ici, en partant d'une matrice inversible, obtenir l'identité).
4. La méthode de Gauss permet de calculer l'inverse d'une matrice A :
 - Partant de A , on pose $B = I$ la matrice identité.
 - Par la méthode de Gauss et des opérations élémentaires on transforme A en la matrice I .
 - On applique exactement les mêmes transformations à la matrice B .
 - Lorsque A s'est transformée en I alors B s'est transformée en A^{-1} .

Modifier légèrement vos deux fonctions (issues des questions 2. et 3.) afin de calculer l'inverse d'une matrice.

Voici une matrice A , sa forme échelonnée, sa forme échelonnée réduite (l'identité), et son inverse :

$$A = \begin{pmatrix} 1 & 2 & 3 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & 4 \\ 0 & 0 & -2 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad A^{-1} = \begin{pmatrix} \frac{1}{4} & -\frac{3}{4} & -\frac{1}{2} \\ 0 & 0 & 1 \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{2} \end{pmatrix}$$

1. Voici la fonction pour la deuxième opération : $L_i \leftarrow L_i + cL_j$.

Code 93 (*gauss.sage (1)*).

```
def op2(A,i,j,c):
    A[i,:] = A[i,:] + c*A[j,:]
    return A
```

$A[i,:]$ correspond à la ligne i (les deux points signifiant de prendre tous les indices des colonnes). Lorsque l'on execute cette fonction sur une matrice A cela modifie la matrice (voir la remarque plus bas). Pour éviter cela on commence par faire une copie de A par $AA = \text{copy}(A)$, puis on exécute la fonction sur cette copie, avec par exemple $op2(AA,0,2,-1)$ pour l'opération $L_0 \leftarrow L_0 - L_2$.

Les autres opérations $op1(A,i,c)$, $op3(A,i,j)$ se définissent de manière analogue.

2. Il s'agit de traduire la première partie de la méthode de Gauss. Pour chaque colonne p en partant de la plus à gauche :
 - (a) on cherche un coefficient non nul dans cette colonne ;
 - (b) on place ce coefficient en position de pivot (p,p) , en permutant deux lignes par la troisième opération élémentaire ;

(c) on annule, un par un, les coefficients qui sont sous ce pivot, par des opérations élémentaires :

$$L_i \leftarrow L_i + cL_p \text{ où } c = -\frac{a_{i,p}}{a_{p,p}}.$$

Code 94 (*gauss.sage (2)*).

```
def echelonne(A):
    n = A.ncols() # taille de la matrice
    for p in range(n): # pivot en A[p,p]
        # a. On cherche un coeff non nul
        i = p
        while i < n and A[i,p] == 0:
            i = i+1
        if i >= n :
            return matrix(n,n) # stoppe ici car pas inversible
        # b. On place la ligne avec coeff non nul en position de pivot
        A = op3(A,p,i)
        # c. On soustrait la ligne du pivot aux lignes en dessous
        for i in range(p+1,n):
            c = -A[i,p]/A[p,p]
            A = op2(A,i,p,c)
    return A
```

3. Pour passer à une forme réduite, on parcourt les colonnes en partant de la droite :

- (a) on commence par multiplier la ligne du pivot pour avoir un pivot valant 1 ;
- (b) on annule, un par un, les coefficients qui sont au-dessus de ce pivot.

Code 95 (*gauss.sage (3)*).

```
def reduite(A):
    n = A.ncols()
    for p in range(n-1,-1,-1):
        # a. On divise pour avoir un pivot valant 1
        c = 1/A[p,p]
        A = op1(A,p,c)
        # b. On élimine les coefficients au-dessus du pivot
        for i in range(p-1,-1,-1):
            c = -A[i,p]
            A = op2(A,i,p,c)
    return A
```

4. Pour calculer l'inverse, on définit deux nouvelles fonctions `echelonne_bis(A,B)` et `reduite_bis(A,B)` qui renvoient chacune, deux matrices modifiées de A et B . À chaque fois que l'on effectue une opération sur A , on effectue la même opération sur B . Par exemple, la dernière boucle de `echelonne_bis` contient la ligne $A = op2(A,i,p,c)$ et la ligne $B = op2(B,i,p,c)$. C'est bien le même coefficient $c = -\frac{a_{i,p}}{a_{p,p}}$ pour les deux opérations. On obtient alors l'inverse comme ceci :

Code 96 (*gauss.sage (4)*).

```
def inverse_matrice(A):
    n = A.ncols()
    B = identity_matrix(QQ,n)
```

```
A,B = echelonne_bis(A,B)
A,B = reduite_bis(A,B)
return B
```

Remarque.

Si on pose $A = \text{matrix}([[1, 2], [3, 4]])$, puis $B = A$, puis que l'on modifie la matrice A par $A[0, 0] = 7$. Alors la matrice B est aussi modifiée !

Que se passe-t-il ?

- A ne contient pas les coefficients de la matrice, mais l'adresse où sont stockés ces coefficients (c'est plus léger de manipuler l'adresse d'une matrice que toute la matrice). Comme A et B pointent vers la même zone qui a été modifiée, les deux matrices A et B sont modifiées.
- Pour pouvoir définir une copie de A , avec une nouvelle adresse mais les mêmes coefficients, on écrit $B = \text{copy}(A)$. Les modifications sur A ne changeront plus B .

5.3. Application linéaire, image, noyau

Travaux pratiques 19.

Soit $f : \mathbb{R}^3 \rightarrow \mathbb{R}^4$ l'application linéaire définie par

$$f : \begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} x + 2z \\ 5x + 2y + 2z \\ 2x + y \\ x + y - 2z \end{pmatrix}.$$

1. Expliciter la matrice A de f (dans les bases canoniques). Comment s'écrit alors l'application f ?
2. Calculer une base du noyau de f . L'application linéaire f est-elle injective ?
3. Calculer une base de l'image de f . L'application linéaire f est-elle surjective ?

1. La matrice associée est

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 5 & 2 & 2 \\ 2 & 1 & 0 \\ 1 & 1 & -2 \end{pmatrix}.$$

L'application linéaire f étant alors définie par $X \mapsto Y = AX$. Voici l'implémentation avec un exemple de calcul.

Code 97 (*matlin.sage (1)*).

```
K = QQ
A = matrix(K,4,3,[[1,0,2],[5,2,2],[2,1,0],[1,1,-2]])
X = vector(K,[2,3,4])
Y = A*X
```

2. et 3. Le noyau s'obtient par la commande `right_kernel()` et l'image par la commande `column_space()` car l'image est exactement le sous-espace vectoriel engendré par les vecteurs colonnes de la matrice.

Code 98 (*matlin.sage (2)*).

```
Ker = A.right_kernel()
Ker.basis()
Im = A.column_space()
```

Im.basis()

- Le noyau est un espace vectoriel de dimension 1 dans \mathbb{R}^3 engendré par $\begin{pmatrix} 2 \\ -4 \\ -1 \end{pmatrix}$. Le noyau n'étant pas trivial, l'application f n'est pas injective.
- L'image est un espace vectoriel de dimension 2 dans \mathbb{R}^4 , dont une base est :

$$\left(\begin{pmatrix} 1 \\ 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 2 \\ 1 \\ 1 \end{pmatrix} \right).$$

L'image n'étant pas tout \mathbb{R}^4 , l'application f n'est pas non plus surjective.

- On peut tester si un vecteur donné est dans un sous-espace. Si on pose par exemple $X = \text{vector}(K, [1, 5, 2, 1])$ alors le test « $X \in \text{Im}$ » renvoie vrai. Ce vecteur est donc bien un élément de l'image de f .

Attention ! Il ne faut pas utiliser directement les méthodes `kernel()` et `image()` car Sage préfère travailler avec les vecteurs lignes et donc ces méthodes calculent le *noyau à gauche* (c'est-à-dire l'ensemble des X tels que $XA = 0$) et l'*image à gauche* (c'est-à-dire l'ensemble des $Y = XA$). Si vous souhaitez utiliser ces méthodes pour calculer le noyau et l'image avec notre convention habituelle (c'est-à-dire à droite) il faut le faire sur la transposée de A :

Code 99 (*matlin.sage (3)*).

```
AA = A.transpose()
Ker = AA.kernel()
Ker.basis()
Im = AA.image()
Im.basis()
```

5.4. Méthodes des moindres carrés

Si on veut résoudre un système linéaire avec plus d'équations que d'inconnues alors il n'y a pas de solution en général. On aimerait pourtant parfois trouver ce qui s'approche le plus d'une solution. Formalisons ceci : soit $A \in M_{n,p}(\mathbb{R})$ avec $n \geq p$ une matrice à coefficients réels, X un vecteur inconnu de taille p et B un vecteur de taille n . Il n'y a en général pas de solution X au système $AX = B$, mais on aimerait que $AX - B$ soit le plus proche du vecteur nul. Ainsi, une **solution des moindres carrés** est un vecteur X tel que $\|AX - B\|$ soit minimale, où la norme considérée est la norme euclidienne. Cette solution des moindres carrés est donnée par la formule :

$$X = (A^T A)^{-1} A^T B \quad (\dagger)$$

(si $n \geq p$ et A est de rang maximal p , ce qu'on suppose, alors on peut montrer que la matrice $A^T A$ est inversible).

Travaux pratiques 20.

- Régression linéaire.** On considère les points $\{(-6, 0), (-2, 1), (1, 2), (4, 4), (6, 4)\}$. Trouver la droite qui approche au mieux ces points (au sens des moindres carrés).

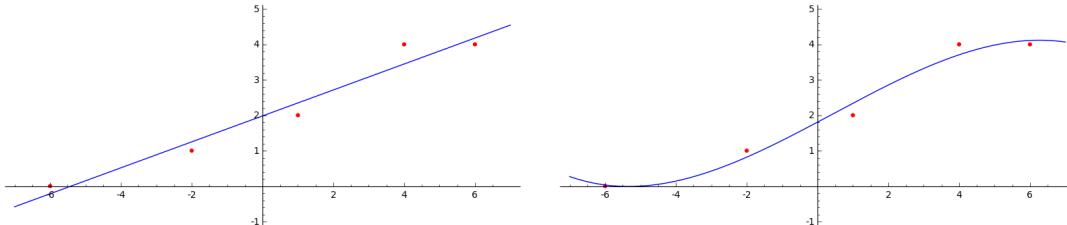
Indications. On pose $f(x) = a + bx$. On aimerait trouver $a, b \in \mathbb{R}$ tels que $f(x_i) = y_i$ pour tous les points (x_i, y_i) donnés. Transformer le problème en un problème des moindres carrés : quel est le vecteur inconnu X ? quelle est la matrice A ? quel est le second membre B ?

- Interpolation polynomiale.** Pour ces mêmes points, quel polynôme de degré 3 approche au mieux

ces points (au sens des moindres carrés).

Indication. Cette fois $f(x) = a + bx + cx^2 + dx^3$.

Voici la droite demandée (à gauche) ainsi que le polynôme de degré 3 (à droite).



1. Bien sûr les points ne sont pas alignés, donc il n'existe pas de droite passant par tous ces points. On cherche une droite d'équation $y = a + bx$ qui minimise (le carré de) la distance entre les points et la droite. Posons $f(x) = a + bx$.

Alors pour nos n points (x_i, y_i) donnés, on voudrait $f(x_i) = y_i$.

$$\begin{cases} f(x_1) = y_1 \\ f(x_2) = y_2 \\ \vdots \\ f(x_n) = y_n \end{cases} \iff \begin{cases} a + bx_1 = y_1 \\ a + bx_2 = y_2 \\ \vdots \\ a + bx_n = y_n \end{cases} \iff \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \iff AX = B.$$

On résout (de façon non exacte) notre problème $AX = B$, par la formule des moindres carrés $X = (A^T A)^{-1} A^T B$. Comme $X = \begin{pmatrix} a \\ b \end{pmatrix}$, on obtient l'équation $y = a + bx$ de la droite des moindres carrés (voir la figure ci-dessus).

La fonction `moindres_carres` résout le problème général des moindres carrés. Il reste ensuite à définir la matrice A et le vecteur B en fonction de notre liste de points afin de trouver X . Le code est le suivant :

Code 100 (`moindres_carres.sage(1)`).

```
def moindres_carres(A,B):
    return (A.transpose()*A)^-1 * A.transpose() * B

points = [(-6,0),(-2,1),(1,2),(4,4),(6,4)]

A = matrix([ [1,p[0]] for p in points ])
B = vector(p[1] for p in points)
X = moindres_carres(A,B)
```

2. L'idée est la même, mais cette fois les inconnues sont les coefficients de la fonction $f(x) = a + bx + cx^2 + dx^3$:

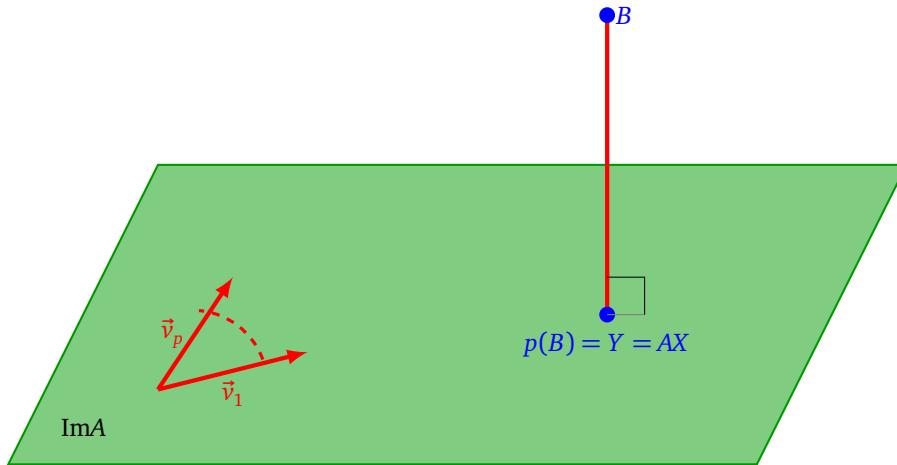
$$\begin{cases} f(x_1) = y_1 \\ f(x_2) = y_2 \\ \vdots \\ f(x_n) = y_n \end{cases} \iff \begin{cases} a + bx_1 + cx_1^2 + dx_1^3 = y_1 \\ a + bx_2 + cx_2^2 + dx_2^3 = y_2 \\ \vdots \\ a + bx_n + cx_n^2 + dx_n^3 = y_n \end{cases} \iff \begin{pmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \iff AX = B$$

Voici le code pour les mêmes points et une interpolation polynomiale de degré d :

Code 101 (*moindres_carres.sage* (2)).

```
d = 3      # degré
A = matrix([ [p[0]^k for k in range(d+1)] for p in points ])
B = vector(p[1] for p in points)
X = moindres_carres(A,B)
```

Pour conclure, voici la preuve de la formule (†) des moindres carrés. Caractérisons géométriquement la condition « $\|AX - B\|$ minimale ». On note $\text{Im}A$ l'image de A , c'est-à-dire le sous-espace vectoriel engendré par les vecteurs colonnes de la matrice A . Notons $p : \mathbb{R}^n \rightarrow \mathbb{R}^n$ la projection orthogonale sur $\text{Im}A$. Enfin, notons $Y = p(B)$ le projeté orthogonal de B sur l'image de A .



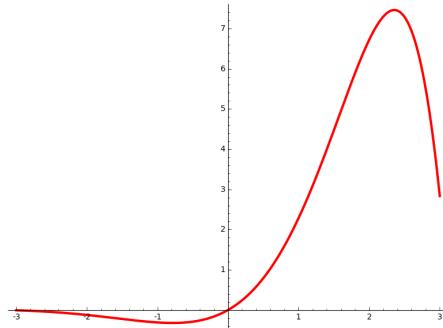
Comme Y est dans l'image de A alors il existe X tel que $AX = Y$ et X est notre «solution» cherchée. En effet, c'est l'une des caractérisations du projeté orthogonal : $Y = p(B)$ est le point de $\text{Im}A$ qui minimise la distance entre B et un point de $\text{Im}A$.

Que Y soit le projeté orthogonal de B sur $\text{Im}A$ signifie que le vecteur $B - Y$ est orthogonal à tout vecteur de $\text{Im}A$:

$$\begin{aligned}
 & \langle AV | B - Y \rangle = 0 \quad \forall V \in \mathbb{R}^p \\
 \iff & \langle AV | B - AX \rangle = 0 \quad \forall V \in \mathbb{R}^p \quad \text{pour } Y = AX \\
 \iff & (AV)^T \cdot (B - AX) = 0 \quad \forall V \in \mathbb{R}^p \quad \text{car } \langle u | v \rangle = u^T \cdot v \\
 \iff & V^T A^T (B - AX) = 0 \quad \forall V \in \mathbb{R}^p \\
 \iff & A^T (B - AX) = 0 \\
 \iff & A^T AX = A^T B \\
 \iff & X = (A^T A)^{-1} A^T B
 \end{aligned}$$

6. Courbes et surfaces

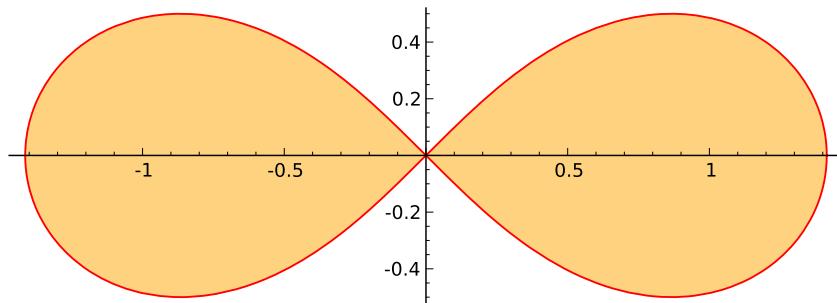
La visualisation est une étape importante dans l'élaboration des preuves en mathématiques. L'avènement des logiciels de calcul formel possédant une interface graphique évoluée a rendu cette phase attrayante. Nous allons explorer les possibilités graphiques offertes par Sage. Nous avons déjà vu comment tracer les graphes de fonctions avec la commande `plot` (voir « Premiers pas avec Sage »), par exemple `plot(sin(x)*exp(x), (x, -3, 3))` trace le graphe de la fonction $f(x) = \sin(x)\exp(x)$ sur l'intervalle $[-3, 3]$.



En plus des graphes de fonctions Sage sait tracer des courbes et des surfaces par d'autres méthodes.

6.1. Courbes paramétrées

La commande `parametric_plot((f(t), g(t)), (t, a, b))` permet de tracer la courbe paramétrée plane donnée par les points de coordonnées $(f(t), g(t))$ lorsque le paramètre t varie dans l'intervalle $[a, b]$. Commençons par la lemniscate de Bernoulli :



Code 102 (*lemniscate-bernoulli.sage*).

```
var('t')
x = sqrt(2)*cos(t)/(1+sin(t)^2)
y = sqrt(2)*cos(t)*sin(t)/(1+sin(t)^2)
G = parametric_plot((x, y), (t, 0, 2*pi))
G.show()
```

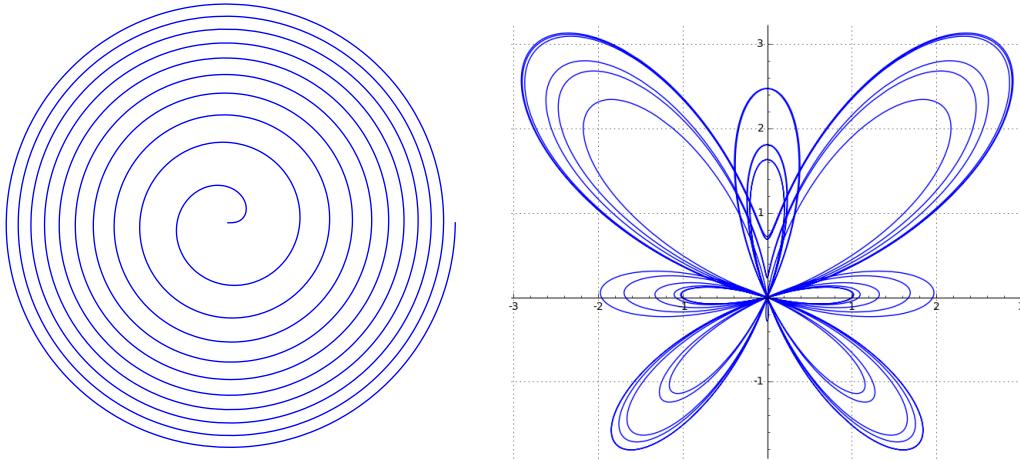
Travaux pratiques 21.

1. Tracer la spirale de Fermat d'équation

$$\begin{cases} x(t) = \sqrt{t} \cos t \\ y(t) = \sqrt{t} \sin t \end{cases} \quad t \in \mathbb{R}_+$$

2. Tracer la courbe du papillon

$$\begin{cases} x(t) = \sin(t) \left(\exp(\cos(t)) - 2 \cos(4t) - \sin^5\left(\frac{t}{12}\right) \right) \\ y(t) = \cos(t) \left(\exp(\cos(t)) - 2 \cos(4t) - \sin^5\left(\frac{t}{12}\right) \right) \end{cases} \quad t \in \mathbb{R}.$$

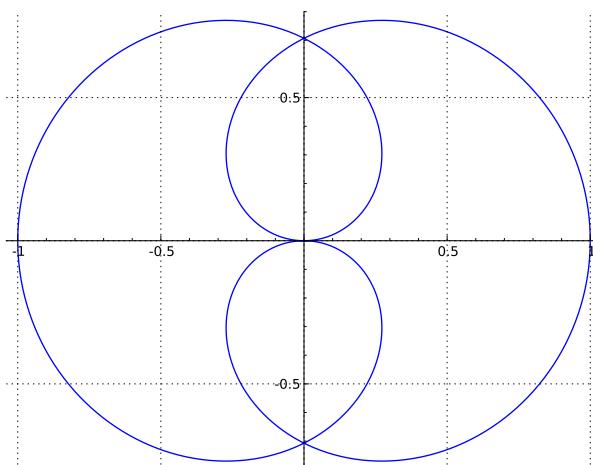


Les commandes de tracé possèdent de nombreuses options, qu'on pourra découvrir grâce à la commande `help(parametric_plot)`. Par exemple :

- Imposer un repère orthonormé : `aspect_ratio = 1`
- Nombre de points pour le tracé : `plot_points = 500`
- Ne pas afficher les axes : `axes = False`
- Afficher une grille : `gridlines = True`
- Remplir l'intérieur : `fill = True`
- Couleur du trait : `color = 'red'`, couleur du remplissage : `fillcolor = 'orange'`

6.2. Courbes en coordonnées polaires

Les courbes en coordonnées polaires sont tracées grâce à la commande `polar_plot(r(t), (t, a, b))`, qui produira l'ensemble des points de coordonnées polaires $[r(t) : t]$ pour t variant dans l'intervalle $[a, b]$. Voici le folium de Dürer d'équation $r(t) = \sin \frac{t}{2}$, $t \in \mathbb{R}$.



Code 103 (*durer-folium.sage*).

```
var('t')
G = polar_plot(sin(t/2),(t,0,4*pi))
G.show()
```

Travaux pratiques 22.

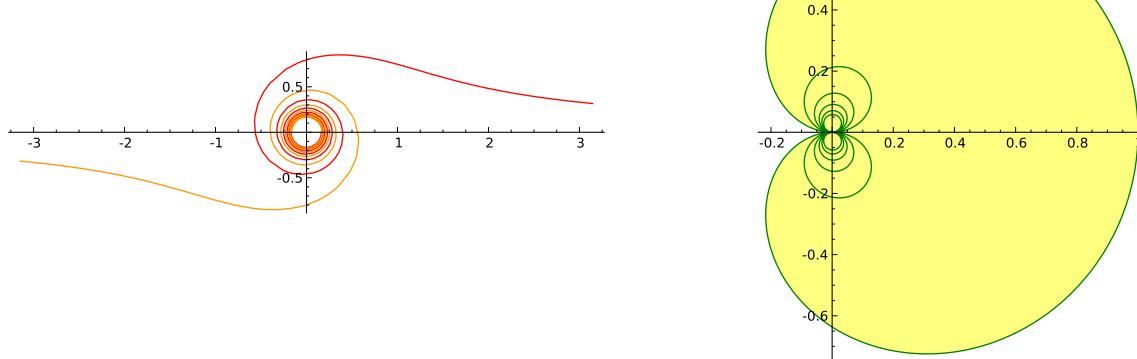
1. Tracer la courbe du Lituus d'équation polaire

$$r(t)^2 = \frac{1}{t} \quad t \in \mathbb{R}^*.$$

2. Tracer la cochléoïde d'équation polaire

$$r(t) = \frac{\sin t}{t} \quad t \in \mathbb{R}^*.$$

Indications : on peut définir deux graphes G1, G2 pour chacun des intervalles de définition. On superpose les graphes avec $G = G1 + G2$, puis on les affiche avec `G.show()`.



6.3. Courbes définies par une équation

Une courbe peut avoir plusieurs types de représentations, comme par exemple un cercle d'équation paramétrique $(\cos t, \sin t)$ ou d'équation implicite $x^2 + y^2 = 1$. La commande

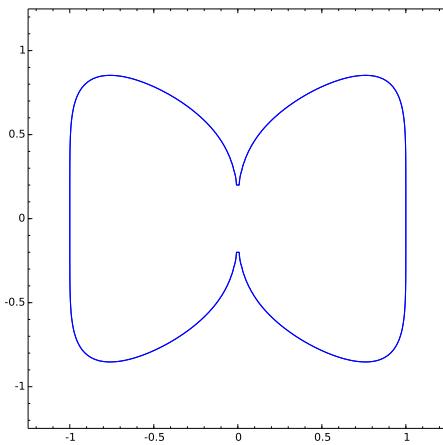
```
implicit_plot(f(x, y), (x, a, b), (y, c, d))
```

permet de tracer l'ensemble des couples (x, y) dans $[a, b] \times [c, d]$ qui sont solutions de l'équation $f(x, y) = 0$. Voici une autre courbe de papillon, cette fois algébrique.

Code 104 (*butterfly-curve-algebraic.sage*).

```
f(x,y) = x^6 + y^6 - x^2
G = implicit_plot(f, (x, -1.2, 1.2), (y, -1.2, 1.2))
G.show()
G.save('butterfly_curve_algebraic.png')
```

Noter comment il est possible de sauvegarder une image du tracé dans un fichier externe.

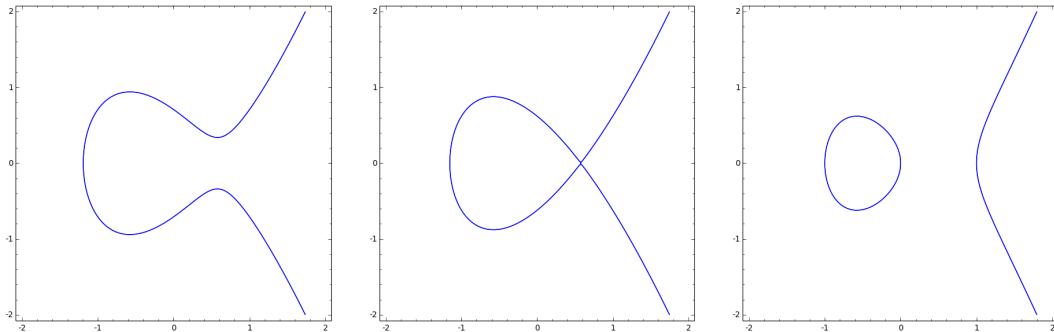
**Travaux pratiques 23.**

1. Définir une fonction qui renvoie la courbe définie par

$$y^2 - x^3 + x + c = 0$$

en fonction du paramètre $c \in \mathbb{R}$.

2. À l'aide de la commande `animate`, réaliser une animation qui affiche l'évolution de la courbe pour $c \in [-1, 1]$.

**6.4. Courbes de l'espace**

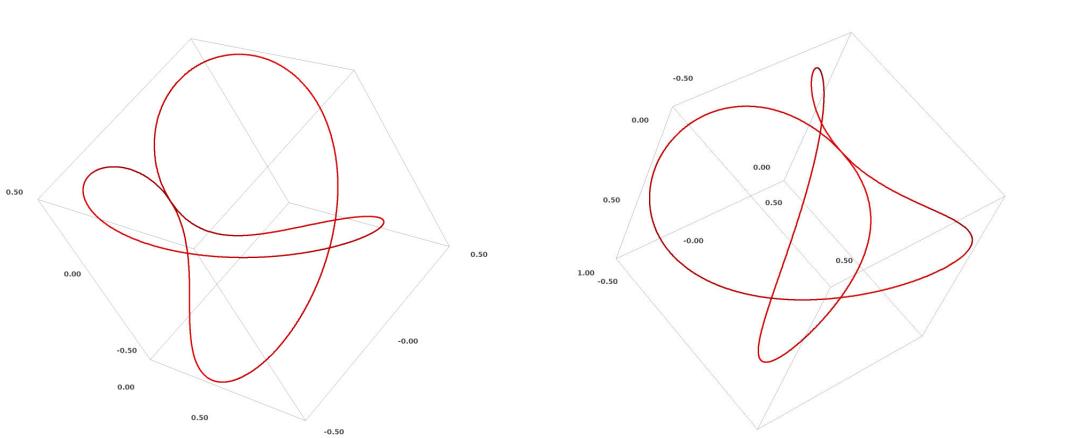
La commande `parametric_plot((x, y, z), (t, a, b))`, (ou bien `parametric_plot3d()`) analogue à celle de la dimension 2, trace la courbe paramétrée de l'espace donnée par les points de coordonnées $(x(t), y(t), z(t))$ lorsque le paramètre t varie dans l'intervalle $[a, b]$.

Travaux pratiques 24.

Tracer la courbe d'Archytas d'équation paramétrique :

$$\begin{cases} x(t) = \cos^2 t \\ y(t) = \cos t \sin t \\ z(t) = \pm \sqrt{\cos t(1 - \cos t)} \end{cases} \quad t \in \left[-\frac{\pi}{2}, +\frac{\pi}{2} \right].$$

Vous obtiendrez une figure qu'il est possible d'orienter dynamiquement avec la souris. Voici quelques-unes de ces vues.



6.5. Surfaces

Découvrez, à l'aide de l'énoncé suivant, les différentes méthodes pour tracer des surfaces avec Sage.

Travaux pratiques 25.

1. Tracer le graphe de la fonction $f(x, y) = x^2y^2 - (x^2 + y^2)^3$ définie sur $(x, y) \in [-1, 1] \times [-1, 1]$. Utiliser la fonction `plot3d`.
2. Tracer la surface d'Enneper définie par l'équation

$$\left(\frac{y^2 - x^2}{2z} + \frac{2}{9}z^2 + \frac{2}{3} \right)^3 - 6 \left(\frac{y^2 - x^2}{4z} - \frac{1}{4}(x^2 + y^2 + \frac{8}{9}z^2) + \frac{2}{9} \right)^2 = 0.$$

Utiliser la fonction `implicit_plot3d`.

3. Tracer la nappe paramétrée définie par :

$$\begin{cases} x(s, t) = t^2 \cos s \\ y(s, t) = s^2 \sin t \\ z(s, t) = \cos t + \sin t \end{cases} \quad s \in [0, 1], \quad t \in [-\pi, +\pi].$$

Utiliser la fonction `parametric_plot3d`.

4. Tracer la surface paramétrée définie en coordonnées cylindriques par :

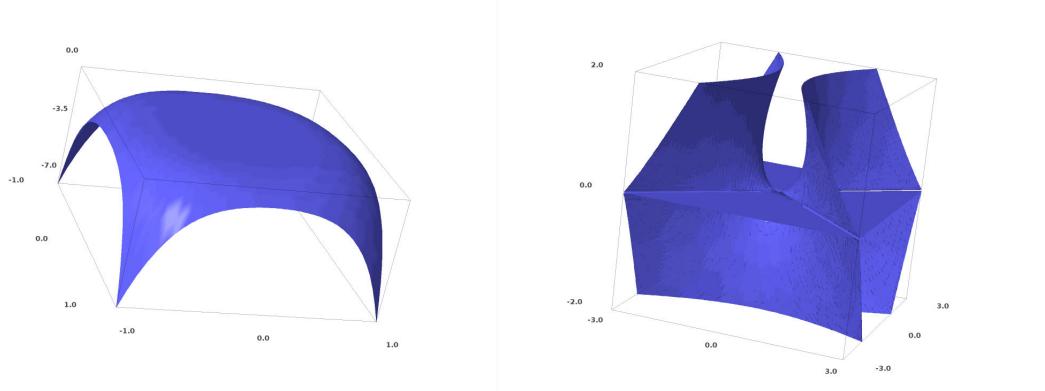
$$r(\theta, z) = z^3 \cos \theta \quad \theta \in [0, 2\pi], \quad z \in [0, 2].$$

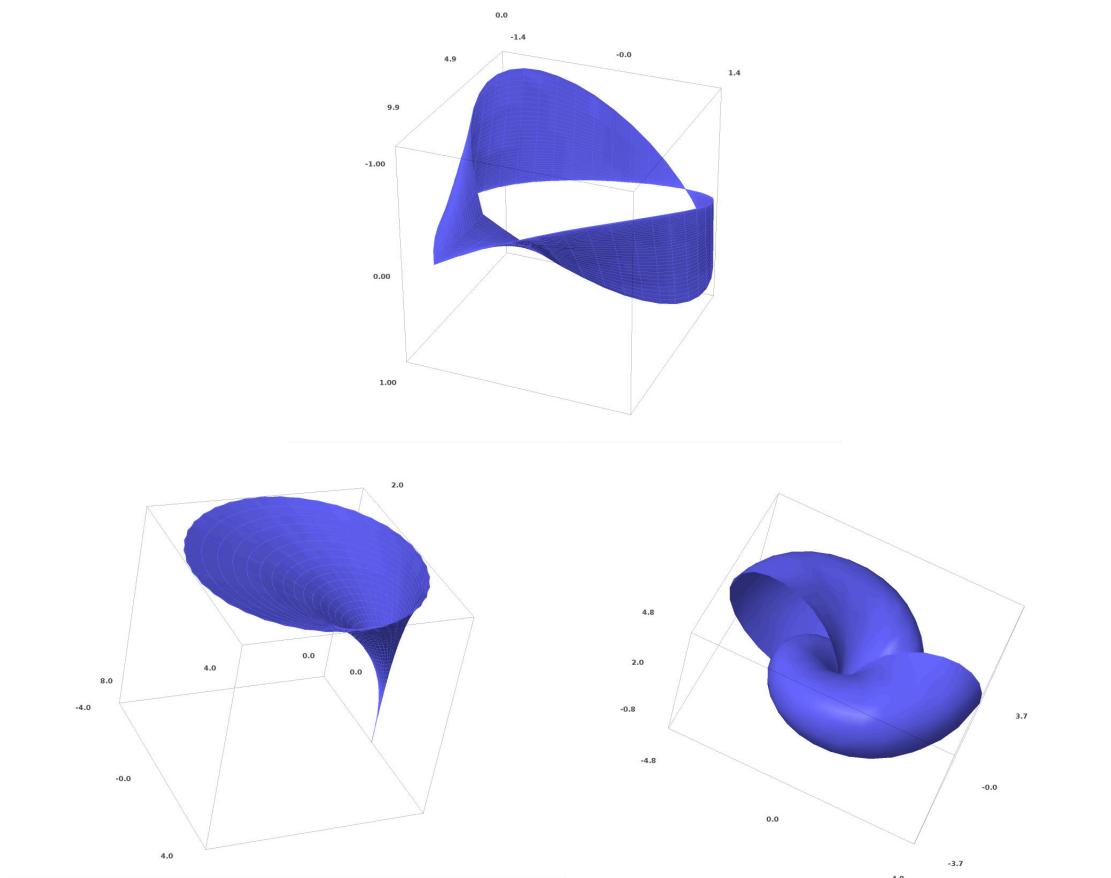
Utiliser la fonction `cylindrical_plot3d`.

5. Tracer la surface paramétrée définie en coordonnées sphériques par

$$r(\theta, \phi) = \theta \sin(2\phi) \quad \theta \in [-1, 2\pi], \quad \phi \in [0, \pi].$$

Utiliser la fonction `spherical_plot3d`.





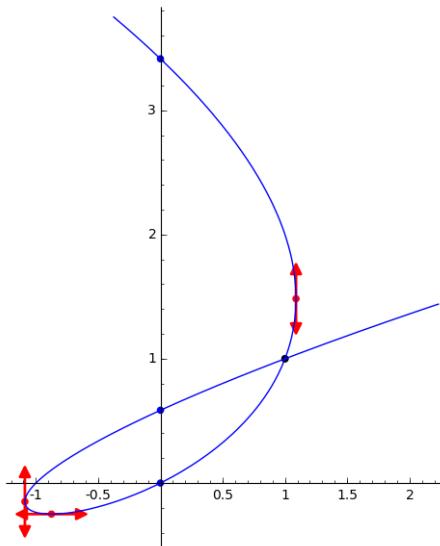
6.6. Étude d'une courbe paramétrée

Travaux pratiques 26.

Étudier en détail la courbe paramétrée du plan définie par

$$\begin{cases} x(t) = t^3 - 2t \\ y(t) = t^2 - t \end{cases} \quad t \in \mathbb{R}.$$

1. Tracer la courbe.
2. Trouver les points d'intersection de la courbe avec l'axe des ordonnées.
3. Trouver les points en lesquels la tangente est verticale, puis horizontale.
4. Trouver les coordonnées des points doubles.



1. La courbe a l'allure d'une boucle.
2. On définit $x = t^3 - 2t$ et $y = t^2 - t$. On obtient les valeurs de t correspondant aux points d'intersection de la courbe avec l'axe ($x = 0$) en résolvant l'équation $x(t) = 0$: `solve(x==0, t)`. On obtient trois solutions $t \in \{-\sqrt{2}, 0, +\sqrt{2}\}$ correspondant aux trois points $\{(0, 2 + \sqrt{2}), (0, 0), (0, 2 - \sqrt{2})\}$
3. On définit les fonctions dérivées $x'(t)$ et $y'(t)$ par `xx = diff(x, t)` et `yy = diff(y, t)`, les valeurs de t pour lesquelles la tangente à la courbe est verticale s'obtiennent en résolvant l'équation $x'(t) = 0$, ce qui s'écrit `solve(xx==0, t)`.
4. Trouver les points doubles est souvent un calcul délicat (même pour un logiciel de calcul formel). Il s'agit ici de résoudre le système d'équations polynomiales :

$$\begin{cases} x(s) = x(t) \\ y(s) = y(t) \end{cases} \quad s, t \in \mathbb{R}.$$

Mais il faut exclure la solution évidente $t = s$. Algébriquement cela signifie que $(s-t)$ divise les polynômes de deux variables $x(s)-x(t)$ et $y(s)-y(t)$. Autrement dit, il s'agit de résoudre :

$$\begin{cases} \frac{x(s)-x(t)}{s-t} = 0 \\ \frac{y(s)-y(t)}{s-t} = 0 \end{cases} \quad s, t \in \mathbb{R}.$$

Le code suivant fournit la solution $(s, t) = \left(\frac{1-\sqrt{5}}{2}, \frac{1+\sqrt{5}}{2}\right)$, il y a un unique point double ayant pour coordonnées $(1, 1)$.

Code 105 (*courbe.sage*).

```
var('s,t')
x = t^3-2*t
y = t^2-t
eqx = x.subs(t=s) - x # Equation x(s)-x(t)
eqxs = (eqx/(s-t)).simplify_rational() # (x(s)-x(t))/(s-t)
eqy = y.subs(t=s) - y # Equation y(s)-y(t)
eqys = (eqy/(s-t)).simplify_rational() # (y(s)-y(t))/(s-t)
points_double = solve([eqxs==0,eqys==0], s,t) # Solutions
```

6.7. La projection stéréographique

Travaux pratiques 27.

Soit \mathcal{S} la sphère centrée à l'origine de \mathbb{R}^3 et de rayon 1. On note $N = (0, 0, 1)$ le pôle Nord. Soit \mathcal{P} le plan équatorial d'équation $(z = 0)$. La **projection stéréographique** est l'application $\Phi : \mathcal{S} \setminus \{N\} \rightarrow \mathcal{P}$ qui à un point S de la sphère associe le point $P = (NS) \cap \mathcal{P}$ défini par l'intersection de la droite (NS) avec le plan \mathcal{P} .

1. En écrivant la relation $\overrightarrow{NP} = k\overrightarrow{NS}$ et sachant que $P \in \mathcal{P}$, trouver k et en déduire P .

Vérifier ainsi que l'application Φ est définie par :

$$\Phi : \mathcal{S} \setminus \{N\} \rightarrow \mathcal{P} \quad \Phi(x, y, z) = \left(\frac{x}{1-z}, \frac{y}{1-z} \right)$$

Définir la fonction correspondante avec Sage.

2. Définir et vérifier que l'application inverse est :

$$\Psi : \mathcal{P} \rightarrow \mathcal{S} \setminus \{N\} \quad \Psi(X, Y) = \left(\frac{2X}{\rho}, \frac{2Y}{\rho}, 1 - \frac{2}{\rho} \right) \quad \text{avec } \rho = 1 + X^2 + Y^2$$

3. Écrire une fonction qui dessine une courbe paramétrée $\mathcal{C}' : (x(t), y(t))$, $t \in [a, b]$ du plan \mathcal{P} , et qui dessine aussi l'image inverse de la courbe sur la sphère, $\mathcal{C} = \Psi(\mathcal{C}')$.

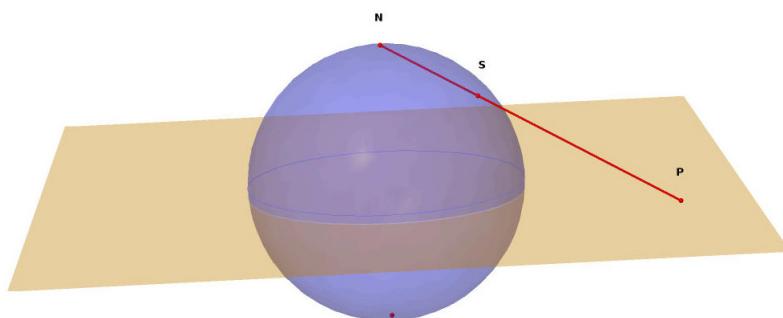
4. Vérifier graphiquement deux propriétés fondamentales de la projection stéréographique :

- « La projection stéréographique envoie les cercles de la sphère sur des cercles ou des droites du plan. »
- « La projection stéréographique préserve les angles. » En particulier, deux courbes qui se coupent à angle droit, s'envoient sur deux courbes qui se coupent à angle droit.

5. Soit \mathcal{C}' la spirale logarithmique d'équation $(e^t \cos t, e^t \sin t)$, $t \in \mathbb{R}$. Tracer la loxodromie de la sphère qui est $\mathcal{C} = \Psi(\mathcal{C}')$.

6. Soit la courbe $\mathcal{C} \subset \mathcal{S}$ définie par $\frac{1}{13t^2-6t+2}(4t, -6t+2, 13t^2-6t)$. Montrer que son image $\mathcal{C}' = \Phi(\mathcal{C}) \subset \mathcal{P}$ est une droite, dont vous déterminerez une équation.

1.



Le vecteur \overrightarrow{NP} est colinéaire au vecteur \overrightarrow{NS} , donc il existe $k \in \mathbb{R}$ tel que $\overrightarrow{NP} = k\overrightarrow{NS}$. Mais en plus on veut que $P \in \mathcal{P}$, c'est-à-dire $z_P = 0$. Ce qui permet de trouver k et d'en déduire $P = N + k\overrightarrow{NS}$. On laisse Sage faire les calculs :

Code 106 (*stereographic.sage (1)*).

```
var('x,y,z')          # Variables de l'espace
var('X,Y')            # Variables du plan
var('k')
N = vector([0,0,1])    # Les points N, S, P comme des vecteurs
```

```

S = vector([x,y,z])
P = vector([X,Y,0])
V = (P-N)-k*(S-N)      # Le vecteur NP - k NS
eq = V[2]                # On veut la troisième coordonnée nulle
sol = solve(eq==0,k)     # Equation en k pour cette condition
k = sol[0].rhs()          # Solution k
P = N + k*(S-N)          # Le point P

```

On obtient $k = \frac{1}{1-z}$, et alors si $S = (x, y, z)$, $P = \left(\frac{x}{1-z}, \frac{y}{1-z}, 0\right)$. En identifiant les points du plan à \mathbb{R}^2 , on a ainsi prouvé $\Phi(x, y, z) = \left(\frac{x}{1-z}, \frac{y}{1-z}\right)$. D'où la définition de la fonction Φ :

Code 107 (*stereographic.sage (2)*).

```

def stereo(x,y,z):
    X = x/(1-z)
    Y = y/(1-z)
    return X,Y

```

2. Voici la fonction Ψ :

Code 108 (*stereographic.sage (3)*).

```

def inverse_stereo(X,Y):
    r = 1+X^2+Y^2
    x = 2*X/r
    y = 2*Y/r
    z = 1-2/r
    return x,y,z

```

On profite du fait que l'on nous donne le candidat, pour se contenter de vérifier que c'est bien la bijection réciproque, c'est-à-dire $\Phi(\Psi(X, Y)) = (X, Y)$ pour tout $(X, Y) \in \mathbb{R}^2$. La composition s'écrit

Code 109 (*stereographic.sage (4)*).

```

newx,newy,newz = inverse_stereo(X,Y)
newX,newY = stereo(newx,newy,newz)

```

et comme on obtient $\text{newX} = X$ et $\text{newY} = Y$, cela prouve le résultat.

Il est possible de composer les fonctions de plusieurs variables, mais il faut rajouter une « * » devant la fonction à composer :

```
stereo(*inverse_stereo(X,Y))
```

cela renvoie X, Y ce qui prouve bien $\Phi(\Psi(X, Y)) = (X, Y)$. (L'opérateur « * » devant une liste permet de la décompresser, par exemple $*(2, 5, 7)$ s'interprète comme $2, 5, 7$ qui peut alors être passé en argument à une fonction.)

Pour prouver $\Psi(\Phi(x, y, z)) = (x, y, z)$, il faudrait se souvenir que $(x, y, z) \in \mathcal{S}$, donc $x^2 + y^2 + z^2 = 1$.

3. Voici comment tracer les courbes. La courbe du plan est tracée comme une ligne polygonale, avec un pas assez petit. Ensuite on calcule l'image par Ψ de chacun de ces points.

Code 110 (*stereographic.sage (5)*).

```

def courbes(X,Y,a,b):

```

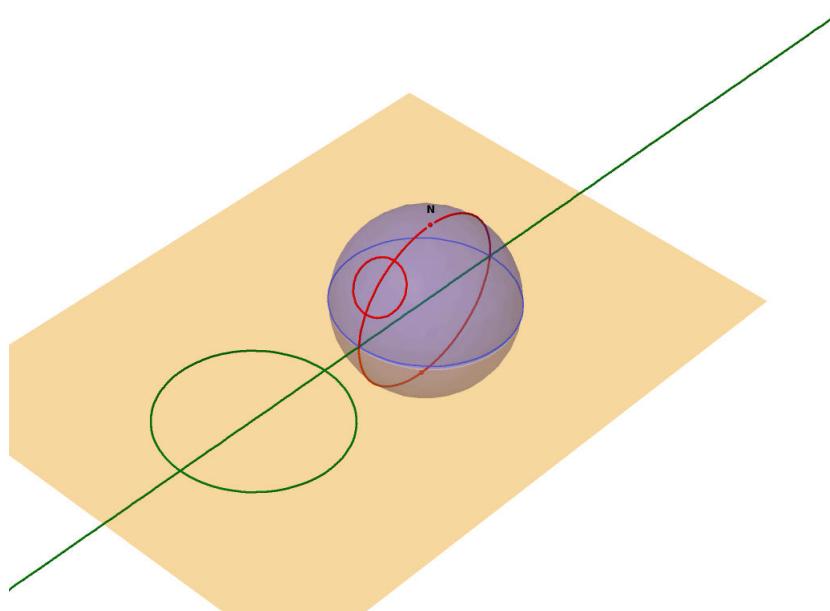
```

XYtab = [ [X.subs(t=myt),Y.subs(t=myt),0] for myt in strange(a,b,0.1) ]
xyztab = [ inverse_stereo(coord[0],coord[1]) for coord in XYtab ]
G = sphere((0,0,0),1)
G = G + line(XYtab)
G = G + line(xyztab)
return G

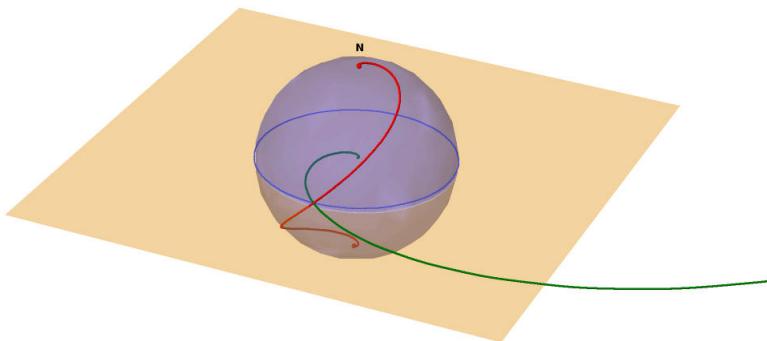
```

Par exemple $G = \text{courbes}(t^3, t^2, -2, 2)$, trace la courbe du plan d'équation paramétrique (t^3, t^2) , $t \in [-2, 2]$, ainsi que son image par la projection stéréographique inverse.

Voici le cas d'un cercle et d'une droite du plan qui se coupent à angle droit, de même que les cercles de la sphères.



4. Voici une loxodromie de la sphère, courbe utilisée par les navigateurs, car elle correspond à une navigation à cap constant.



5. Pour x, y, z définis par la paramétrisation, on calcule l'image par la projection stéréographique avec

`X,Y = stereo(x,y,z)`, on n'oublie pas de simplifier à l'aide de `full_simplify()`. Cela donne une paramétrisation de l'image, $x(t) = 2t$, $y(t) = -3t + 1$, qui est bien l'équation d'une droite.

7. Calculs d'intégrales

7.1. Sage comme une super-calculatrice

Travaux pratiques 28.

Calculer à la main les primitives des fonctions suivantes. Vérifier vos résultats à l'ordinateur.

1. $f_1(x) = x^4 + \frac{1}{x^2} + \exp(x) + \cos(x)$

2. $f_2(x) = x \sin(x^2)$

3. $f_3(x) = \frac{\alpha}{\sqrt{1-x^2}} + \frac{\beta}{1+x^2} + \frac{\gamma}{1+x}$

4. $f_4(x) = \frac{x^4}{x^2-1}$

5. $f_5(x) = x^n \ln x$ pour tout $n \geq 0$

Pour une fonction f , par exemple `f(x) = x*sin(x^2)` donnée, la commande `integral(f(x),x)` renvoie une primitive de f . Le résultat obtenu ici est $-1/2*\cos(x^2)$.

Quelques remarques :

- La machine renvoie une primitive. Pour avoir l'ensemble des primitives, il faut bien entendu ajouter une constante.
- La fonction `log` est la fonction logarithme népérien usuel, habituellement notée `ln`.
- `integral(1/x,x)` renvoie `log(x)` alors que $\int \frac{dx}{x} = \ln|x|$. Sage omet les valeurs absolues, ce qui ne l'empêche pas de faire des calculs exacts même pour les valeurs négatives. (En fait Sage travaille avec le logarithme complexe.).
- Souvent, pour intégrer des fractions rationnelles, on commence par écrire la décomposition en éléments simples. C'est possible avec la commande `partial_fraction`. Par exemple pour $f = x^4/(x^2-1)$ la commande `f.partial_fraction(x)` renvoie la décomposition sur \mathbb{Q} : $f(x) = x^2 + 1 - \frac{1}{2} \frac{1}{x+1} + \frac{1}{2} \frac{1}{x-1}$.

Travaux pratiques 29.

Calculer à la main et à l'ordinateur les intégrales suivantes.

1. $I_1 = \int_1^3 x^2 + \sqrt{x} + \frac{1}{x} + \sqrt[3]{x} dx$

2. $I_2 = \int_0^{\frac{\pi}{2}} \sin x (1 + \cos x)^4 dx$

3. $I_3 = \int_0^{\frac{\pi}{6}} \sin^3 x dx$

4. $I_4 = \int_0^{\sqrt{3}} \frac{3x^3 - 2x^2 - 5x - 6}{(x+1)^2(x^2+1)} dx$

Par exemple `integral(sin(x)^3,x,0,pi/6)` renvoie $\frac{2}{3} - \frac{3\sqrt{3}}{8}$.

7.2. Intégration par parties

Travaux pratiques 30.

Pour chaque $n \geq 0$, nous allons déterminer une primitive de la fonction

$$f_n(x) = x^n \exp(x).$$

1. Sage connaît-il directement la formule ?
 2. Calculer une primitive F_n de f_n pour les premières valeurs de n .
 3. (a) Émettre une conjecture reliant F_n et F_{n-1} .
 - (b) Prouver cette conjecture.
 - (c) En déduire une fonction récursive qui calcule F_n .
 4. (a) Émettre une conjecture pour une formule directe de F_n .
 - (b) Prouver cette conjecture.
 - (c) En déduire une expression qui calcule F_n .
1. La commande `integral(x^n*exp(x),x)` renvoie le résultat $(-1)^n \gamma(n+1, -x)$. Nous ne sommes pas tellement avancés ! Sage renvoie une *fonction spéciale* Γ que l'on ne connaît pas. En plus (à l'heure actuelle, pour la version 6.6), dériver la primitive avec Sage ne redonne pas la même expression que f_n .
2. Par contre, il n'y a aucun problème pour calculer les primitives F_n pour les premières valeurs de n . On obtient :
- $$\begin{aligned} F_0(x) &= \exp(x) \\ F_1(x) &= (x-1)\exp(x) \\ F_2(x) &= (x^2-2x+2)\exp(x) \\ F_3(x) &= (x^3-3x^2+6x-6)\exp(x) \\ F_4(x) &= (x^4-4x^3+12x^2-24x+24)\exp(x) \\ F_5(x) &= (x^5-5x^4+20x^3-60x^2+120x-120)\exp(x) \end{aligned}$$

3. (a) Au vu des premiers termes, on conjecture $F_n(x) = x^n \exp(x) - nF_{n-1}(x)$. On vérifie facilement sur les premiers termes que cette conjecture est vraie.

Sage ne sait pas (encore) vérifier formellement cette identité (avec n une variable formelle). La commande suivante échoue à trouver 0 :

```
integral(x^n*exp(x),x) - x^n*exp(x) + n*integral(x^(n-1)*exp(x),x)
```

- (b) Nous prouverons la validité de cette formule en faisant une intégration par parties (on pose par exemple $u = x^n$, $v' = \exp(x)$ et donc $u' = nx^{n-1}$, $v = \exp(x)$) :

$$F_n(x) = \int x^n \exp(x) dx = [x^n \exp(x)] - \int nx^{n-1} \exp(x) dx = x^n \exp(x) - nF_{n-1}(x).$$

- (c) Voici l'algorithme récursif pour le calcul de F_n utilisant la relation de récurrence.

Code 111 (*integrales-ipp (1)*).

```
def FF(n):
    if n==0:
        return exp(x)
    else:
        return x^n*exp(x) - n*FF(n-1)
```

4. Avec un peu de réflexion, on conjecture la formule

$$F_n(x) = \exp x \sum_{k=0}^n (-1)^{n-k} \frac{n!}{k!} x^k$$

où $\frac{n!}{k!} = n(n-1)(n-2)\cdots(k+1)$.

Pour la preuve, on pose $G_n(x) = \exp(x) \sum_{k=0}^n (-1)^{n-k} \frac{n!}{k!} x^k$. On a

$$G_n(x) = \exp(x) \sum_{k=0}^{n-1} (-1)^{n-k} \frac{n!}{k!} x^k + \exp(x)x^n = -nG_{n-1}(x) + \exp(x)x^n.$$

Ainsi G_n vérifie la même relation de récurrence que F_n . De plus $G_0(x) = \exp x = F_0(x)$. Donc pour tout n , $G_n = F_n$.

On peut donc ainsi calculer directement notre intégrale par la formule :

```
exp(x)*sum((-1)^(n-k)*x^k*factorial(n)/factorial(k), k, 0, n)
```

7.3. Changement de variable

Soit f une fonction définie sur un intervalle I et $\varphi : J \rightarrow I$ une bijection de classe \mathcal{C}^1 . La formule de changement de variable pour les primitives est :

$$\int f(x) dx = \int f(\varphi(u)) \cdot \varphi'(u) du.$$

La formule de changement de variable pour les intégrales, pour tout $a, b \in I$, est :

$$\int_a^b f(x) dx = \int_{\varphi^{-1}(a)}^{\varphi^{-1}(b)} f(\varphi(u)) \cdot \varphi'(u) du.$$

Travaux pratiques 31.

1. Calcul de la primitive $\int \sqrt{1-x^2} dx$.

- (a) Poser $f(x) = \sqrt{1-x^2}$ et le changement de variable $x = \sin u$, c'est-à-dire on pose $\varphi(u) = \sin u$.
- (b) Calculer $f(\varphi(u))$ et $\varphi'(u)$.
- (c) Calculer la primitive de $g(u) = f(\varphi(u)) \cdot \varphi'(u)$.
- (d) En déduire une primitive de $f(x) = \sqrt{1-x^2}$.

2. Calcul de la primitive $\int \frac{dx}{1 + (\frac{x+1}{x})^{1/3}}$.

- (a) Poser le changement de variable défini par l'équation $u^3 = \frac{x+1}{x}$.
- (b) En déduire le changement de variable $\varphi(u)$ et $\varphi^{-1}(x)$.

3. Écrire une fonction `integrale_chgtvar(f, eqn)` qui calcule une primitive de $f(x)$ par le changement de variable défini par l'équation `eqn` reliant u et x .

En déduire $\int (\cos x + 1)^n \cdot \sin x dx$, en posant $u = \cos x + 1$.

4. Même travail avec `integrale_chgtvar_bornes(f, a, b, eqn)` qui calcule l'intégrale de $f(x)$ entre a et b par changement de variable.

En déduire $\int_0^{\frac{\pi}{4}} \frac{dx}{2 + \sin^2 x}$, en posant $u = \tan x$.

1. (a) La fonction est $f(x) = \sqrt{1-x^2}$ et on pose la fonction $\varphi(u) = \sin u$, c'est-à-dire que l'on espère que le changement de variable $x = \sin u$ va simplifier le calcul de l'intégrale :

```
f = sqrt(1-x^2)      phi = sin(u)
```

- (b) On obtient $f(\varphi(u))$ en substituant la variable x par $\sin u$: cela donne $f(\varphi(u)) = \sqrt{1 - \sin^2 u} = \sqrt{\cos^2 u} = |\cos u|$. Ce qui s'obtient par la commande `frondphi = f(x=phi)` (puis en simplifiant). (Notez que Sage « oublie » les valeurs absolues, car il calcule en fait la racine carrée complexe et pas réelle. Ce sera heureusement sans conséquence pour la suite. En effet pour que $f(x)$ soit définie, il faut $x \in [-1, 1]$, comme $x = \sin u$ alors $u \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ et donc $\cos u \geq 0$.) Enfin $\varphi'(u) = \cos u$ s'obtient par `dphi = diff(phi,u)`.
- (c) On pose $g(u) = f(\varphi(u)) \cdot \varphi'(u) = \cos^2 u = \frac{1+\cos(2u)}{2}$. Donc $\int g(u) du = \frac{u}{2} + \frac{\sin(2u)}{4}$.
- (d) Pour obtenir une primitive de f , on utilise la formule de changement de variable :

$$\int \sqrt{1-x^2} dx = \int f(x) dx = \int g(u) du = \frac{u}{2} + \frac{\sin(2u)}{4}.$$

Mais il ne faut pas oublier de revenir à la variable x , Comme $x = \sin u$ alors $u = \arcsin x$, donc

$$\int \sqrt{1-x^2} dx = \frac{\arcsin x}{2} + \frac{\sin(2\arcsin x)}{4}.$$

Les commandes sont donc `G = integral(g,u)` puis `F = G(u = arcsin(x))` donne la primitive recherchée. Après simplification de $\sin(2\arcsin x)$, on obtient :

$$\int \sqrt{1-x^2} dx = \frac{\arcsin x}{2} + \frac{x}{2} \sqrt{1-x^2}.$$

2. Pour calculer la primitive $\int \frac{dx}{1+(\frac{x+1}{x})^{1/3}}$, l'énoncé nous recommande le changement de variable $u^3 = \frac{x+1}{x}$. La seule difficulté supplémentaire est qu'il faut exprimer x en fonction de u et aussi u en fonction de x . Pour cela on définit l'équation `u^3 == (x+1)/x` reliant u et x : `eqn = u^3 == (x+1)/x`. On peut maintenant résoudre l'équation afin d'obtenir $\phi(u)$ (x en fonction de u) : `phi = solve(eqn,x)[0].rhs()` et $\phi^{-1}(x)$ (u en fonction de x) : `phi_inv = solve(eqn,u)[0].rhs()`. Le reste se déroule comme précédemment.

3. On automatise la méthode précédente :

Code 112 (*integrales-chgtvar (1)*).

```
def integrale_chgtvar(f,eqn):
    phi = solve(eqn,x)[0].rhs()          # x en fonction de u : fonction phi(u)=x
    phi_inv = solve(eqn,u)[0].rhs()        # u en fonction de x : inverse de phi : phi_inv(x)
    frondphi = f(x=phi)                  # la fonction f(phi(u))
    dphi = diff(phi,u)                  # sa dérivée
    g = frondphi*dphi                  # la nouvelle fonction g(u)
    g = g.simplify_full()
    G = integral(g,u)                  # g doit être plus facile à intégrer
    F = G(u = phi_inv)                 # on revient à la variable x
    F = F.simplify_full()
    return F
```

Ce qui s'utilise ainsi :

Code 113 (*integrales-chgtvar (2)*).

```
f = (cos(x)+1)^n*sin(x)
eqn = u == cos(x)+1
integrale_chgtvar(f,eqn)
```

Et montre que

$$\int (\cos x + 1)^n \cdot \sin x \, dx = -\frac{1}{n+1}(\cos x + 1)^{n+1}.$$

4. Pour la formule de changement de variable des intégrales, on adapte la procédure précédente en calculant l'intégrale $\int_{\varphi^{-1}(a)}^{\varphi^{-1}(b)} g(u) \, du$.

Pour cela on calcule $\varphi^{-1}(a)$ (par `a_inv = phi_inv(x=a)`) et $\varphi^{-1}(b)$ (par `b_inv = phi_inv(x=b)`).

Pour calculer $\int_0^{\frac{\pi}{4}} \frac{dx}{2+\sin^2 x}$, on pose $u = \tan x$. Donc $x = \phi(u) = \arctan u$ et $u = \varphi^{-1}(x) = \tan x$. Nous avons $\phi'(u) = \frac{1}{1+u^2}$. D'autre part, comme $a = 0$ alors $\varphi^{-1}(a) = \tan 0 = 0$ et comme $b = \frac{\pi}{4}$ alors $\varphi^{-1}(b) = \tan \frac{\pi}{4} = 1$. Ainsi la formule de changement de variable :

$$\int_a^b f(x) \, dx = \int_{\varphi^{-1}(a)}^{\varphi^{-1}(b)} f(\varphi(u)) \cdot \varphi'(u) \, du$$

devient

$$I = \int_0^{\frac{\pi}{4}} \frac{1}{2+\sin^2 x} \, dx = \int_0^1 \frac{1}{2+\sin^2(\arctan u)} \frac{1}{1+u^2} \, du.$$

Mais $\sin^2(\arctan u) = \frac{u^2}{1+u^2}$ donc

$$I = \int_0^1 \frac{1}{2+\frac{u^2}{1+u^2}} \frac{1}{1+u^2} \, du = \int_0^1 \frac{du}{2+3u^2} \, du = \left[\frac{1}{\sqrt{6}} \arctan\left(\frac{\sqrt{6}}{2}\right) \right]_0^1 = \frac{1}{\sqrt{6}} \arctan\left(\frac{\sqrt{6}}{2}\right).$$

Ce que l'ordinateur confirme !

8. Polynômes

8.1. Manipuler les polynômes

Travaux pratiques 32.

Soient $P(X) = X^4 - 3X^2 - 1$, $Q(X) = (X + 1)^4$ des polynômes de $\mathbb{Q}[X]$. Après avoir déclaré l'anneau des polynômes $\mathbb{Q}[X]$ par `R.<X> = QQ[]` (où `QQ` désigne le corps des rationnels), répondre aux questions suivantes :

1. Est-ce que $\deg(P \cdot Q) = \deg P + \deg Q$?
 2. Est-ce que $\deg(P - Q) = \max(\deg P, \deg Q)$?
 3. Développer Q . Quel est le coefficient devant X^3 ?
 4. Quel est le quotient de la division euclidienne de P par $(X + 1)^2$? Et le reste ?
 5. Quelles sont les racines de Q ? Et celles de P ?
-
1. (et 2.) La commande `degree()` permet d'obtenir le degré ; ainsi, après avoir déclaré l'anneau de polynômes $\mathbb{Q}[X]$ et avoir défini les deux polynômes P et Q , on teste les égalités sur les degrés.

Code 114 (`intro-polynome.sage (1)`).

```
R.<X> = QQ[]
P = X^4 - 3*X^2 - 1
Q = (X+1)^4
(P*Q).degree() == P.degree() + Q.degree()
(P-Q).degree() == max(P.degree(), Q.degree())
```

La première égalité est vraie, par contre la seconde est fausse (il n'y a pas toujours égalité des degrés). Les énoncés vrais en toute généralité pour des polynômes non nuls sont :

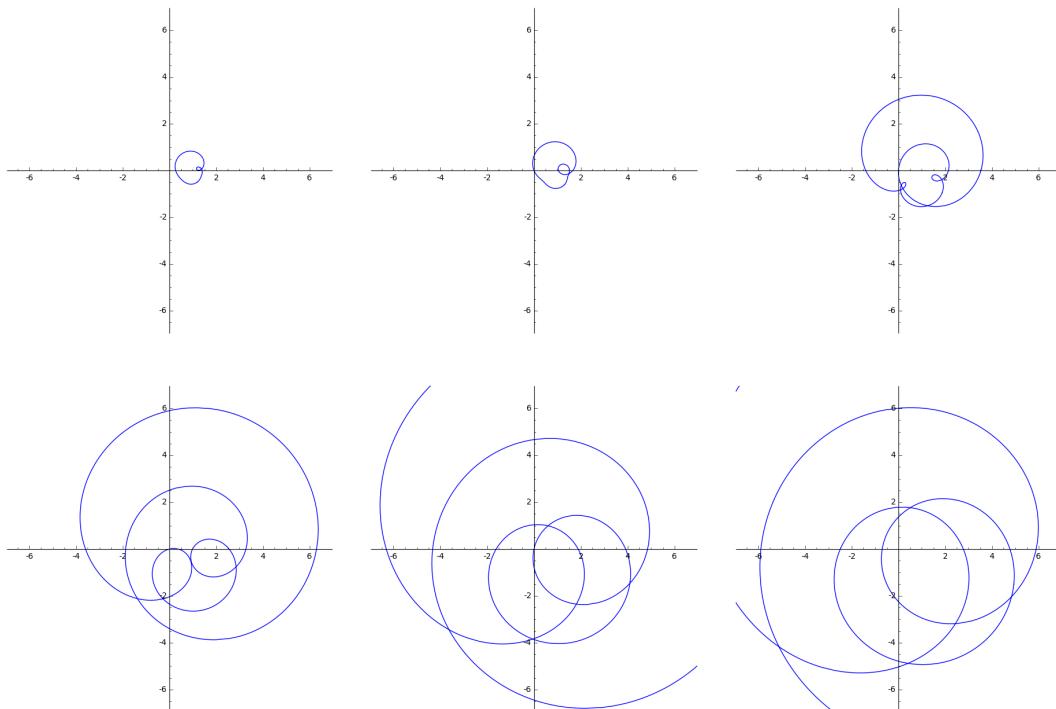
$$\deg(P \cdot Q) = \deg P + \deg Q \quad \text{et} \quad \deg(P + Q) \leq \max(\deg P, \deg Q)$$

3. On développe un polynôme par `expand(Q)`. On récupère les coefficients comme si le polynôme était une liste : `Q[k]` renvoie le coefficient a_k devant X^k .
4. On obtient le quotient et le reste comme avec les entiers. Le quotient `P // (X+1)^2` vaut $X^2 - 2X$. Le reste `P % (X+1)^2` vaut $2X - 1$.
5. La question est ambiguë ! Il faut préciser dans quel ensemble on cherche les racines. Est-ce dans \mathbb{Q} , \mathbb{R} ou \mathbb{C} ? Souhaitons-nous une racine exacte ou approchée ?
 - `P.roots()` : renvoie les racines du corps de base (ici \mathbb{Q}). Renvoie ici une liste vide car P n'a pas de racines rationnelles.
 - `Q.roots()` renvoie $[-1, 4]$ car -1 est une racine de multiplicité 4.
 - `P.roots(QQbar)` : racines exactes dans \mathbb{C} (pour un polynôme à coefficients dans \mathbb{Q}). Ici renvoie deux racines réelles et deux racines imaginaires pures : $-1.817354021023971?, 1.817354021023971?, -0.5502505227003375?*I, 0.5502505227003375?*I$. Le point d'interrogation en fin d'écriture signifie que Sage calcule avec les valeurs exactes, mais n'affiche que les premières décimales.
 - `P.roots(RR)` : racines réelles *approchées* : $-1.81735402102397, 1.81735402102397$. On quitte donc la résolution formelle pour une résolution numérique approchée.
 - `P.roots(CC)` : racines complexes *approchées*.

Travaux pratiques 33.

- Pour un polynôme $P \in \mathbb{C}[X]$ et un réel $r > 0$, tracer l'image par P du cercle centré à l'origine de \mathbb{C} et de rayon r .
- Faites varier r (ou mieux faites une animation). En quoi cela illustre-t-il le théorème de d'Alembert-Gauss?
- Application à $P(X) = X^4 - X^3 + X^2 - iX + 1 \in \mathbb{C}[X]$.

Voici quelques images de l'animation pour des valeurs de r valant successivement : $r_0 = 0.5$, $r_1 = 0.6176\dots$, $r_2 = 0.9534\dots$, $r_3 = 1.2082\dots$, $r_4 = 1.4055\dots$ et $r_5 = 1.5$.



Quand la courbe \mathcal{C}_r passe-t-elle par l'origine ? La courbe passe par l'origine s'il existe un nombre complexe re^{it} tel que $P(re^{it}) = 0$ autrement dit lorsque P admet une racine de module r . D'après le théorème de d'Alembert-Gauss un polynôme de degré n a au plus n racines. Alors il y a au plus n valeurs r_1, \dots, r_n pour lesquelles \mathcal{C}_{r_i} passe par l'origine. Pour notre exemple de degré 4, il y a 4 racines, qui conduisent ici à 4 modules distincts, r_1, \dots, r_4 .

Voici comment procéder :

- On commence par définir l'anneau $\mathbb{C}[X]$ par $\text{R.}\langle X \rangle = \text{CC}[]$. Les calculs seront donc des calculs approchés avec des nombres complexes. Notre polynôme P est défini par $P = X^4 - X^3 + X^2 - I*X + 1$.
- Le cercle de rayon r est l'ensemble des complexes de la forme re^{it} pour $t \in [0, 2\pi]$. La courbe \mathcal{C}_r cherchée est donc l'ensemble

$$\mathcal{C}_r = \{P(re^{it}) \mid t \in [0, 2\pi]\}.$$

Voici la fonction qui renvoie ce tracé :

Code 115 (*intro-polynome.sage (2)*).

```
def plot_image_cercle(P,r):
    var('t')
    zreal = P(r*exp(I*t)).real()
    zimag = P(r*exp(I*t)).imag()
    G = parametric_plot( (zreal,zimag), (t,0,2*pi) )
    return G
```

- Une animation est une juxtaposition d'images. On la définit par :

```
A = animate( [plot_image_cercle(P,r) for r in strange(0.5,1.5,0.05)] )
```

puis on l'affiche par `A.show()`.

Remarque.

Par défaut Sage fait les calculs dans un ensemble appelé SR (*Symbolic Ring*). Dans cet ensemble vous pouvez définir des expressions polynomiales et en trouver les racines. L'accès aux fonctions spécifiques aux polynômes (comme le degré, les coefficients,...) est un peu différent. Voici comment déclarer un polynôme, récupérer son degré ainsi que le coefficient devant X^2 :

Code 116 (*intro-polynome.sage (3)*).

```
var('X')
P = X^4 - 3*X^2 - 1
P.degree(X)
P.coefficient(X,2)
```

8.2. Algorithme de Horner

L'algorithme de Horner permet d'évaluer rapidement un polynôme P en une valeur α . Voyons comment il fonctionne à la main avec l'exemple de $P(X) = X^5 + X^4 - 5X^3 - 3X - 2$ et $\alpha = 2$.

Sur la première ligne du tableau, on écrit les coefficients de P , $a_n, a_{n-1}, \dots, a_1, a_0$. On reporte en troisième ligne première colonne, a_n , le coefficient dominant de P . On multiplie ce nombre par α , on l'écrit en deuxième ligne, deuxième colonne, puis on ajoute a_{n-1} que l'on écrit en troisième ligne, deuxième colonne. Et on recommence. À chaque étape, on multiplie le dernier nombre obtenu par α et on lui ajoute un coefficient de P . Le dernier nombre obtenu est $P(\alpha)$.

a_k	1	1	-5	0	-3	-2
$\alpha = 2$		2	6	2	4	2
b_k	1	3	1	2	1	0

Ici le dernier nombre est 0, donc $P(2) = 0$. Conclusion : 2 est racine de P .

Avec le même polynôme et $\alpha = -1$ le tableau se complète ainsi :

a_k	1	1	-5	0	-3	-2
$\alpha = -1$		-1	0	5	-5	8
b_k	1	0	-5	5	-8	6

ce qui implique $P(-1) = 6$.

Le travail à faire suivant formalise cette procédure et montre que l'algorithme de Horner permet des calculs efficaces avec les polynômes.

Travaux pratiques 34.

On fixe un corps K , et $\alpha \in K$. Soit $P(X) = a_nX^n + a_{n-1}X^{n-1} + \dots + a_kX^k + \dots + a_1X + a_0 \in K[X]$.

On pourra prendre pour les applications $P(X) = X^5 + X^4 - 5X^3 - 3X - 2$ et $\alpha = 2$, puis $\alpha = -1$.

1. Compter et comparer le nombre de multiplications nécessaires dans K pour calculer $P(\alpha)$, par :

(a) le calcul direct :

$$P(\alpha) = a_n\alpha^n + a_{n-1}\alpha^{n-1} + \dots + a_k\alpha^k + \dots + a_1\alpha + a_0,$$

(b) l'algorithme de Horner :

$$P(\alpha) = ((a_n\alpha + a_{n-1})\alpha + a_{n-2})\alpha + \dots + a_1)\alpha + a_0.$$

(c) Écrire une fonction qui calcule $P(\alpha)$ par l'algorithme de Horner.

2. On formalise le calcul précédent en définissant la suite :

$$b_n = a_n \quad \text{puis pour } n-1 \geq k \geq 0 : b_k = \alpha b_{k+1} + a_k.$$

(a) Montrer que dans la division euclidienne de $P(X)$ par $X - \alpha$ qui s'écrit :

$$P(X) = (X - \alpha)Q(X) + P(\alpha)$$

le reste $P(\alpha)$ est égal à b_0 et le quotient $Q(X)$ est égal au polynôme $b_nX^{n-1} + \dots + b_2X + b_1$ dont les coefficients sont les b_k pour $k \geq 1$.

Indications. Pour ce faire développer $(X - \alpha)Q(X)$ et retrouver pour les coefficients de Q la même relation de récurrence que celle définissant la suite des b_k .

(b) Modifier votre fonction afin qu'elle calcule la suite $(b_n, b_{n-1}, \dots, b_1, b_0)$ et qu'elle renvoie le quotient Q et le reste $b_0 = P(\alpha)$.

(c) Écrire une fonction qui calcule l'expression d'un polynôme $P(X)$ dans la base des $(X - \alpha)^k$, c'est-à-dire calculer les $c_k \in K$ tels que :

$$P(X) = c_n(X - \alpha)^n + c_{n-1}(X - \alpha)^{n-1} + \dots + c_1(X - \alpha) + c_0.$$

1. (a) Le calcul d'un monôme $a_k \cdot \alpha^k$ de degré k nécessite k multiplications, donc pour calculer $P(\alpha)$ il faut $n + (n-1) + \dots + k + \dots + 1 + 0 = \frac{n(n+1)}{2}$ multiplications (et n additions).

(b) La méthode de Horner

$$P(\alpha) = ((a_n \alpha + a_{n-1})\alpha + a_{n-2})\alpha + \cdots + a_1)\alpha + a_0.$$

nécessite seulement n multiplications (et n additions). On passe d'un coût d'ordre $\frac{1}{2}n^2$ à un coût d'ordre n ; le gain est donc énorme.

(c) Pour l'implémentation, on note que la formule de récurrence se fait pour des indices k allant en décroissant.

Code 117 (*horner.sage (1)*).

```
def eval_horner(P,alpha):
    n = P.degree()
    val = P[n]
    for k in range(n-1,-1,-1):
        val = alpha*val + P[k]
    return val
```

2. (a) On note la division euclidienne de $P(X)$ par $X - \alpha$ sous la forme $P(X) = (X - \alpha)Q(X) + R$. Le polynôme Q est de degré $n - 1$ et on le note $Q(X) = b'_n X^{n-1} + \cdots + b'_k X^{k-1} + \cdots + b'_2 X + b'_1$ (on veut montrer $b'_k = b_k$). Le reste R est de degré strictement inférieur à $\deg(X - \alpha) = 1$ donc R est un polynôme constant. En évaluant l'égalité de la division euclidienne en α , on a immédiatement $P(\alpha) = R$, que l'on note pour l'instant b'_0 .

L'égalité $P(X) = (X - \alpha)Q(X) + R$ s'écrit :

$$P(X) = (X - \alpha)(b'_n X^{n-1} + \cdots + b'_k X^{k-1} + \cdots + b'_2 X + b'_1) + b'_0.$$

On développe le terme de droite et on regroupe les monômes de même degré :

$$P(X) = b'_n X^n + (b'_{n-1} - \alpha b'_n)X^{n-1} + \cdots + (b'_k - \alpha b'_{k+1})X^k + \cdots + (b'_1 - \alpha b'_2)X + (b'_0 - \alpha b'_1)$$

On identifie coefficient par coefficient :

$$\left\{ \begin{array}{lcl} b'_n & = & a_n \\ b'_{n-1} - \alpha b'_n & = & a_{n-1} \\ \dots & & \\ b'_k - \alpha b'_{k+1} & = & a_k \\ \dots & & \\ b'_0 - \alpha b'_1 & = & a_0 \end{array} \right. \quad \text{donc} \quad \left\{ \begin{array}{lcl} b'_n & = & a_n \\ b'_{n-1} & = & \alpha b'_n + a_{n-1} \\ \dots & & \\ b'_k & = & \alpha b'_{k+1} + a_k \\ \dots & & \\ b'_0 & = & \alpha b'_1 + a_0 \end{array} \right.$$

Les suites (b'_k) et (b_k) sont définies par le même terme initial a_n et la même relation de récurrence, elles sont donc égales.

(b) On modifie légèrement le code précédent. La suite $(b_n, b_{n-1}, \dots, b_1)$ donne les coefficients de Q (attention au décalage) et b_0 donne le reste qui n'est autre que $P(\alpha)$.

Code 118 (*horner.sage (2)*).

```
def division_horner(P,alpha):
    n = P.degree()
    if n <= 0: return 0,P
    b = [None] * (n+1)      # Liste triviale de longueur n+1
    b[n] = P[n]
    for k in range(n-1,-1,-1):
        b[k] = alpha*b[k+1] + P[k]
    Q = sum( b[k+1]*X^k for k in range(0,n) )
    R = b[0]
```

```
return Q,R
```

- Pour notre polynôme $P(X) = X^5 + X^4 - 5X^3 - 3X - 2$ et $\alpha = 2$, l'appel de la fonction définie ci-dessus : `division_horner(P, alpha)` renvoie un reste R nul et un quotient $Q(X) = X^4 + 3X^3 + X^2 + 2X + 1$. Ce qui signifie que $X - 2$ divise $P(X)$, ou encore $P(2) = 0$.
 - Pour ce même polynôme et $\alpha = -1$, la commande `division_horner(P, alpha)` renvoie $Q(X) = X^4 - 5X^2 + 5X - 8$ et un reste $R = 6$.
- (c) On obtient les c_k comme restes de divisions euclidiennes successives par $X - \alpha$. On commence par la division euclidienne de $P(X)$ par $X - \alpha$. On a vu comment calculer le quotient Q_1 et le reste $b_0 = P(\alpha)$. En évaluant l'expression $P(X) = c_n(X - \alpha)^n + \dots + c_1(X - \alpha) + c_0$ en α on voit que l'on a aussi $c_0 = P(\alpha)$ donc c_0 est le reste de la première division par $X - \alpha$.
On a donc $P(X) = (X - \alpha)Q_1(X) + c_0$. On recommence en divisant Q_1 par $X - \alpha$, on obtient un quotient Q_2 et un reste qui va être c_1 . Donc $P(X) = (X - \alpha)((X - \alpha)Q_2 + c_1) + c_0$.
On continue ainsi de suite jusqu'à ce que le quotient soit nul (au bout de $n + 1$ étapes), on a alors

$$P(X) = ((c_n(X - \alpha) + c_{n-1})(X - \alpha) + c_{n-2})(X - \alpha) + \dots + c_1(X - \alpha) + c_0$$

soit en développant :

$$P(X) = c_n(X - \alpha)^n + \dots + c_1(X - \alpha) + c_0$$

Le code suivant renvoie les coefficients (c_0, c_1, \dots, c_n) .

Code 119 (`horner.sage (3)`).

```
def developpe_horner(P,alpha):
    Q = P
    coeff = []
    while Q != 0:
        Q,R = division_horner(Q,alpha)
        coeff.append(R)
    return coeff
```

- Pour $P(X) = X^5 + X^4 - 5X^3 - 3X - 2$ et $\alpha = 2$, la commande `developpe_horner(P, alpha)` renvoie la liste `[0, 49, 74, 43, 11, 1]`, ce qui signifie :

$$P(X) = 1 \cdot (X - 2)^5 + 11 \cdot (X - 2)^4 + 43 \cdot (X - 2)^3 + 74 \cdot (X - 2)^2 + 49 \cdot (X - 2).$$

- Pour le même polynôme et $\alpha = -1$, la fonction renvoie `[6, -17, 11, 1, -4, 1]`, donc

$$P(X) = 1 \cdot (X + 1)^5 - 4 \cdot (X + 1)^4 + 1 \cdot (X + 1)^3 + 11 \cdot (X + 1)^2 - 17 \cdot (X + 1) + 6.$$

8.3. Interpolation de Lagrange

Théorème 1 (Interpolation de Lagrange).

Soient (x_i, y_i) , $i = 0, \dots, n$, une suite de $n + 1$ points, d'abscisses deux à deux distinctes. Il existe un unique polynôme P de degré inférieur ou égal à n tel que

$$P(x_i) = y_i \quad \text{pour } i = 0, \dots, n.$$

En particulier, pour toute fonction f continue sur un intervalle contenant x_0, \dots, x_n , il existe un unique polynôme P de degré inférieur ou égal à n tel que

$$P(x_i) = f(x_i) \quad \text{pour } i = 0, \dots, n.$$

Travaux pratiques 35.

1. Montrer l'unicité du polynôme P dans le théorème d'interpolation de Lagrange.
2. Pour l'existence, étant donnés x_0, \dots, x_n , on définit les polynômes de Lagrange L_0, L_1, \dots, L_n :

$$L_i(X) = \prod_{j \neq i} \frac{X - x_j}{x_i - x_j}$$

Montrer que le polynôme :

$$P(X) = \sum_{i=0}^n y_i L_i(X)$$

répond au problème : $P(x_i) = y_i$ ($i = 0, \dots, n$) et $\deg P \leq n$.

3. Écrire une fonction qui, étant donnée une liste de points, renvoie le polynôme d'interpolation P .
4. Interpoler la fonction définie par $f(x) = \sin(2\pi x)e^{-x}$ sur l'intervalle $[0, 2]$, avec une subdivision régulière de $n + 1$ points. Tracer les graphes de la fonction f et des polynômes d'interpolation correspondant à différentes valeurs de n .
5. Faire le même travail avec la fonction définie par $f(x) = \frac{1}{1+8x^2}$ sur l'intervalle $[-1, 1]$, avec une subdivision régulière de $n + 1$ points. Quel problème apparaît ? C'est le *phénomène de Runge*.

1. Supposons que P et Q soient deux polynômes de degré $\leq n$ vérifiant tous les deux $P(x_i) = y_i$. Alors le polynôme $P - Q$ vérifie $(P - Q)(x_i) = 0$, $i = 0, \dots, n$. Ainsi $P - Q$ est un polynôme de degré inférieur à n ayant $n + 1$ racines. D'après le théorème de d'Alembert-Gauss, c'est nécessairement le polynôme nul. Ainsi $P - Q = 0$, donc $P = Q$.

2. Les polynômes L_i sont de degré exactement n et sont définis de sorte que

$$L_i(x_i) = 1 \quad \text{et} \quad L_i(x_j) = 0 \quad \text{pour } j \neq i.$$

Il est alors clair que $P(X) = \sum_{j=0}^n y_j L_j(X)$ est aussi de degré inférieur à n et vérifie pour $i = 0, \dots, n$:

$$P(x_i) = \sum_{j=0}^n y_j L_j(x_i) = y_i L_i(x_i) = y_i.$$

3. On commence par définir :

- `R.<X> = RR[]` : l'anneau de polynômes (les coefficients sont des réels approchés) ;
- `f = sin(2*pi*x)*exp(-x)` : la fonction ;
- `liste_points = [(2*i/n, f(x=2*i/n)) for i in range(n+1)]` : une liste de points du graphe de f .

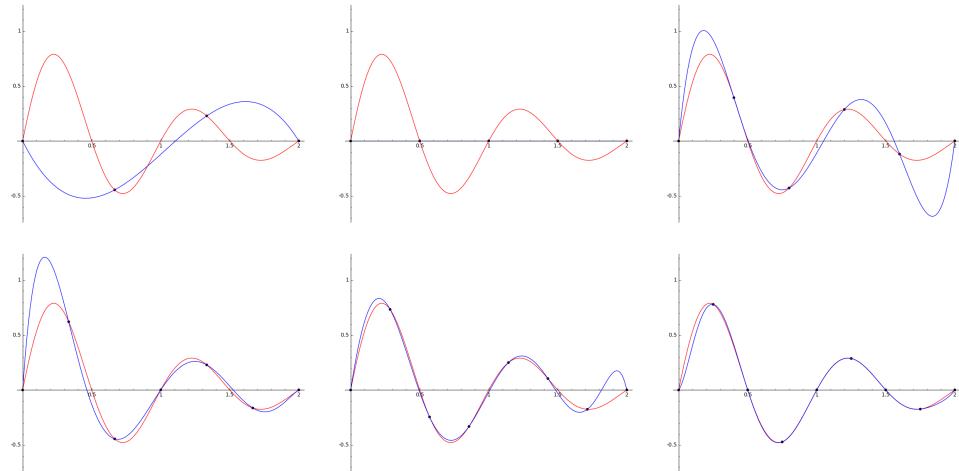
La fonction suivante renvoie le polynôme interpolateur d'une liste de points.

Code 120 (*interpolation.sage*).

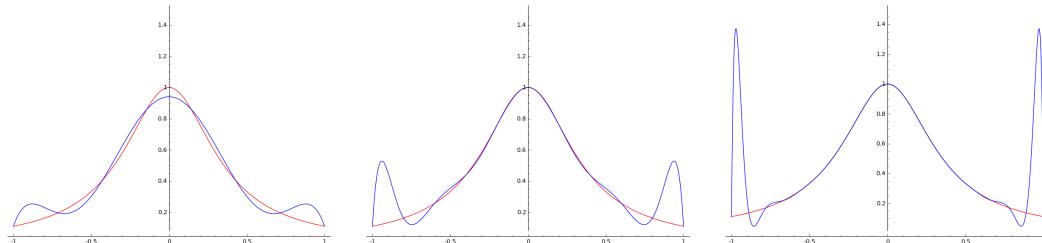
```
def interpolation_lagrange(liste_points):
    n = len(liste_points)-1
    allx = [p[0] for p in liste_points] # Abscisses
    ally = [p[1] for p in liste_points] # Ordonnées
    liste_lagrange = []
    for i in range(n+1):
        A = prod(X-x for x in allx if x != allx[i])          # Numérateur
        B = prod(allx[i]-x for x in allx if x != allx[i])    # Dénominateur
        L = A/B                                                 # Polynôme de Lagrange
        liste_lagrange.append(L)
    # Le polynôme interpolateur :
    lagrange = sum( liste_lagrange[i]*ally[i] for i in range(n+1) )
```

```
return lagrange
```

4. Voici les tracés (en rouge la fonction, en bleu le polynôme) pour l'interpolation de $f(x) = \sin(2\pi x)e^{-x}$ avec $n = 3, 4, 5, 6, 7, 8$. Pour $n = 4$, le polynôme est le polynôme nul. À partir de $n = 8$, il devient difficile de distinguer le graphe de la fonction, du graphe du polynôme.



5. Voici l'interpolation de $f(x) = \frac{1}{1+8x^2}$ pour $n = 7, 10, 18$. La zone centrale de la courbe est bien interpolée mais autour de -1 et $+1$, des «bosses» apparaissent. Ces bosses deviennent de plus en plus grandes en se décalant vers les extrémités. Il y a convergence simple mais pas convergence uniforme. Pour éviter ce phénomène, il faudrait mieux choisir les x_i .



9. Équations différentielles

9.1. Équations différentielles $y' = f(x, y)$

La commande `desolve` permet de résoudre formellement des équations différentielles. Consultez la documentation, accessible par `help(desolve)`.

Travaux pratiques 36.

1. (a) Résoudre l'équation différentielle :

$$y' = y + x + 1.$$

Il s'agit donc de trouver toutes les fonctions $x \mapsto y(x)$ dérivables, telles que $y'(x) = y(x) + x + 1$, pour tout $x \in \mathbb{R}$.

- (b) Résoudre l'équation différentielle avec condition initiale :

$$y' = y + x + 1 \quad \text{et} \quad y(0) = 1.$$

2. (a) Résoudre sur \mathbb{R}_+^* l'équation différentielle :

$$x^2 y' = (x - 1)y.$$

(b) Trouver la solution vérifiant $y(1) = 2$.

(c) Peut-on trouver une solution sur \mathbb{R} ?

3. Calculer la **fonction logistique** $y(x)$, solution de l'équation différentielle

$$y' = y(1 - y) - 1.$$

1.

Code 121 (*equadiff-intro.sage (1)*).

```
var('x')
y = function('y',x)
yy = diff(y,x)
desolve(yy == y+x+1, y)
desolve(yy == y+x+1, y, ics=[0,1])
```

(a) On commence par déclarer la variable x . On définit une fonction $x \mapsto y(x)$ ainsi que sa dérivée $y'(x)$ par `diff(y, x)`. On prendra ici la convention de noter y' par `yy`. L'équation différentielle $y' = y + x + 1$ s'écrit donc `yy == y+x+1`. On lance la résolution de l'équation différentielle par la fonction `desolve`, avec pour argument l'équation différentielle à résoudre, ainsi que la fonction inconnue, qui est ici y . Sage renvoie :

$$-((x + 1)*e^{-x} - _C + e^{-x})*e^x$$

$_C$ désigne une constante. Après simplification évidente, les solutions sont les :

$$y(x) = -x - 2 + C \exp(x) \quad \text{où} \quad C \in \mathbb{R}.$$

(b) Il est possible de préciser une condition initiale grâce à l'argument optionnel `ics` (voir la dernière ligne du code ci-dessus), ici on spécifie donc que l'on cherche la solution vérifiant $y(0) = 1$. L'unique solution est alors : $y(x) = -x - 2 + 3 \exp(x)$, c'est-à-dire $C = 3$.

2.

Code 122 (*equadiff-intro.sage (2)*).

```
equadiff = x^2*yy == (x-1)*y
assume(x>0)
desolve(equadiff, y)
desolve(equadiff, y, ics=[1,2])
forget()
assume(x<0)
desolve(equadiff, y)
```

(a) En se restreignant à \mathbb{R}_+^* ($x > 0$), on trouve $y(x) = Cx \exp(\frac{1}{x})$, $C \in \mathbb{R}$.

(b) La solution vérifiant $y(1) = 2$ est $y(x) = 2x \exp(\frac{1}{x} - 1)$, c'est-à-dire $C = 2 \exp(-\frac{1}{e})$.

(c) Sur \mathbb{R}_-^* ($x < 0$), les solutions sont aussi de la forme $y(x) = Cx \exp(\frac{1}{x})$.

La seule solution définie sur \mathbb{R} est donc la fonction nulle $y(x) = 0$ (et $C = 0$). En effet si $C \neq 0$ il n'y a pas de limite finie en 0.

3.

Code 123 (*equadiff-intro.sage* (3)).

```
equadiff = yy == y*(1-y) - 1
sol_impl = desolve(equadiff, y)
sol = solve(sol_impl,y)
```

Sage ne résout pas directement cette équation différentielle. En effet, il renvoie une équation vérifiée par y :

$$-\frac{2}{3}\sqrt{3}\arctan\left(\frac{1}{3}\sqrt{3}(2y(x)-1)\right)=C+x. \quad (1)$$

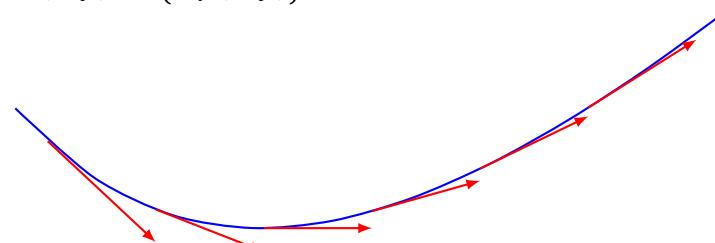
On demande alors à Sage de résoudre cette équation, pour obtenir :

$$y(x)=-\frac{\sqrt{3}}{2}\tan\left(\frac{\sqrt{3}}{2}C+\frac{\sqrt{3}}{2}x\right)+\frac{1}{2}. \quad (2)$$

L'équation (1) a l'avantage d'être valide quel que soit x , alors que pour l'équation (2) il faudrait préciser l'intervalle de définition.

9.2. Interprétation géométrique

- Les **courbes intégrales** d'une équation différentielle $y' = f(x, y)$ sont les graphes des solutions de cette équation.
- À chaque point d'une courbe intégrale, on associe un vecteur tangent. Si $y(x)$ est une solution de l'équation différentielle alors au point $(x, y(x))$ la tangente est portée par le vecteur $(x', y'(x))$. D'une part $x' = 1$ et d'autre part comme y est solution de l'équation différentielle alors $y'(x) = f(x, y)$. Ainsi un **vecteur tangent** en (x, y) est $(1, f(x, y))$.



- Le **champ de vecteurs** associé à l'équation différentielle $y' = f(x, y)$ est, en chaque point (x, y) du plan, le vecteur $(1, f(x, y))$.
- Voici ce qui signifie géométriquement « résoudre » une équation différentielle. À partir d'un champ de vecteur, c'est-à-dire la donnée d'un vecteur attaché à chaque point du plan, il s'agit de trouver une courbe intégrale, c'est-à-dire une courbe qui en tout point est tangente aux vecteurs.
- Comme on ne s'occupera pas de la norme des vecteurs, la donnée d'un champ de vecteur revient ici à associer à chaque point, la pente de la tangente au graphe d'une solution passant par ce point.

Travaux pratiques 37.

Nous allons étudier graphiquement l'équation différentielle

$$y' = -xy.$$

1. Représenter le champ de vecteurs associé à cette équation différentielle.
2. (a) Résoudre l'équation.
(b) Tracer, sur un même graphe, les courbes intégrales correspondant aux solutions définies par $y(0) = k$, pour différentes valeurs de $k \in \mathbb{R}$.
(c) Que remarquez-vous ? Quel théorème cela met-il en évidence ?
3. Tracer quelques isoclines de cette équation différentielle.

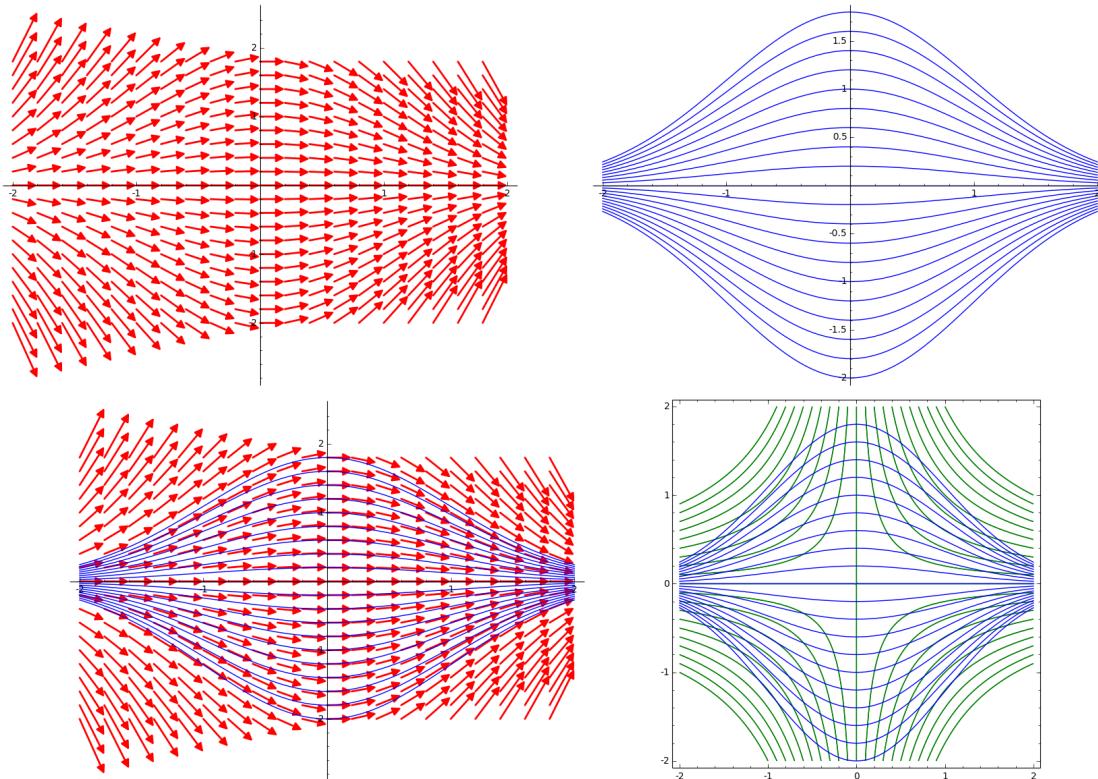
Les **isoclines** de l'équation différentielle $y' = f(x, y)$ sont les courbes sur lesquelles la tangente d'une courbe intégrale a une pente donnée c , c'est-à-dire qu'une isocline est un ensemble

$$\{(x, y) \in \mathbb{R}^2 \mid f(x, y) = c\}.$$

À ne pas confondre avec les solutions ! En chaque point d'une isocline, la solution passant par ce point croise cette isocline avec une pente c .

Voici quelques graphes :

- Le champ de vecteurs (en rouge).
- Les courbes intégrales (en bleu).
- Le champ de vecteurs superposé aux courbes intégrales : les vecteurs sont bien tangents aux courbes intégrales.
- Les isoclines (en vert). Pour une isocline fixée, vérifier que chaque intersection avec les courbes intégrales se fait en des points où la tangente a toujours la même pente.



1. Voici comment tracer le champ de vecteurs associé à $y' = f(x, y)$.

Code 124 (*equadiff-courbe.sage* (1)).

```
def champ_vecteurs(f,xmin,xmax,ymin,ymax,delta):
    G = Graphics()
    for i in range(xmin, xmax, delta):
        for j in range(ymin, ymax, delta):
            pt = vector([i,j])
            v = vector([1,f(u=i,v=j)])
            G = G + arrow(pt,pt+v)
    return G
```

- Les variables de la fonction f sont ici u et v pour ne pas interférer avec la variable x et la fonction y .
- N'hésitez pas à renormaliser les vecteurs v , (par exemple en $v/10$) pour plus de lisibilité.
- En fait la fonction `plot_vector_field`, prédéfinie dans Sage, trace aussi les champs de vecteurs.

2. (a) Les solutions de l'équation sont les

$$y(x) = C \exp\left(-\frac{x^2}{2}\right) \quad \text{où} \quad C \in \mathbb{R}.$$

(b) Le code suivant résout l'équation différentielle et trace la solution avec la condition initiale $y(0) = k$, pour k variant de y_{\min} à y_{\max} avec un pas de δ .

Code 125 (*equadiff-courbe.sage (2)*).

```
def courbes_integrales(equadiff,a,b,kmin,kmax,delta):
    G = Graphics()
    for k in range(kmin, kmax, delta):
        sol = desolve(equadiff, y, ics=[0,k])
        G = G + plot(sol, (x, a, b))
    return G
```

(c) Les courbes intégrales forment une partition de l'espace : par un point il passe exactement une courbe intégrale. C'est une conséquence du théorème d'existence et d'unicité de Cauchy-Lipschitz.

3. Les isoclines sont définies par l'équation $f(x, y) = c$. Elles se tracent par la commande

```
implicit_plot(f-c, (x,xmin,xmax), (y,ymin, ymax))
```

Ici les isoclines ont pour équation $-xy = c$, ce sont donc des hyperboles.

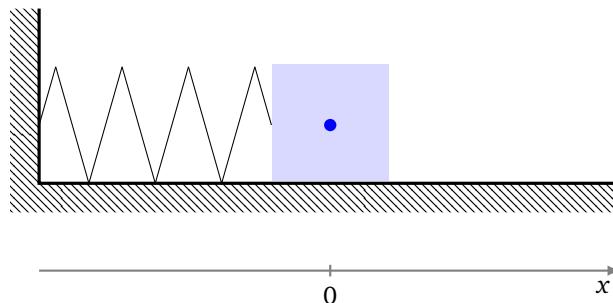
9.3. Équations différentielles du second ordre

Travaux pratiques 38.

L'équation différentielle

$$x''(t) + f x'(t) + \frac{k}{m} x(t) = 0$$

correspond au mouvement d'une masse m attachée à un ressort de constante $k > 0$ et des frottements $f \geq 0$.



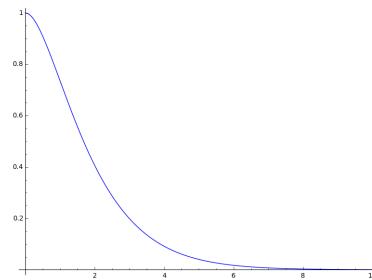
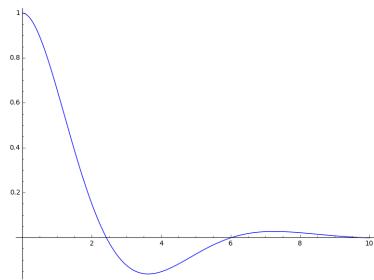
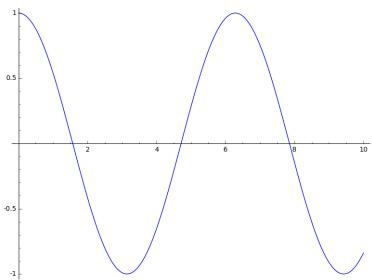
Résoudre et tracer les solutions pour différentes valeurs de $f \geq 0$ (prendre $k = 1$, $m = 1$), avec les conditions initiales :

$$x(0) = 1 \quad \text{et} \quad x'(0) = 0.$$

Pour une valeur de f , l'équation se résout par :

```
desolve(diff(x,t,2) + f*diff(x,t) + k/m*x(t) == 0, x, ics=[0,1,0])
```

Voici les solutions pour $f = 0$, $f = 1$, $f = 2$:



- Cas $f = 0$. Il n'y a pas de frottement. L'équation différentielle s'écrit $x'' + \frac{k}{m}x = 0$. Pour $k = 1$, $m = 1$ alors l'équation est juste $x'' + x = 0$ dont les solutions sont de la forme

$$x(t) = \lambda \cos t + \mu \sin t \quad \lambda, \mu \in \mathbb{R}.$$

Avec les conditions initiales $x(0) = 1$ et $x'(0) = 0$, l'unique solution est

$$x(t) = \cos t.$$

Il s'agit donc d'un mouvement périodique.

- Cas $0 < f < 2$. Les frottements sont faibles. Par exemple pour $f = 1$ la solution est

$$\left(\frac{\sqrt{3}}{3} \sin\left(\frac{\sqrt{3}}{2}t\right) + \cos\left(\frac{\sqrt{3}}{2}t\right) \right) e^{-\frac{1}{2}t}.$$

Il s'agit d'un mouvement amorti oscillant autour de la position d'origine $x = 0$.

- Cas $f \geq 2$. Les frottements sont forts. Par exemple pour $f = 2$ la solution est

$$x(t) = (t + 1)e^{-t}.$$

Il n'y a plus d'oscillations autour de la position d'origine.

Auteurs du chapitre

- Arnaud Bodin, Niels Borne, François Recher
- D'après un cours de calcul formel de Saïd Belmehdi, Jean-Paul Chehab, Bernard Germain-Bonne, François Recher donné à l'université Lille 1.
- Relu par Marc Mezzarobba et Paul Zimmermann.

Les auteurs

Ce livre est diffusé sous la licence *Creative Commons – BY-NC-SA – 3.0 FR*. Sur le site Exo7 vous pouvez le télécharger gratuitement et aussi récupérer les fichiers sources.

Version 0.1 – Juin 2016