Algorithmes pour l'enseignement des mathématiques

13 décembre 2022

Table des matières

Chapitre 1

Généralités sur les algorithmes

1.1 Introduction

Définition 1.1. *Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre une classe de problèmes.*

Exemples : Algorithmes vus à l'école primaire : addition de deux nombres entiers, multiplication de deux nombres entiers en la posant, division euclidienne, algorithme d'Euclide ...

Le terme *algorithme* provient du nom du mathématicien persan Al-Khwârismî, né dans les années 780 et mort vers 850. Cependant, les algorithmes sont antérieurs à ce mathématicien, puisque, par exemple, on a retrouvé une tablette d'argile datant d'entre 2000 et 1650 avant J.-C indiquant que les babyloniens avaient des méthodes algorithmiques pour calculer l'inverse de certains nombres (voir [Cha94]).

Dans ce cours, nous verrons quelques bases de programmation. Les exemples que nous donnerons seront rédigés en *pseudo-langage*, c'est à dire en langage « presque naturel », sans faire référence à un langage de programmation particulier. Lorsque nous implémenterons des algorithmes sur un ordinateur en revanche, nous utiliserons principalement le langage *Python*. Comme référence pour ce langage, on pourra par exemple utiliser [Swi12], qui est disponible librement en pdf.

1.2 Les types de données

1.2.1 Variables et valeurs

Pour mener à bien des calculs, il est nécessaire de manipuler des quantités données. Par exemple, pour calculer $P(2)$ avec $P(x) = x^2 - 1$, on remplace la valeur 2 partout où apparaît la variable x. Le vocabulaire de l'algorithmique garde les mêmes mots.

Une valeur est une quantité connue au moment du calcul. Le calcul concret va faire intervenir des valeurs. Par exemple, nous verrons souvent apparaître :

— des valeurs entières : 0, 72, -3, 25643, ...
— des valeurs flottantes (qui représentent des nombres réels) : 3.14159, 2.5, -21.876543, ...
— des valeurs booléennes : vrai ou faux
— des chaînes de caractères, par exemple : 'bonjour'

Une variable est un nom, qui va servir à décrire des calculs. En effet, en mathématiques on utilise la notation $P(x) = x^2 - 1$, et si on calcule l'image de x par la fonction $x \mapsto P(x)$, alors on devra porter x au carré et soustraire 1. Ceci reste vrai pour n'importe quelle valeur[1] $x = 1$, $x = \pi$ ou $x = -2.5$. En algorithmique, on utilisera aussi ce concept, et les variables serviront à stocker des valeurs.

Ceci permettra d'organiser le calcul par étapes, en stockant les résultats intermédiaires. On pourra imaginer qu'une variable est le nom d'un tiroir dans lequel on range une valeur, et dont on peut lire la valeur qui y est rangée.

Bien que $\mathbb{Z}$ soit inclus dans $\mathbb{R}$, il est souvent important de voir les entiers comme des entiers en Python et non comme des réels (cela se fait automatiquement en général). Cela permet par exemple de faire certaines opérations sur ces nombres, qui ne sont pas autorisées pour les réels (par exemple réduire un nombre modulo 2). Cela permet aussi de faire des calculs exacts et d'éviter les problèmes d'arrondis propres aux flottants. Par exemple en Python, $1+10^{-13} = 1$. Pour accéder au type d'un objet a (par exemple « entier », « flottant », ...), on peut taper type(a). On constate que pour Python, 1 est vu comme un entier, alors que 1.0 est vu comme un flottant. Et $1 + 1.0$ est un flottant.

1.2.2 Les listes

Définition. *Une **liste** est une suite ordonnée d'expressions.*

Exemple. $[4, 5, 7]$, $[5, [2, 7]]$,...

La liste [ ] est appelée la **liste vide**.

Définition. *La **longueur** d'une liste est le nombre d'éléments de la liste.*

Exemples.
- *La longueur de* $[a_1, \ldots, a_n]$ *vaut* n.
- *La longueur de la liste vide vaut 0.*
- *La longueur de* $[3, 1, [2, 7], [3, 5, 6], [[2], 5]]$ *vaut 5.*

Généralement, les indices sont des entiers allant de 0 à la longueur de la liste à laquelle on retire 1.

On accède à une case du tableau en précisant son indice entre crochets.

Exemple. *Pour* $L = [0, 5, 7, 3, 12]$, , $L[0] = 0$ *et* $L[4] = 12$.

Attention !!!. *Bien vérifier, lorsqu'on accède à une case de la liste, que celle-ci existe bien. Par exemple,* $L[5]$ *renverra « erreur ».*

1. Notons que ceci est vrai car le domaine de définition de $x \mapsto P(x)$ est $\mathbb{R}$ tout entier.

1.3 Les actions fondamentales

Un algorithme est une suite d'actions. On souhaite en faire une description précise et sans ambiguïté. Dans ce but, nous convenons dans cette partie d'un pseudo-langage en même temps que nous dressons la liste des actions fondamentales.

1.3.1 L'affectation

Plus haut, nous avons parlé de valeurs et de variables. L'affectation consiste à stocker une valeur dans une variable, et on la note

$$variable \leftarrow valeur.$$

Par exemple :

$$\begin{aligned} xEntier &\leftarrow 3 \\ xFlottant &\leftarrow 3.14159 \\ xBooleen &\leftarrow vrai \end{aligned}$$

Utilisation de la valeur stockée dans une variable Une fois qu'une variable a reçu une valeur, on peut utiliser la variable pour rappeler la valeur. Par exemple, si x contient la valeur 3, alors on peut former l'expression :

$$x + 3$$

qui est une valeur, et qui vaut 6.

Modification de la valeur stockée Il est important de comprendre que seule la variable à gauche de la flèche est modifiée. Ainsi, l'affectation :

$$x \leftarrow pam + 3$$

ne change que le contenu de la variable x, et pas celui de la variable pam. En effet, à droite de la flèche, on n'utilise pam que pour sa valeur. On peut aussi écrire :

$$\begin{aligned} x &\leftarrow 3 \\ x &\leftarrow x + 4 \end{aligned}$$

A la fin de ces deux instructions, la variable x contient la valeur 7.

Ordre des instructions Dans nos algorithmes, on pourra en général suivre les valeurs prises par les variables. Il faut toutefois garder à l'esprit que l'ordre dans lequel on effectue les instructions est important.

Par exemple, à la fin des instructions :

$$\begin{aligned} i &\leftarrow 12 \\ i &\leftarrow 34 \end{aligned}$$

la variable i contient la valeur 34.

A la fin des instructions :

$$\begin{aligned} i &\leftarrow 34 \\ i &\leftarrow 12 \end{aligned}$$

la variable i contient la valeur 12. Examinons un autre exemple. A la fin des instructions :

$$\begin{aligned} i &\leftarrow 34 \\ j &\leftarrow i+2 \\ i &\leftarrow i+1 \end{aligned}$$

la variable i contient la valeur 35 et la variable j contient la valeur 36. Par contre, à la fin des instructions :

$$\begin{aligned} i &\leftarrow 34 \\ i &\leftarrow i+1 \\ j &\leftarrow i+2 \end{aligned}$$

la variable i contient la valeur 35 et la variable j contient la valeur 37.

1.3.2 Les instructions conditionnelles

Ces structures servent lorsqu'on est confronté à des situations pour lesquelles une ou plusieurs instructions ne peuvent être exécutées que dans certaines conditions.

Elle sert lorsqu'un bloc d'instructions ne peut être exécuté que si une condition est vraie.

Schéma.
```
Condition(paramètres)
si condition
  | faire...
```

Exemples :

Algorithme de calcul de la valeur absolue d'un nombre. L'algorithme suivant prend en entrée un réel (ou rationnel, ou entier, ...) et détermine sa valeur absolue :

Algorithme Abs(a)
si a<0
| renvoyer $-a$
sinon
| renvoyer a.

Algorithme de calcul (exact) des racines réelles d'un polynôme de degré 2 à coefficients dans $\mathbb{R}$. L'algorithme suivant prend en entrée trois réels (ou rationnels, ou entiers, ...) a, b, c, avec $a \neq 0$ et renvoie la liste des solutions de l'équation $ax^2 + bx + c = 0$ d'inconnue $x \in \mathbb{R}$.

Algorithme Racines(a, b, c)
Delta $\leftarrow b^2 - 4ac$
si Delta<0
| renvoyer [ ]
si Delta=0
| renvoyer $[-\frac{b}{2a}]$
si Delta>0
| renvoyer $[-\frac{b}{2a} + \sqrt{\frac{\text{Delta}}{2a}}, -\frac{b}{2a} - \sqrt{\frac{\text{Delta}}{2a}}]$.

Lorsqu'il y a un choix exclusif entre deux blocs d'instructions à exécuter, on peut également utiliser « sinon » (else)

Schéma. Alternatif(paramètres)
si condition
| faire...
sinon
| faire...

L'exemple suivant précise l'algorithme Pair.

Exemple. Pair(a)
si a mod $2 = 0$
| renvoyer vrai
sinon
| renvoyer faux

Lorsqu'il y a un choix exclusif entre plusieurs blocs d'instructions, on peut aussi utiliser « sinon si » (elif dans Python).

Schéma. Multiple(paramètres)
si condition 1
| faire...
sinon si condition 2
| faire...
sinon si condition 3
| faire...
...

1.3.3 Boucles itératives

1.3.3.1 Boucle « pour » (*for*)

Sert lorsqu'on doit répéter un nombre de fois déterminé la même tâche.

Schéma. For(paramètres)
pour i de (valeur de départ) à (valeur d'arrivée)
| faire...

La variable i est appelée **compteur**.

A chaque passage dans la boucle, la valeur du compteur est automatiquement augmentée de 1.

Insistons sur le fait que nous nous **interdisons** de modifier la valeur de i par une affectation [2]. Si nous permettions de modifier i à l'intérieur de la boucle, il deviendrait très difficile de prédire le comportement de l'algorithme.

Par contre, dans les instructions, il est possible de faire intervenir la valeur de i. Lors du premier passage dans la boucle, i aura la valeur 1, puis au deuxième passage, elle aura la valeur 2, et ainsi de suite jusqu'à n.

Enfin, notons que :

— le nom du compteur de boucle importe peu, c'est-à-dire qu'il n'est pas obligatoire de l'appeler i ;

— le compteur de boucle peut prendre des valeurs qui sont dans des ensembles variés, entiers ou non, du moment qu'ils sont finis.

Exemple. Somme(n)
$s \leftarrow 0$
pour i de 1 à n
| $s \leftarrow s + i$
renvoyer s

1.3.3.2 Boucle « tant que » (*while*)

Sert à exécuter une commande tant qu'une condition est vraie.

Attention !!!.

- *Avant la boucle, prévoir une condition initiale.*
- *Éviter la boucle infinie, et pour cela, vérifier que la boucle s'arrête à un moment de l'itération.*

2. D'ailleurs, cette interdiction est mise en place dans les langages de programmation qui se soucient de la rigueur des programmes.

Schéma. While(paramètres)
condition initiale
tant que condition vraie
| faire...

Exemple. Somme2(n)
$s \leftarrow 0$
$i \leftarrow 0$
tant que $i \leq n$
| $s \leftarrow s + i$
| $i \leftarrow i + 1$
renvoyer s

L'algorithme suivant prend en entrée deux entiers n, b strictements positifs et renvoyer $(q, r) \in \mathbb{N}^2$ tel que $n = bq + r$ et $r \in [\![0, b-1]\!]$.

Exemple. Division_euclidienne(n, b)
$r \leftarrow n$
$q \leftarrow 0$
tant que $r \geq b$
| $r \leftarrow r - b$
| $q \leftarrow q + 1$
renvoyer (q, r)

Exemple à éviter. $n \leftarrow 0$
tant que $n \bmod 2 = 0$
| $n \leftarrow n + 2$
renvoyer n.

1.4 Récursivité

1.4.1 Définitions

Définition. *On appelle récursive toute fonction ou procédure qui s'appelle elle-même.*

Attention !!!. *Comme dans le cas d'une boucle, il faut un cas d'arrêt où l'on ne fait pas d'appel récursif.*

Schéma. Récursive(paramètres)
si cas particulier
| faire...
sinon
| ...
| Récursive(paramètres modifiés)
| ...

La fonction suivante décrit $n!$.

Exemple. Factorielle(n)
si $n = 0$
  | renvoyer 1
sinon
  | renvoyer $n\times$ Factorielle($n-1$)

Un algorithme est dit **itératif** s'il n'est pas récursif.

Récursif ou itératif ? Un algorithme est dit itératif s'il n'est pas récursif. Losque l'on a un programme récursif, on peut souvent le réécrire en un programme récursif. Par exemple pour calculer $n!$, on peut aussi utiliser l'algorithme suivant.

Algorithme FactIt(n)
$a = 1$
$k \leftarrow 1$
tant que $k \leq n$:
  | $a \leftarrow ak$
  | $k \leftarrow k + 1$
Renvoyer a.

L'un des avantages des algorithmes récursifs est qu'ils sont parfois plus faciles à programmer, par exemple lorsque l'on utilise des récurrences. En revanche, ils utilisent souvent plus de mémoire que les algorithmes itératifs, ce qui peut amener à dépasser les capacités de l'ordinateur. Par exemple pour calculer 12! en récursif, l'ordinateur va devoir stocker 11!, 10!, ... jusqu'à 1. Alors qu'en itératif, il n'a besoin que de stocker a et k. De manière générale, pour un même problème, les algorithmes itératifs sont souvent plus efficaces que les algorithmes récursifs. Par exemple sous Python, le programme échoue à calculer (10 000)! en récursif, alors qu'il calcule sans problème (100 000)! en itératif.

1.4.2 Calcul des puissances ou la méthode diviser pour régner

En informatique, la méthode « diviser pour régner » consiste à diviser un problème en sous-problèmes, à résoudre les sous-problèmes, puis à résoudre le problème initial en utilisant les solutions des sous-problèmes. Pour calculer efficacement les puissances d'un nombre ou d'un élément de $\mathbb{Z}/N\mathbb{Z}$, pour $N \in \mathbb{N}$, on utilise souvent l'exponentiation rapide, qui est un exemple de cette méthode.

Si x est un élément d'un monoïde M (par exemple $(\mathbb{Z}, .)$ ou $(\mathbb{Z}/N\mathbb{Z}, .)$, pour $N \in \mathbb{N}$) et $n \in \mathbb{N}$, on peut calculer x^n de manière « naïve » par l'une des méthodes suivantes.

Exemple. Calcul de x^n

1) boucle *for*

```
puis1(x,n)
r ← 1
pour i de 1 à n
   | r ← x × r
renvoyer r
```

2) boucle *while*

```
puis2(x,n)
r ← 1
i ← 0
tant que i ⩽ n
   | r ← x × r
   | i ← i + 1
renvoyer r
```

3) récursivité

```
puis3(x,n)
si n = 0
   | renvoyer 1
sinon
   | renvoyer x× puis3(x,n − 1)
```

Dans les trois cas, il faut effectuer $O(n)$ produits pour obtenir le résultat final.

Question. *Peut-on effectuer ce calcul en effectuant moins de produits dans M ?*

Réponse. *Oui, avec **l'algorithme d'exponentiation rapide**.*

Cet algorithme se base sur l'idée suivante : pour $x \in M$ et $n \in \mathbb{N}$,

$$x^n = \begin{cases} (x^{n/2})^2 & \text{si } n \text{ pair;} \\ x(x^{(n-1)/2})^2 & \text{si } n \text{ impair.} \end{cases}$$

Algorithme 1.

```
expo-rapide(x,n)
si n = 0
   | renvoyer 1
sinon
   | si n mod 2 = 0
   |    | r ← expo-rapide(x, n/2)
   |    | renvoyer r²
   | sinon
   |    | r ← expo-rapide(x, (n−1)/2)
   |    | renvoyer x × r².
```

On peut alors se demander combien de calculs de produits sont nécessaires pour calculer x^n avec cette méthode. Commençons par un exemple. Si on veut calculer x^{12}, on effectue :

$$x^{12} = (x^6)^2 = \left(\left(x^3\right)^2\right)^2 = \left(\left(x.x^2\right)^2\right)^2.$$

On a effectué une multiplication et trois élévations au carré, qui peuvent aussi être vues comme des multiplications. On a donc effectué 4 multiplications.

Pour $n \in \mathbb{N}$, on note $C(n)$ le nombre de produits qu'il faut effectuer pour calculer x^n, avec l'algorithme ci-dessus. On a $C(1) = 0$. Soit $n \in \mathbb{N}^*$. Alors il existe $k = k(n) \in \mathbb{N}$ tel que $2^k \leq n < 2^{k+1}$. Montrons, par récurrence sur k, que $C(n) \leq 2k$. On a $k(1) = 0$, donc l'hypothèse est vérifiée. Soit $k \in \mathbb{N}$. On suppose que pour tout $n \in [\![2^k, 2^{k+1}[\![$, on a $C(n) \leq 2k$. Soit $n \in [\![2^{k+1}, 2^{k+2}]\!]$. Si n est pair, on a $(x^n) = (x^{n/2})^2$. On calcule donc $x^{n/2}$, ce qui « coûte » $C(n/2)$ multiplications, puis met $x^{n/2}$, ce qui coûte une multiplication supplémentaire. On a $2^k \leq n/2 < 2^{k+1}$, donc $C(n/2) \leq C(n/2) + 1 \leq 2k + 1 \leq 2(k + 1)$.

Supposons maintenant n impair. Alors $x^n = (x^{(n-1)/2})^2.x$. On effectue donc le calcul de $x^{n/2}$, que l'on met au carré puis que l'on multiplie par x. Au total, on a donc effectué $C((n-1)/2) + 1 + 1$ multiplications. De plus, $2^{k+1} < n < 2^{k+2}$ (n est impair), donc $2^{k+1} - 1 \leq n < 2^{k+2}$ donc $2^k \leq (n-1)/2 < 2^{k+1}$, donc $C((n-1)/2) \leq 2k$, donc $C(n) \leq 2k + 2 = 2(k+1)$. On a donc montré que $C(n) \leq 2k(n)$, pour tout $n \in \mathbb{N}$.

Il reste à exprimer $k(n)$ en fonction de n. On a $2^k \leq n < 2^{k+1}$, donc $\log_2(2^k) \leq \log_2(n) < \log_2(2^{k+1})$ (car $\log_2$ est strictement croissante), donc $k = k(n) \leq \log_2(n) < k(n) + 1$, donc $k(n) = \lfloor \log_2(n) \rfloor$.

Pour $m \in \mathbb{Z}$, on pose $F(m) = \frac{m}{2}$, si m est pair et $F(m) = \frac{m-1}{2}$, si m est impair. On a alors

$$C(n) \leq C(F(n)) + 2. \tag{1.1}$$

En effet, si n est pair, $x^n = x^{n/2}.x^{n/2}$, d'où $C(n) \leq C(F(n)) + 1$ et si n est impair, $x^n = x.x^{F(n)}.x^{F(n)}$, d'où $C(n) \leq C(F(n)) + 2$.

On a en particulier montré

$$C(n) \leq 2\log_2(n) = O(\log_2(n)).$$

En comparaison avec l'algorithme « naïf », il y a donc beaucoup moins de multiplications à effectuer dans M ($O(\log_2(n))$ contre $O(n)$). Si $M = \mathbb{N}$, le coût de calcul (temps de calcul) d'une multiplication $x.y$ augmente lorsque x et y augmentent. Il n'est donc pas évident a priori que le calcul de x^n soit plus rapide avec l'algorithme d'exponentiation rapide qu'avec l'algorithme naïf (car x^m croît avec m, si $x \geq 2$). En revanche, si $M = \mathbb{Z}/N\mathbb{Z}$, pour $N \in \mathbb{Z}$, le coût du calcul de $x.y$ est majoré par une constante (dépendant de N mais indépendante de x et y). L'algorithme d'exponentiation rapide est alors beaucoup plus performant que l'algorithme naïf.

Chapitre 2

Tris

Ce chapitre s'inspire de https ://interstices.info/les-algorithmes-de-tri/.

En mathématiques et en informatique, il est fréquent de devoir ordonner des éléments dans un ordre donné. Cela permet par exemple de calculer la médiane d'une liste de nombres. On s'intessera ici au problème suivant : étant donné une liste $L = [a_0, \ldots, a_{n-1}]$ de nombres entiers, donner la liste $[a'_0, \ldots, a'_{n-1}] = [a_{\sigma(0)}, \ldots, a_{\sigma(n-1)}]$, où $\sigma : [\![0, n-1]\!] \to [\![0, n-1]\!]$ est une bijection, telle que $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$.

Il existe de nombreux algorithmes pour résoudre ce problème. Nous en présentons quelques uns, et étudions leurs avantages et inconvénients.

2.1 Le tri par sélection

Le principe est le suivant : on cherche d'abord l'entier (ou un des entiers) i tel que a_i est le plus petit élément de la liste. On échange ensuite a_0 et a_i. On obtient alors une suite $L_1 = [a_i, b_1, , \ldots, b_{n-1}]$. On réitère ensuite le processus avec $[b_1, \ldots, b_{n-1}]$, et ainsi de suite. On obtient alors l'algorithme récursif suivant :

Algorithme Tri_selection(L)
$n \leftarrow$ long(L)
si $n = 1$:
|renvoyer L :
sinon :
| $j \leftarrow 0$
| pour i de 1 à $n-1$:
|si $L[i] < L[j]$:
|$j \leftarrow i$
| $a \leftarrow L[j]$
| $L[j] \leftarrow L[0]$ #ceci permet de changer la liste en modifiant uniquement sa j-ième entrée
| $L[0] \leftarrow a$
| Renvoyer $L[0]$+Tri_selection($L[1:]$) # où $L[1:]$ signifie $[L[1], \ldots, L[n-1]]$.

Comptons maitenant le nombre d'opérations effectuées par cet algorithme. Pour déterminer le plus petit élément de la liste, on effectue n comparaisons. On effectue au plus n affectations (lorsque on affecte la valeur i à j), puis on échange deux valeurs, ce que l'on fait en trois opérations. Au total, on a donc au plus $2n+3$ opérations. Il faut ensuite trier la liste restante, qui contient $n-1$ éléments. De même, on a besoin d'au plus $2(n-1)+3$ opérations. Au total, on a donc au plus $2n+3+2(n-1)+3+2(n-2)+3+\ldots+3$ opérations, ce qui fait $\frac{n(n-1)}{2}+3n$ opérations, ce qui est asymptotiquement équivalent à $\frac{1}{2}n^2$, et $O(n^2)$.

2.2 Le tri par propagation ou tri à bulles

Le principe de ce tri est le suivant. On parcourt la liste en comparant les paires d'éléments consécutifs, et dès que l'on constate une inversion (c'est à dire si $L[i+1] < L[i]$), on permute les deux éléments correspondants (i.e on remplace $L[i], L[i+1]$ par $L[i+1], L[i]$) dans la liste. À la fin de la première étape, le dernier élément de la liste est plus grand que tous les autres. On réitère alors ce procédé, en l'appliquant aux $n-1$ premiers termes de cette nouvelle liste, etc.

Exercice : écrire un algorithme correspondant à ce tri en pseudo-langage.

Étudions maintenant le nombre d'opérations nécessaires à un tri par cette méthode. À la première étape, il faut faire n comparaisons, et au plus n échanges. Chaque échange compte comme trois opérations, car il faut d'abord stocker la première des valeurs que l'on échange dans une variable intermédiaire, puis effectuer deux nouvelles affectations. Il y a donc $\frac{n(n-1)}{2}$ comparaisons, et au plus $\frac{n(n-1)}{2}$ échanges. En tout, on obtient une complexité en $O(n^2)$.

2.3 Le tri par insertion

Cette méthode est souvent utilisée pour trier des cartes. Le principe est le suivant. On compare a_0 et a_1 et on créé la liste $M = [a_0, a_1]$ ou $M = [a_1, a_0]$ en fonction du résultat. On insère ensuite a_2 dans la liste M, en fonction des comparaisons avec $M[0]$ et $M[1]$. On a donc une liste M de longueur 3. On réitère ce processus jusqu'à obtenir une liste M triée de longueur n.

On peut l'implémenter de la façon suivante :

Algorithme Tri_insertion(L)
$n = \text{long}(L)$
pour i de 1 à $n-1$:
| tmp $\leftarrow L[i]$ # tmp pour « temporaire »
| $k \leftarrow i$
| tant que $L[k-1] >$ tmp et $k > 0$:
| | $L[k] \leftarrow L[k-1]$
| | $k \leftarrow k-1$
| $L[k] \leftarrow$ tmp
Renvoyer L.

La variable tmp est la valeur qu'on va « placer » en l'insérant dans la liste $[L[0], \ldots, L[i]]$, qui est déjà triée. Dans la boucle « tant que », on compare $L[k-1]$ et tmp. Si $L[k-1] >$ tmp, alors on déplace tmp à gauche de $L[k-1]$ et on recommence avec $L[k-2]$. Comme la liste $[L[0], \ldots, L[i]]$ est déjà triée, si jamais on trouve que $L[k-1] \leq$ tmp, il n'y a pas besoin de comparer tmp et les $L[j]$, pour $j < k-1$.

On a alors les résultats suivants (admis) :

1. le cas optimal est lorsque la liste est déjà triée, dans ce cas il y a $n-1$ comparaisons à faire,
2. en « moyenne », le nombre de comparaison est de l'ordre de $n^2/4$ affectations et comparaisons, (en moyenne signifie qu'on fait la moyenne sur tous les $[a_{\sigma(0)}, \ldots, a_{\sigma(n-1)}]$, où $\sigma \in \mathfrak{S}_{n-1}$),
3. dans le pire des cas (lorsque le tableau est trié à l'envers, il y a de l'ordre de $n^2/2$ affectations et comparaisons.

2.4 Le tri fusion

C'est une méthode du type « diviser pour régner ». Le principe est le suivant. Si la liste n'a qu'un élément, il n'y a rien à faire. Sinon, on sépare la liste en deux parties de longueurs égales ou différentes de 1 (en fonction de la parité de la longueur de la liste), et on trie ces deux parties. On fusionne ces deux listes triées.

Pour la fusion, on procède comme suit. Notons L_1 et L_2 les deux listes que l'on veut fusionner et ℓ_1, ℓ_2 leurs longueurs. Soient i_1 et i_2 des variables qu'on initialise à 0. On créé une liste vide L. Tant que $i_1 < \ell_1$ et $i_2 < \ell_2$, on compare le terme d'indice i_1 de L_1 et le

terme d'indice i_2 de L_2 et on note $j \in \{1,2\}$ la liste dont cet élément est issu. On ajoute alors cet élément à L et on augmente i_j de 1. Lorsque $i_1 = \ell_1 - 1$ ou $i_2 = \ell_2 - 1$, on ajoute tous les termes restants à L.

Comptons alors le nombre d'affectations et de comparaisons utilisée pour ce tri. Si $n \in \mathbb{N}$, on note $C(n)$ le nombre de comparaisons et d'affectations maximal que l'on effectue pour trier une liste de longueur n avec cette méthode. Pour trier une liste de longueur au plus n, il faut trier une liste de longueur $\lfloor \frac{n}{2} \rfloor$, une liste de longueur $\lceil \frac{n}{2} \rceil$, et fusionner ces deux listes. L'algorithme que l'on a donné pour fusionner les listes utilise au plus $\lceil \frac{n}{2} \rceil$ comparaisons et de même pour les affectations. On a donc

$$C(n) \leq C(\lceil \frac{n}{2} \rceil) + C(\lfloor \frac{n}{2} \rfloor) + 2\lceil \frac{n}{2} \rceil \leq 2C(\lceil \frac{n}{2} \rceil) + 2\lceil \frac{n}{2} \rceil. \tag{2.1}$$

Soient maintenant $n \in \mathbb{N}$ et $\ell = \log_2(n)$. On a alors $n \in [\![2^\ell, 2^{\ell+1} - 1]\!]$ et $C(n) \leq C(2^{\ell+1})$. En appliquant (2.1), on a $C(2^{\ell+1}) \leq 2C(2^\ell) + 2.2^{\ell+1}$, donc

$$\frac{C(2^{\ell+1})}{2^{\ell+1}} \leq \frac{C(2^\ell)}{2^\ell} + 2,$$

et par récurrence, $\frac{C(2^{\ell+1})}{2^{\ell+1}} \leq 2\ell$ (car $C(1) = 0$), donc $C(2^{\ell+1}) \leq 2^{\ell+1}\ell$. On a donc montré :

$$C(n) \leq 2n\lceil \log_2(n) \rceil.$$

On obtient donc un nombre de comparaisons en $O(n \log_2(n))$, ce qui est beaucoup mieux que les méthodes que l'on a vues jusqu'ici.

2.5 Le tri rapide ou quicksort

Cette méthode, l'une des plus utilisées actuellement a été inventée par C.A.R Hoare en 1961. C'est une méthode du type « diviser pour régner ». L'idée est la suivante. On choisit aléatoirement un élément de la liste, qu'on appelle le pivot. On partitionne la liste en deux parties, les éléments plus petits que le pivot, que l'on place à gauche du pivot et les éléments plus grands que le pivot, que l'on place à droite. Le pivot est alors à sa place définitive. On recommence alors le procédé avec la liste des éléments plus petits que le pivot et avec la liste des éléments plus grands que le pivot. Avec cet algorithme, on obtient en moyenne un tri en $O(n \log(n))$ et dans le pire des cas un $O(n^2)$. En pratique, c'est l'algorithme le plus efficace connu.

On peut par exemple utiliser l'algorithme suivant (voir TD).

Algorithme Partition(L) :
$n = \text{long}(L)$
pivot $\leftarrow L[n-1]$
$k \leftarrow 0$
pour i de 0 à $n-1$:
    | si $L[i] <$ pivot :
        | $L[k], L[i] \leftarrow L[i], L[k]$
        | $k \leftarrow k+1$
$L[k], L[n-1] \leftarrow L[n-1], L[k]$
Renvoyer L.

Chapitre 3

Approximations de nombres réels

Dans ce chapitre, nous voyons comment obtenir des valeurs approchées de certains nombres réels. Nous étudions notamment le calcul de limite de suites récurrentes, la recherche de zéros de fonctions, le calcul de certaines fonctions développables en séries entières. Ce chapitre se base sur le livre *Algorithmes* de exo7 (http ://exo7.emath.fr/cours/livre-algorithmes.pdf) et sur [Dem06] (voir aussi [Fil13]).

3.1 Nombres réels et flottants

En mathématiques, un nombre réel est un nombre qui peut avoir une infinité de chiffres après la virgule dans son écriture décimale. Il n'est pas possible de stocker une infinité d'informations dans un ordinateur. Pour représenter les réels, on utilise donc les flottants, qui sont en quelque sorte une approximation de ces derniers. Un flottant est un nombre de la forme $\epsilon a e\ n$, où $\epsilon \in \{-1, 1\}$, a est un nombre décimal de $[1, 10[$ qui a 16 chiffres après la virgule et n est un entier (positif ou négatif). Le nombre ϵa s'appelle la *mantisse* et n l'*exposant* et la notation $e\ n$ signifie 10^n. Par exemple $12,34567891011121314$ est codé par

$$1.2345678910111213e\ 1.$$

Si on effectue $1.0+10^{-16}$, on obtient 1. En revanche, si on effectue $0.1+10^{-16}$, on obtient $1.0000000000000001\ e-1$. Ce phénomène rend par exemple l'addition non associative pour les flottants. Ces erreurs sont particulièrement problématiques lorsque l'on fait la différence de deux « grands » nombres proches : on perd beaucoup en précision. Par exemple si on effectue $(10^{15} + \pi) - 10^{15}$, on obtient 3.125. En fait, comme la mantisse ne contient que 17 chiffres, $10^{15} + \pi = 10^{15} + 3.14$ pour Python. De même, si l'on fait la somme de deux flottants a et b, avec $|a| \gg |b|$ (par exemple a/b de l'ordre de 10^{16}), on perd beaucoup de précision sur b.

Losque l'on calcule les sommes partielles d'une série $\sum u_n$, avec $u_n \to 0$, si on calcule $\sum_{k=0}^{n+1} u_k$ en effectuant $(\sum_{k=0}^{n} u_k) + u_{n+1}$, on se trouve souvent dans cette situation. Pour pallier ce problème, on peut calculer la somme en commençant par la fin. Par exemple, si on souhaite calculer $\sum_{k=1}^{n} \frac{1}{k^2}$, pour $n \in \mathbb{N}^*$, commencer par la fin permet de sommer des termes

du même ordre de grandeur. En pratique, comme Python est assez précis (la mantisse contient 17 chiffres), cela ne fait pas une grande différence. Si on calcule $\sum_{k=1}^{10^8} \frac{1}{k^2}$ par ces deux méthodes (en faisant $1+1/4+1/9+\ldots$ et $1/(10^8)^2+1/(10^8-1)^2+1/(10^8-2)^2+\ldots$), on obtient une différence de l'ordre de 10^{-8}.

3.2 Recherche de zéros de fonctions

3.2.1 Recherche de zéros par dichotomie

Soient $I=[a,b]$ un segment de $\mathbb{R}$ et $f:[a,b]\to\mathbb{R}$ une fonction continue. On suppose que $f(a)f(b)<0$ (c'est à dire que $f(a)$ et $f(b)$ sont de signes contraires). D'après le théorème des valeurs intermédiaires, il existe $\ell\in[a,b]$ tel que $f(\ell)=0$. On veut trouver une valeur approchée d'un tel ℓ. Une méthode très élémentaire pour résoudre ce problème est de procéder par dichotomie. Comme pour tout $x\in[a,b]$, on a $|x-\ell|\leq b-a$. On connaît donc une valeur approchée à $b-a$ près d'un zéro de f (notons qu'il n'y a pas nécessairement unicité d'un tel zéro). On calcule $f(\frac{a+b}{2})$. Si $f(a)f(\frac{a+b}{2})<0$, alors on sait qu'il existe $\ell\in[a,\frac{a+b}{2}]$ tel que $f(\ell)=0$. On pose alors $a_1=a$ et $b_1=\frac{a+b}{2}$. Sinon, $f(b)f(\frac{a+b}{2})<0$, donc il existe $\ell\in[\frac{a+b}{2},b]$ tel que $f(\ell)=0$. On pose alors $a_1=\frac{ab}{2}$ et $b_1=b$. En réitérant ce procédé avec $[a,\frac{a+b}{2}]$ ou avec $[\frac{a+b}{2},b]$, on obtient une valeur approchée à $\frac{b-a}{2}$ d'un tel zero de f. En réitérant ce procédé n fois, on obtient donc deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ (en posant $a_0=a$ et $b_0=b$). La suite (a_n) est croissante, (b_n) est décroissante, et $b_n-a_n\to 0$ (car $b_{n+1}-a_{n+1}=\frac{b_n-a_n}{2}=\frac{b-a}{2^{n+1}}$, pour tout $n\in\mathbb{N}$). Les suites (a_n) et (b_n) sont donc adjacentes et elles convergent vers une limite commune ℓ. Comme $f(a_n)\leq 0$ et $f(b_n)>0$, pour tout $n\in\mathbb{N}$, on a $f(\ell)\leq 0$ et $f(\ell)\geq 0$, donc $f(\ell)=0$. À chaque étape, il faut calculer la valeur de f en un point (ou du moins déterminer son signe), donc il faut calculer la valeur de f en $n+1$ points à la n-ième étape (pour la première étape, il faut calculer la valeur de f en deux points).

Exemple : On cherche à calculer une valeur approchée de $\sqrt{10}$. On définit $f:\mathbb{R}\to\mathbb{R}$ par $f(x)=x^2-10$, pour $x\in\mathbb{R}$. On pose $a=3$ et $b=4$. En itérant 10 fois, on obtient les valeurs suivantes pour (a_n,b_n) : $(3,4),(3,3.5),(3,3.25),(3.125,3.25),(3.125,3.1875)$,
$(3.15625,3.1875),(3.15625,3.171875),(3.15625,3.1640625),(3.16015625,3.1640625)$,
$(3.162109375,3.1640625),(3.162109375,3.1630859375)$. On sait donc que $\sqrt{10}\in[3.162,3.163]$.

3.2.2 La recherche de zéros par la méthode de la sécante

Soit $f:[a,b]\to\mathbb{R}$ une fonction convexe et continue, où $a,b\in\mathbb{R}$, $a\leq b$. On suppose que $f(a)\leq 0$ et $f(b)>0$ et on cherche $x\in[a,b]$ tel que $f(x)=0$. L'idée de la méthode de la sécante est la suivante. On approche la fonction f sur le segment $[a,b]$ par la fonction affine valant $f(a)$ en a et $f(b)$ en b et on regarde où la fonction approchée s'annule. On trace donc le segment reliant $A=(a,f(a))$ et $B=(b,f(b))$. On considère alors l'abscisse $a_1\in[a,b]$ du point d'intersection entre $[A,B]$ et l'axe des abscisses (comme $f(a)\leq 0$ et

$f(b) > 0$, cette intersection est bien un singleton). Comme f est convexe, le graphe de f est situé en dessous du segment $[A, B]$. En particulier, $f(a_1)$ est en dessous de 0, donc $f(a_1) \leq 0$ et $f(b) > 0$. On réitère alors ce procédé, et on obtient une suite $(a_n)_{n\in\mathbb{N}}$ (en posant $a_0 = a$).

Soit $n \in \mathbb{N}$. Exprimons a_{n+1} en fonction de a_n. La droite reliant $A_n = (a_n, f(a_n))$ et B a pour équation $y = (x - a_n)\frac{f(b)-f(a_n)}{b-a_n} + f(a_n)$. On a donc $0 = (a_{n+1} - a_n)\frac{f(b)-f(a_n)}{b-a_n} + a_n$ (en effet, $y = 0$ lorsque $x = a_{n+1}$). Comme $f(a_n) \leq 0$ et $f(b) > 0$, $f(b) - f(a_n)$ est strictement positif et en particulier, il est non nul. On a donc :

$$a_{n+1} = a_n + \frac{f(a_n)(a_n - b)}{f(b) - f(a_n)}. \tag{3.1}$$

Proposition. *Soit $f : [a, b] \to \mathbb{R}$ une fonction convexe et continue, où $a, b \in \mathbb{R}$, $a \leq b$. On suppose que $f(a) \leq 0$ et $f(b) > 0$ Soit (a_n) la suite définie par $a_0 = a$ et*

$$a_{n+1} = a_n + \frac{f(a_n)(a_n - b)}{f(b) - f(a_n)},$$

pour $n \in \mathbb{N}$. Alors (a_n) est croissante et converge vers un zéro de f.

Démonstration. Si $n \in \mathbb{N}$, on a déjà vu que $f(a_n) \leq 0$ et que a_{n+1} est l'abscisse du point d'intersection entre le segment $[(a_n, f(a_n), B]$ et l'axe des abscisses, donc $a_n \leq a_{n+1} \leq b$. La suite (a_n) est donc croissante et majorée et elle converge donc vers une limite ℓ. Par continuité de f, $f(a_n) \underset{n\to+\infty}{\longrightarrow} f(\ell)$. De plus, $f(a_n) \leq 0$, donc $f(\ell) \leq 0$. En particulier, $\ell < b$ et $f(b) - f(\ell) \geq f(b) > 0$. En passant à la limite dans (3.1), on obtient

$$\ell = \ell + f(\ell)\frac{\ell - b}{f(b) - f(\ell)},$$

donc $f(\ell)\frac{\ell-b}{f(b)-f(\ell)} = 0$, donc $f(\ell) = 0$. □

Par la proposition précédente, on a donc $a_n \leq \ell \leq b$, avec (a_n) tendant vers ℓ. Comme b est fixe, cela ne nous fournit pas de majoration satisfaisante de l'erreur $|a_n - \ell|$. Dans le cas où f est dérivable et telle que $f'(\ell) \neq 0$, on peut obtenir une majoration de cette erreur à l'aide du théorème des accroissements finis.

Proposition. *Soit I un segment de $\mathbb{R}$, $g : I \to \mathbb{R}$ et $\ell \in I$. On suppose que $g(\ell) = 0$ et qu'il existe $m \in \mathbb{R}_+^*$ tel que $|g'(x)| \geq m$ pour tout $x \in I$. Alors pour tout $x \in I$, on a $|x - \ell| \leq \frac{|g(x)|}{m}$.*

Démonstration. D'après le théorème des accroissements finis, si $x \in I\backslash\{\ell\}$, il existe $c \in]x, \ell[$ tel que $\frac{g(x)-f(\ell)}{x-\ell} = g'(c) = \frac{g(x)}{x-\ell}$. On a donc $|\frac{g(x)}{x-\ell}| \geq m$ donc $|x - \ell| \leq \frac{g(x)}{m}$. □

Rappel : Si I est un intervalle de $\mathbb{R}$ et $f : I \to \mathbb{R}$ est dérivable, alors f est convexe si et seulement si f' est croissante.

Lorsque f est de classe $\mathcal{C}^1$ et telle que $f'(\ell) \neq 0$, on peut utiliser ce résultat pour majorer l'erreur, puisque $f(a_n) \to 0$. Comme on a $f(a_n) \to 0$, on peut utiliser ce résultat général pour majorer l'erreur. Dans le cas où $f'(a) > 0$, on a $f'(x) \geq f'(a_n) \geq f'(a) > 0$ pour tout $x \in [a_n, b]$, pour $n \in \mathbb{N}$ (car f' est croissante comme f est convexe), donc $|a_n - \ell| \leq \frac{f(a_n)}{f'(a_n)}$.

Exemple : On cherche à obtenir une valeur approchée de $\sqrt{10}$. On pose toujours $f : \mathbb{R} \to \mathbb{R}$, $x \mapsto x^2 - 10$. On part de $a = 3$ et $b = 4$, donc $a_0 = 3$. On a alors $a_{n+1} = a_n + \frac{(a_n^2-10)(a_n-4)}{16-a_n^2} = a_n + \frac{10-a_n}{a_n+4}$, pour $n \in \mathbb{N}$. En itérant 10 fois, on obtient : $3, 3.14285714285714, 3.16, 3.16201117318436, 3.16224648985959, 3.16227401437595,$
$3.16227723374491, 3.16227761029256, 3.16227765433475, 3.16227765948606, 3.16227766008857.$
On obtient $f(3.16227766008857) \simeq 5.10^{-10}$. De plus, si $x \in \mathbb{R}$, $f'(x) = 2x$. En appliquant la proposition 3.2.2 avec $M = 6$, on obtient que 3.16227766008857 est une valeur approchée à 10^{-10} près de $\sqrt{10}$.

Remarque 3.1. *L'hypothèse de convexité que l'on a faite sur f peut-être largement affaiblie. En fait, on peut supposer que f est une fonction de classe $\mathcal{C}^2$ sur un intervalle I de $\mathbb{R}$ qui admet un zéro ℓ à l'intérieur de I et telle que $f'(\ell) \neq 0$. Alors en prenant a et b suffisamment proches de x, la suite définie par* (3.1) *converge vers ℓ (voir [Dem06] pour un énoncé plus précis).*

3.2.3 La méthode de Newton

Soit f une fonction de classe $\mathcal{C}^2$ sur un segment I. On suppose qu'il existe $\ell \in \mathring{I}$ (l'intérieur de I) tel que $f(\ell) = 0$ et que $f'(y) \neq 0$ pour tout $y \in I$. L'idée de la méthode de Newton est la suivante. On suppose que l'on connaît une valeur approchée grossière a de ℓ. On trace la tangente à f en a et on considère l'abscisse de l'intersection de cette tangente avec l'axe des abscisses. On obtient alors une valeur a_1. Lorsque a est pris suffisamment proche de ℓ, on peut réitérer ce procédé autant de fois qu'on le souhaite et on obtient une suite (a_n) convergeant vers ℓ.

Soit $n \in \mathbb{N}$. Supposons que l'on a défini a_n et que $a_n \in I$. Alors l'équation de la tangente à f en a_n est $y = f'(a_n)(x - a_n) + f(a_n)$. On a donc

$$a_{n+1} = a_n - \frac{f(a_n)}{f'(a_n)},$$

(car on doit avoir $y = 0$ pour $x = a_{n+1}$). On a alors le résultat suivant :

Théorème 3.2. *(voir [Dem06, IV 2.4]) Soit f une fonction de classe $\mathcal{C}^2$ sur un intervalle I. On suppose qu'il existe $\ell \in I$ tel que $f(\ell) = 0$ et on suppose que I est de la forme $[\ell - r, \ell + r]$, où $r \in \mathbb{R}_+^*$. On suppose de plus que $f'(x) \neq 0$, pour tout $x \in I$. Soit $M = \max_{x \in I} |\frac{f''(x)}{f'(x)}|$ et $h = \min(r, \frac{1}{M})$. Alors pour tout $a \in [\ell - h, \ell + h]$, la suite (a_n) est bien définie et on a, pour tout $n \in \mathbb{N}$,*

$$|a_n - \ell| \leq \frac{1}{M}(M|a_0 - \ell|)^{2^n}.$$

Remarque 3.3. *En pratique, il est souvent difficile de déterminer si l'on est dans l'intervalle* $[\ell-h, \ell+h]$. *Tout d'abord on ne connaît pas* ℓ, *et de plus, il faut réussir à majorer* $|\frac{f''}{f'}|$, *ce qui n'est pas toujours facile. Dans certains cas simples, on montrera que* (a_n) *converge pour* a_0 *appartenant à un certain intervalle explicite (sans utiliser ce théorème). Ce théorème est alors rassurant sur la vitesse de convergence de la suite, puisque on est sûr qu'à partir d'un moment on se retrouvera dans l'intervalle* $[\ell - h, \ell + h]$.

Exemples : Appliquons la méthode de Newton à $f(x) = x^2 - 10$, pour $x \in \mathbb{R}$. On a donc $a_{n+1} = \frac{1}{2}(a_n + \frac{10}{a_n})$, pour $n \in \mathbb{N}$. En prenant $a_0 = 3$ et en itérant 10 fois, on obtient :
$3, 3.16666666666667, 3.16228070175439, 3.16227766016984, 3.16227766016838,$
$3.16227766016838, 3.16227766016838, 3.16227766016838, 3.16227766016838,$
$3.16227766016838, 3.16227766016838.$

On constate qu'à partir de la cinquième itération, la suite ne varie plus. On obtient $f(3.16227766016838) \simeq 3.10^{-15}$, donc 3.16227766016838 est une valeur approchée à 10^{-15} près de $\sqrt{10}$.

Remarquons que si $x \in \mathbb{R}_+^*$, la méthode de Newton pour calculer $\sqrt{x}$ consiste à partir d'un $a_0 > 0$ et à définir (a_n) par $a_{n+1} = \frac{1}{2}(a_n + \frac{a_n}{x})$. Cette suite, appelée *suite de Héron* est connue depuis l'antiquité pour calculer des racines carrées.

Prenons maintenant $f : \mathbb{R} \to \mathbb{R}$ définie par $f(x) = x^3 - 4x + 1$, pour tout $x \in \mathbb{R}$. À l'aide d'une étude de fonctions, on constate que f admet exactement 3 racines réelles. Notons les ℓ_1, ℓ_2 et ℓ_3, avec $\ell_1 < \ell_2 < \ell_3$. Grâce à un tableau de variations, on peut montrer que $\ell_1 \in I_1 :=]-2.5; -2[$, $\ell_2 \in I_2 :=]0; 0.5[$ et $\ell_3 \in I_3 :=]1.5; 2[$. Soit $\varphi : \mathbb{R}\backslash\{\pm\sqrt{4/3}\} \to \mathbb{R}$ définie par $x \mapsto x - \frac{f(x)}{f'(x)}$. On a $\varphi(x) = \frac{2x^3-1}{3x^2-4}$, pour tout x. En partant de $a_0 = -2$, on obtient les valeurs suivantes, en itérant jusqu'à 5 fois : $-2.125, -2.11497545008183, -2.11490754458310,$
$-2.11490754147676, -2.11490754147676.$

En partant de $a_0 = 0$, on obtient $0.25, 0.254098360655738, 0.254101688362835,$
$0.254101688365052, 0.254101688365052.$

En partant de $a_0 = 2$: $1.875, 1.86097852028640, 1.86080587916044,$
$1.86080585311170, 1.86080585311170.$

Si on calcule ensuite f aux dernières valeurs que que l'on a obtenues pour ces trois valeurs de départ, on obtient des valeurs de l'ordre de 10^{-14} et donc on a bien des valeurs approchées des racines de f par exemple grâce à la proposition 3.2.2 (il faudrait minorer f' sur I_1, I_2 et I_3 pour avoir plus de précisions).

En pratique, il faut d'abord connaître des encadrements des zéros que l'on cherche : pour appliquer le théorème 3.2, il faut que la première valeur que l'on prend pour a_0 soit assez proche de ℓ, sinon on n'est pas sûr que la suite (a_n) va converger vers le bon zéro (ni même qu'elle va converger). On peut par exemple commencer par obtenir un encadrement de ℓ en étudiant la fonction et en procédant par dichotomie puis appliquer la méthode de Newton, qui est beaucoup plus rapide que la dichotomie, comme nous le verrons.

3.2.4 Comparaison des différentes méthodes

L'avantage de la première méthode est qu'elle est très élémentaire et qu'elle converge vers un zéro, pourvu que l'on ait $f(a) \leq 0$ et $f(b) \geq 0$. En revanche, elle est assez lente : pour obtenir une décimale supplémentaire, il faut entre trois et quatre itérations à chaque fois (cf TD).

L'avantage de la méthode de Newton est qu'elle est très rapide : le nombre de décimales correctes double quasiment à chaque nouvelle itération (asymptotiquement, cf TD). Par contre il faut connaître un encadrement du zéro que l'on veut calculer, sinon il n'y a pas nécessairement convergence. De plus, il faut être capable de calculer numériquement la dérivée de f, ce qui n'est pas toujours facile.

La méthode de la sécante est plus lente que la méthode de Newton, même si elle reste assez rapide (le nombre de décimales correctes est asymptotiquement approximativement multiplié par $\frac{1+\sqrt{5}}{2} \simeq 1.6$ à chaque itération, cf [Dem06, IV 2.4]). Par contre, il n'y a pas besoin de calculer f'.

3.3 Calculs approchés de fonctions

De nombreuses fonctions usuelles (par exemple $\exp, \sin, \cos, \arctan, x \mapsto \ln(1+x), \ldots$) sont développables en séries entières au voisinage de 0, c'est à dire qu'il existe $r > 0$ et une suite $(a_n) \in \mathbb{R}^{\mathbb{N}}$ tels que pour tout $x \in]-r, r[$, $\Sigma a_n x^n$ converge absolument (c'est à dire que $\Sigma |a_n||x|^n$ converge) et la fonction vaut $\sum_{n=0}^{+\infty} a_n x^n$. Par exemple exp, sin et cos sont développables en séries entières sur $\mathbb{R}$ et on a, pour tout $x \in \mathbb{R}$:

$$\exp(x) = \sum_{n=0}^{+\infty} \frac{x^n}{n!}, \ \sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots = \sum_{n=0}^{+\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

et

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots = \sum_{n=0}^{+\infty} (-1)^n \frac{x^{2n}}{(2n)!}.$$

Si $f(x) = \sum_{n=0}^{+\infty} a_n x^n$, alors on peut essayer de calculer une valeur approchée de $f(x)$ en calculant $\sum_{n=0}^{N} a_n x^n$, pour N assez grand. Il faut avoir des informations sur l'erreur

$$r_N(x) := \sum_{n=N+1}^{+\infty} a_n x^n = f(x) - \sum_{n=0}^{N} a_n x^n,$$

pour savoir quel N choisir.

3.3.1 Un exemple simple, le cas des séries alternées

Si $x \in \mathbb{R}$ est tel que $(a_n x^n)$ est de signe alterné et $(|a_n x^n|)$ est décroissante et tends vers 0, alors on peut facilement majorer $r_N(x)$ en fonction de $|a_N x^N|$.

Proposition 3.4. *Soit $(a_n) \in \mathbb{R}^{\mathbb{N}}$. Supposons que $((-1)^n a_n x^n)$ est positive et que $(|a_n x^n|)$ est décroissante et tends vers 0. Alors pour tout $N \in \mathbb{N}$, on a*

$$\sum_{n=0}^{2N+1} a_n x^n \leq f(x) \leq \sum_{n=0}^{2N} a_n x^n \text{ et } |r_N(x)| \leq |a_N x^N|.$$

Démonstration. Pour $N \in \mathbb{N}$, on pose $S_N(x) = \sum_{n=0}^{N} a_n x^n$. Si $n \in \mathbb{N}$, on a alors $S_{2n+2}(x) = S_{2n}(x) + a_{2n+1}x^{2n+1} + a_{2n+2}x^{2n+2}$. Par hypothèse, $a_{2n+1}x^{2n+1} + a_{2n+2}x^{2n} = -|a_{2n+1}x^{2n+1}| + |a_{2n+2}x^{2n+2}| \leq 0$. La suite $(S_{2n}(x))_{n\in\mathbb{N}}$ est donc décroissante. De même, $(S_{2n+1}(x))_{n\in\mathbb{N}}$ est croissante. De plus, $S_{2n+1}(x) - S_{2n}(x) = a_{2n+1}x^{2n+1} \underset{n\to+\infty}{\longrightarrow} 0$. On en déduit que les suites $(S_{2n}(x))_{n\in\mathbb{N}}$ et $(S_{2n+1}(x))_{n\in\mathbb{N}}$ sont adjacentes. Si $N \in \mathbb{N}$, on a alors

$$S_{2N+1}(x) \leq \lim_{n\to+\infty} S_{2n+1}(x) = f(x) = \lim_{n\to+\infty} S_{2n}(x) \leq S_{2N}(x). \tag{3.2}$$

Soit $n \in \mathbb{N}$. Si n est pair, on a

$$S_{n+1}(x) = S_n(x) + a_{n+1}x^{n+1} \leq f(x) \leq S_n(x),$$

donc $a_{n+1}x^{n+1} \leq f(x) - S_n(x) \leq 0$ et

$$|r_{n+1}(x)| = |f(x) - S_n(x)| \leq |a_{n+1}x^{n+1}|.$$

Si n est impair, on a $S_n(x) \leq f(x) \leq S_{2n+1}(x) = S_n(x) + a_{n+1}x^{n+1}$, donc $0 \leq f(x) - S_n(x) \leq a_{n+1}x^{n+1}$ et

$$|r_{n+1}(x)| = |f(x) - S_n(x)| \leq |a_{n+1}x^{n+1}|,$$

d'où le résultat.

□

Lemme 3.5. *(exercice) Soit $(u_n) \in (\mathbb{R}_+^*)^{\mathbb{N}}$. On suppose que $\frac{u_{n+1}}{u_n} \leq 1$, pour tout $n \in \mathbb{N}$. Alors (u_n) est décroissante.*

Appliquons maintenant cette proposition sur des cas simples.

Exemples :

Calcul de $\frac{1}{1+x}$, pour $x \in [0,1[$ On a $\frac{1}{1+x} = \sum_{n=0}^{+\infty}(-1)^n x^n$, pour $x \in [0,1[$. Par exemple, $\frac{1}{1+\frac{1}{10}} = \sum_{n=0}^{+\infty}(-\frac{1}{10})^n = 1 - 0,1 + 0,01 - 0,001 + \ldots$.

Lemme 3.6. *Soit $x \in \mathbb{R}_+$. Alors $(x_n)_{n\in\mathbb{N}}$ est décroissante à partir d'un certain rang, et $\frac{x^n}{n!} \underset{n\to+\infty}{\longrightarrow} 0$.*

Démonstration. Supposons $x \neq 0$. Soit $n \in \mathbb{N}$. Alors $\frac{x^{n+1}/(n+1)!}{x^n/n!} = \frac{x}{n+1} \underset{n\to+\infty}{\longrightarrow} 0$. On en déduit qu'il existe $N \in \mathbb{N}$ tel que $\frac{x^{n+1}/(n+1)!}{x^n/n!} \leq \frac{1}{2}$, pour tout $n \geq N$. Soit $i \in \mathbb{N}$. Alors $u_{N+i} = \frac{u_{N+i}}{u_{N+i-1}}\frac{u_{N+i-1}}{u_{N+i-2}}\ldots\frac{u_{N+1}}{u_N}u_N \leq \frac{1}{2^i}u_N \underset{i\to+\infty}{\longrightarrow} 0$. □

Calcul de $\exp(x)$ Soit $x \in [-1,0]$. Alors $((-1)^n \frac{x^n}{n!})_{n\in\mathbb{N}}$ est positive et $(|\frac{x^n}{n!}|)_{n\in\mathbb{N}}$ est décroissante. Avec les notations précédentes, on a donc $|r_N(x)| \leq \frac{|x|^N}{N!} \leq \frac{1}{N!}$. On a $20! \geq 10^{18}$, donc en prenant $N = 20$ et en calculant $S_{20}(x)$, on obtient une valeur approchée à 10^{-18} près. On obtient par exemple $e^{-1} \simeq 0.367879441171442$.

Pour calculer $\exp(x)$, pour $x \in]-\infty, -1[$, la méthode précédente ne s'applique pas directement car $(|\frac{x^n}{n!}|)_{n\in\mathbb{N}}$ n'est pas décroissante. En effet, $|x| = |\frac{x^1}{1!}| > |\frac{x^0}{0!}| = 1$. Par contre, d'après le lemme 3.6, $(|\frac{x^n}{n!}|)_{n\in\mathbb{N}}$ est décroissante à partir d'un certain rang dépendant de x (exercice : déterminer ce rang). On peut donc appliquer la proposition précédente à la suite $(\frac{x^n}{n!})_{n\in\mathbb{N}_{\geq k}}$.

Pour calculer $\exp(x)$, pour $x \in \mathbb{R}_+$, on peut alors calculer $\frac{1}{\exp(-x)}$.

Calcul de $\sin(x)$ Soit $x \in \mathbb{R}$. Comme sin est impaire, on peut supposer que $x \geq 0$. La suite $((-1)^n \frac{x^{2n+1}}{(2n+1)!})_{n\in\mathbb{N}}$ est alternée. Par contre, si $|x| \geq 1$, $(\frac{x^{2n+1}}{(2n+1)!})_{n\in\mathbb{N}}$ n'est pas décroissante. En utilisant le lemme 3.5, on peut démontrer que $(\frac{x^{2n+1}}{(2n+1)!})$ est décroissante à partir d'un certain rang, ce qui permet de calculer des valeurs approchées de $\sin(x)$, pour $x \in \mathbb{R}$. Notons également qu'en utilisant le fait que sin est 2π périodique, on peut se ramener au cas où $x \in [0, \pi]$ (mais il faut connaître un bonne approximation de π pour pouvoir calculer $x - k\pi$, pour $k \in \mathbb{Z}$). On peut de même calculer $\cos(x)$, pour $x \in \mathbb{R}$.

Cette méthode s'applique également à arctan (ce qui permet de calculer une valeur approchée de $\pi = 4\arctan(1)$), $x \mapsto \ln(1+x)$ $(x \in]-1, 1[)$,

3.3.2 Utilisation de l'inégalité de Taylor-Young avec reste intégral

Soient $n \in \mathbb{N}$, I un intervalle de $\mathbb{R}$ et $f : I \to \mathbb{R}$ une fonction de classe $\mathcal{C}^{n+1}$. On a alors la **formule de Taylor-Young avec reste intégral :**

$$\forall a, b \in I, f(b) = \sum_{k=0}^{n} \frac{f^{(k)}(b)}{k!}(b-a)^k + \int_a^b \frac{(b-u)^n}{n!} f^{(n+1)}(u)du.$$

On en déduit l'**inégalité de Taylor-Young :** si $M = \max|\{f^{n+1}(u) \mid u \in [a,b]\}$, alors

$$|f(b) - \sum_{k=0}^{n} \frac{f^{(k)}(b)}{k!}(b-a)^k| \leq M\frac{|b-a|^{n+1}}{(n+1)!}.$$

Supposons maintenant que f est une fonction développable en série entière sur un voisinage $]-r, r[$ de 0, où $r > 0$. Soit $(a_n) \in \mathbb{R}^{\mathbb{N}}$ telle que $f(x) = \sum_{n=0}^{+\infty} a_n x^n$, pour tout $x \in]-r, r[$. Si $x \in]-r, r[$ alors en appliquant cette inégalité avec $a = 0$ et $b = x$, on obtient

$$|f(x) - \sum_{k=0}^{n} a_k x^k| = |r_{n+1}(x)| \leq \max_{u\in[0,x]} |f^{(n+1)}(u)| \frac{|x|^{n+1}}{(n+1)!}.$$

Ainsi, si on sait majorer les $f^{(n+1)}$, pour $n \in \mathbb{N}$, on peut en déduire une valeur approchée de f.

Exemples : Supposons que $f = \exp$. On a alors $f^{(n)} = \exp$ pour tout $n \in \mathbb{N}$. Si $x \in \mathbb{R}_+$, $\max_{u\in[0,x]} |\exp(u)| = \exp(x)$ et par l'inégalité de Taylor-Young, on obtient : $0 \leq \exp(x) - \sum_{k=0}^{n} \frac{x^n}{n!} \leq \frac{x^{n+1}e^x}{(n+1)!}$. Il reste à donner une majoration de e^x pour obtenir une valeur approchée de e^x. Si $x \in \mathbb{R}_-$, on obtient $\max_{u\in[0,x]} |\exp(u)| = \exp(0) = 1$ et donc : $|\exp(x) - \sum_{k=0}^{n} \frac{x^n}{n!}| \leq \frac{|x|^{n+1}}{(n+1)!}$.

Cette méthode fonctionne aussi très bien avec cos et sin, puisque les dérivées successives de ces fonctions sont de la forme $\pm\cos$, $\pm\sin$, qui sont des fonctions bornées par 1.

3.4 Calculs approchés d'intégrales

Soit f une fonction continue définie sur un intervalle $[a, b]$ et à valeurs dans $\mathbb{R}$. Une méthode pour calculer numériquement $\int_a^b f$ est de déterminer une primitive F de f est de calculer $F(b) - F(a)$. Cependant, pour de nombreuses fonctions on ne sait pas « expliciter » de primitive de f. En pratique, on peut calculer $\int_a^b f$ sans connaître de primitive de f. On présente différentes méthodes ici. Cette partie se base sur https ://www.maths-france.fr/MathSup/Cours/27-rectangles-trapezes.pdf.

3.4.1 Méthode des rectangles

L'idée est d'approcher la fonction que l'on veut intégrer par des fonctions constantes par morceaux. Si $n \in \mathbb{N}^*$, on subdivise l'intervalle $[a, b]$ en n sous-intervalles, $[a_i, a_{i+1}]$, $i \in [\![0, n-1]\!]$ tous de même longueur $\frac{b-a}{n}$ et on approche f par la fonction g_n constante et égale à $f(a_i)$ sur chaque intervalle $[a_i, a_{i+1}]$. On a alors $\int_a^b g_n \underset{n\to+\infty}{\longrightarrow} \int_a^b f$. Dans cette partie, on détaille ce principe.

Commençons par le cas $n = 1$.

Lemme 3.7. *Soit $f : [a, b] \to \mathbb{R}$ une fonction de classe $\mathcal{C}^1$. Soit $M \in \mathbb{R}_+^*$ tel que $|f'(x)| \leq M$ pour tout $x \in [a, b]$. Alors $|\int_a^b f - (b-a)f(a)| \leq \frac{M(b-a)^2}{2}$.*

Démonstration. On a $(b-a)f(a) = \int_a^b f(a)$. D'après le théorème des accroissements finis, on a $|f(t) - f(a)| \leq (t-a)M$, pour tout $t \in [a, b]$. Ainsi

$$|\int_a^b f - (b-a)f(a)| \leq \int_a^b |f(t) - f(a)|dt \leq \int_a^b M(t-a)dt \leq [\frac{M(t-a)}{2}]_a^b = \frac{M(b-a)^2}{2}.$$

□

Passons maintenant au cas général. Pour $n \in \mathbb{N}^*$, on pose $S_n(f) = \frac{1}{n}\sum_{i=0}^{n-1} f(a + \frac{i}{n}(b-a))$. Soit $g_n : [a, b] \to \mathbb{R}$ la fonction en escaliers définie par $g_n(t) = f(a + i\frac{b-a}{n})$ pour tout $t \in [a + \frac{i}{n}(b-a), a + \frac{i+1}{n}(b-a)[$, pour tout $i \in [\![0, n-1]\!]$. Alors $S_n(f) = \int_a^b g_n$.

Proposition. *Soit $f : [a, b] \to \mathbb{R}$ une fonction de classe $\mathcal{C}^1$. Soit $M \in \mathbb{R}_+^*$ tel que $|f'(x)| \leq M$ pour tout $x \in [a, b]$. Soit $n \in \mathbb{N}^*$. Alors*

$$|\int_a^b f - S_n(f)| \leq \frac{M(b-a)^2}{2n}.$$

Démonstration. Pour $i \in [\![0, n]\!]$, on pose $x_i = a + +\frac{i}{n}(b-a)$. Alors

$$\begin{aligned}
|\int_a^b f - S_n(f)| &= |\sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} (f - f(a_i))\,| \\
&\leq \sum_{i=0}^{n-1} |\int_{x_i}^{x_{i+1}} |f - f(a_i)| \text{ d'après l'inégalité triangulaire,} \\
&\leq \sum_{i=0}^{n-1} \frac{M(x_{i+1}-x_i)^2}{2}, \text{ d'après le lemme 3.7,} \\
&= nM\frac{(b-a)^2}{2n^2} = \frac{M(b-a)^2}{2n}.
\end{aligned}$$

□

Exemple. *On cherche à approcher $I = \int_0^1 e^{x^2} dx$. On a alors $f'(x) = 2xe^{x^2}$ pour tout $x \in [0, 1]$. Comme $x \mapsto 2x$ et $x \mapsto e^{x^2}$ sont croissantes et positives sur $[0, 1]$, f' est croissante sur $[0, 1]$. On a donc $0 \leq f'(x) \leq 2e$ pour tout $x \in [0, 1]$. On peut donc prendre $M = 2e$. Alors d'après la proposition 3.4.1, on a*

$$|I - \frac{1}{n}\sum_{i=0}^{n-1} e^{i^2/n^2}| \leq \frac{e}{n},$$

pour $n \in \mathbb{N}^*$. Comme $2e < 5.5$, on choisit $n = 55$ et on a $\frac{e}{n} \leq 0.05$. On obtient $1.447....$ On en déduit que 1.4 est une valeur approchée à 0.1 prés de I. En prenant maintenant $n = 550$, on obtient $1.4610...$ et on en déduit que 1.46 est une valeur approchée de I à 10^{-2} près.

3.4.2 Quelques autres méthodes

Plutôt que d'approcher f par une fonction constante par morceaux, on peut approcher f par une fonction affine par morceaux. Pour cette méthode, on approche f par la fonction affine qui vaut $f(x_i)$ en x_i et $f(x_{i+1})$ en x_{i+1}. Si $n \in \mathbb{N}^*$, on définit $g_n : [a, b] \to \mathbb{R}$ par $g_n(t) = f(x_i) + (t - x_i)\frac{f(x_{i+1})-f(x_i)}{x_{i+1}-x_i}$, pour tout $i \in [\![0, n-1]\!]$ et $t \in]x_i, x_{i+1}[$. Alors on peut montrer que si on suppose que f est de classe $\mathcal{C}^2$ sur $[a, b]$ et si $M_2 \in \mathbb{R}_+^*$ est tel que $|f''(x)| \leq M_2$ pour tout $x \in [a, b]$, alors $|\int_a^b f - \int_a^b g_n| \leq \frac{M_2(b-a)^2}{12n^2}$ pour tout $n \in \mathbb{N}^*$. L'avantage de cette méthode est qu'on obtient une convergence en $O(\frac{1}{n^2})$, contre une convergence en $O(\frac{1}{n})$

pour la méthode des rectangles. On peut encore raffiner et approcher f non plus par des fonctions affines par morceaux (qui sont des fonctions polynômiales de degré 1) mais par des fonctions polynômiales de degré 2 par morceaux. C'est la méthode de Simpson et on obtient alors une convergence en $O(\frac{1}{n^4})$, en supposant f de classe $\mathcal{C}^4$.

Mentionnons aussi la méthode de Monte Carlo qui est une méthode probabiliste pour calculer $\int_a^b f$. L'idée est de tirer au hasard des nombres $x_1, \ldots, x_n \in [a, b]$ et de calculer $\frac{1}{n}\sum_{i=1}^n f(x_i)$. Lorsque n tend vers $+\infty$, $\frac{1}{n}\sum_{i=1}^n f(x_i)$ donne alors une valeur approchée de $\int_a^b f$.

Chapitre 4

Arithmétique et applications

Les nombres premiers jouent un rôle fondamentale en arithmétique et ont de nombreuses applications, notamment en cryptographie. Par exemple, pour le crypto-système RSA, qui permet d'échanger des données entre deux personnes de manière chiffrée, il faut choisir deux nombres premiers p et q « grands » (ayant 1024 bits, c'est à dire de l'ordre de $2^{1024} \sim 10^{300}$). Pour obtenir de tels nombres premiers, une méthode consiste à engendrer un nombre entier « aléatoire » de l'ordre de 2^{1024} puis de tester si ce nombre est premier à l'aide d'un test de primalité.

4.1 Premiers algorithmes

Définition 4.1. *Soit $p \in \mathbb{Z} \setminus \{1, -1\}$. On dit que p est premier s'il a exactement quatre diviseurs.*

Comme tout nombre entier x admet $x, -x, 1$ et -1 comme diviseurs, un nombre p est premier si et seulement si $p \notin \{1, -1\}$ et si ses seuls diviseurs sont $1, -1, p, -p$.

Un nombre p est premier si et seulement si $-p$ est premier. On s'intéressera donc principalement aux nombres premiers positifs. On note $\mathbb{P}$ l'ensemble des nombres premiers positifs.

Lemme 4.2. *Soit $n \in \mathbb{Z}$ tel que $|n| \geq 2$. Soit p son plus petit diviseur supérieur ou égal à 2. Alors p est premier. En particulier, n admet un diviseur premier.*

Démonstration. Soit q un diviseur de p différent de 1 ou -1. Alors $|q|$ divise p donc $|q|$ divise n donc $|q| \geq p$ donc $|q| = p$, ce qui prouve le lemme. □

Proposition 4.3. *Soit n un entier composé positif. Alors il a un diviseur premier $\leq \sqrt{n}$.*

Par contraposée, on a donc : si pour tout $p \in \mathbb{P}$ tel que $p \leq \sqrt{n}$ premier, p ne divise pas n, alors n est premier.

Démonstration. Soient $n \in \mathbb{N}$ ($n \geq 2$) composé et p son plus petit diviseur ≥ 2 (qui est premier par le lemme 4.2).

Alors p divise n donc il existe $k \in \mathbb{Z}$ tel que $n = kp$. Par définition, $k \geq p$, donc $pk = n \geq p^2$, d'où $p \leq \sqrt{n}$. □

Théorème 4.4. *(admis, Hadamard, La Vallée Poussin, 1896) Pour $n \in \mathbb{N}$, on note p_n le n-ème nombre premier. Alors*

$$p_n \underset{n\to+\infty}{\sim} n\ln(n).$$

Ainsi, si $n \in \mathbb{N}$ on veut trouver un nombre premier compris entre 2 et n en prenant un nombre k au hasard et en vérifiant qu'il est premier, la probabilité que k soit premier est de l'ordre de $1/\ln(n)$. Pour n de l'ordre de 2^{1024}, on obtient environ 1 chance sur 700.

Tests élémentaires

Crible d'Eratosthène

Soit $n \in \mathbb{N}^*$. On écrit dans un tableau les $n-1$ nombres de 2 à n. On conserve 2 puis on barre tous ses multiples stricts. Le premier nombre du tableau non barré est 3. On le conserve et on barre tous ses multiples stricts. Le premier nombre du tableau non barré est 5. On le conserve et on barre tous ses multiples stricts, et ainsi de suite. On fait ainsi apparaître les premiers nombres premiers $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, ... Lorsqu'on a barré tous les multiples stricts de p_i pour $i \leq k$, le plus petit nombre supérieur à p_k qui n'est pas encore barré est p_{k+1}. Dès que l'on arrive à un nombre premier $p \geq n$, tous les nombres non barrés dans la tableau sont premiers et on a donc tous les nombres premiers inférieurs ou égaux à n. Cet algorithme donne une méthode simple pour déterminer tous les nombres premiers inférieurs à un entier positif donné : il ne necessite aucun calcul, seulement le déplacement d'un curseur. Il utilise par contre une mémoire importante.

Exemple. *Pour $n = 30$:*

	②	③	~~4~~	⑤	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~

Le programme suivant teste s'il existe un diviseur de d compris entre 2 et $\sqrt{n}$. Si n est premier, il renvoie "premier" et s'il est composé, il renvoie "composé" et le plus petit diviseur non trivial (premier) de n.

Algorithme 10. Primalité(n)

```
si n = 2
  | renvoyer premier
sinon
  | si n mod 2 = 0
  |   | renvoyer composé et 2
  | sinon
  |   | i ← 3
  |   | tant que n mod i = 0 et i ≤ √n
  |   |   | i ← i + 2
  |   | si i ≤ √n
  |   |   | renvoyer composé et i
  |   | sinon
  |   |   | renvoyer premier
```

L'avantage est qu'on obtient le plus petit diviseur premier de n.

Nombre de divisions à effectuer : $O(\sqrt{n})$. Le gros défaut est que cet algorithme est très long : si on prend n un nombre premier de l'ordre de $2^{1024} \sim 10^{308}$, avec un ordinateur testant 1000 milliards de nombre par secondes, $\sqrt{n} \sim 10^{150}$ et il faudrait au moins 10^{130} années avant de renvoyer un résultat : ce n'est pas possible.

On peut améliorer un peu cet algorithme si on connaît la liste de tous les nombres premiers inférieurs ou égaux à $\sqrt{n}$, en testant uniquement les diviseurs premiers. Cependant, le gain de temps n'est pas suffisant : d'après le théorème de Hadamard - La Vallée Poussin, il y a environ $\frac{\sqrt{n}}{\ln(\sqrt{n})}$ nombres premiers entre 2 et $\sqrt{n}$. Par exemple pour n de l'ordre de 2^{1024}, comme $\ln(10^{150}) \sim 700$, cela divise simplement le nombre de divisions à effectuer par $700/2 \sim 350$, ce qui laisse beaucoup trop de divisions à faire. De plus, cela nécessite de stocker tous les nombres premiers jusqu'à 2^{512}, ce qui prend une mémoire considérable.

4.2 Test de non-primalité de Fermat

Nous allons maintenant introduire le test de non-primalité de Fermat, beaucoup plus rapides, mais probabilistes et non complètement fiable. En pratique, il n'est pas utilisé, mais il est remplacé par le test de Miller-Rabin, qui repose sur le même principe mais est légèrement plus sophistiqué.

Théorème 4.5 (Petit théorème de Fermat)**.** *Soit p un nombre premier. Alors si a est un entier non divisible par p, on a $a^{p-1} \equiv 1[p]$.*

Ce théorème fournit un autre test de non-primalité : si on peut trouver un entier a tel que $a \wedge n = 1$ et $a^{n-1} \not\equiv 1[n]$, alors n n'est pas premier.

Proposition 4.6. *Soit $n \in \mathbb{N}_{\geq 2}$. Supposons qu'il existe $\overline{a} \in (\mathbb{Z}/n\mathbb{Z}) \setminus \{0\}$ tel que $\overline{a}^{n-1} \neq 1$. Alors $|\{\overline{a} \in \mathbb{Z}/n\mathbb{Z} | \overline{a}^{n-1} \neq \overline{1}\}| \geq \frac{n}{2}$.*

On peut donc envisager le test de primalité suivant. On choisit $k \in \mathbb{N}$.

Si $a, b \in \mathbb{N}$, on note Hasard(a, b) une fonction renvoyant un entier choisi aléatoirement entre a et b.

Algorithme 12. Test-Fermat(n, k)
pour i de 1 à k :
| $a \leftarrow$ Hasard$(1, n-1)$
| si $(a \bmod n)^{n-1} \neq 1 \bmod n$:
| | renvoyer « n est composé »
renvoyer « n est peut-être premier »

Par la proposition précédente, si Test-Fermat(n, k) renvoie « n est peut-être premier », alors la probabilité qu'il existe $\overline{a} \in (\mathbb{Z}/n\mathbb{Z})^\times$ tel que $\overline{a}^{n-1} \neq \overline{1}$ est inférieure ou égale à $\frac{1}{2^k}$. Par exemple si $k = 30$, cette probabilité est inférieure à 1 sur 1 milliard. Le problème de ce test est qu'il existe des nombres dits de Carmichael.

Définition 4.7. *Soient $a, n \in \mathbb{N}^*$. On dit que n est un nombre **pseudo-premier de Fermat de base** a si n est composé et si*

$$a^{n-1} \equiv 1[n]. \quad (*)$$

*On dit que n est un **nombre de Carmichael** (ou un nombre pseudo-premier de Fermat) si n est composé et pour tous $a \in \mathbb{Z}$ tels que $a \wedge n = 1$, on a $a^{n-1} \equiv 1[n]$.*

Il existe une infinité de nombre de Carmichael (c'est un théorème de Alford, Grandville et Pomerance de 1994). Le plus petit nombre de Carmichael est $561 = 3 \times 11 \times 17$. L'ensemble des nombres de Carmichael est assez mal compris. Empiriquement il y en a beaucoup moins que de nombre premiers. Il y a par exemple 22 nombres de Carmichael entre 1 et 10 000 alors qu'il y a 1 229 nombres premiers entre 1 et 10 000. Un nombre de 50 bits (respectivement 100) choisi aléatoirement a moins d'une chance sur 10^6 (respectivement 10^{13}) de faire échouer le test. Par rapport aux tests présentés précedemment, il est beaucoup plus rapide. Si on fixe k (par exemple $k = 30$), alors en utilisant l'exponentiation rapide, on a un coût en $O(k \log(n)^3)$ opérations élémentaires, alors que les tests que l'on a vus précédemment ont un coût supérieur à $\sqrt{n}/\ln(n)$ opérations élémentaires. En revanche, lorsque le test renvoie que n est composé, il ne fournit pas de diviseur de n. En pratique ce test n'est pas très utile, car il existe des tests à peine plus sophistiqués (les tests de Miller-Rabin et de Solovay-Strassen par exemple) pour lesquels il n'existe pas de nombre « pseudo-premiers ».

Exemple. *On prend $n = 7$ et $a = 4$. On veut calculer a^{n-1} modulo 7. Pour cela, on se place dans $\mathbb{Z}/7\mathbb{Z}$. On a $6 = 2^2 + 2$, donc*

$$\overline{4}^6 = (\overline{4})^4.\overline{4}^2 = (\overline{4}^2)^2.\overline{4}^2.$$

On a $\overline{4}^2 = \overline{16} = \overline{2.7+2} = \overline{2}$. On a $\overline{4}^4 = \overline{2}^2 = \overline{4}$, donc $\overline{4}^6 = \overline{4}.\overline{2} = \overline{8} = \overline{7+1} = \overline{1}$, donc 4^6 est bien congru à 1 modulo 7.

On prend $n = 25$. On se place dans $\mathbb{Z}/25\mathbb{Z}$. Si $a = 5$, $\overline{a}^2 = \overline{0}$, donc $\overline{a}^{24} = \overline{0}$. Prenons $a = 3$. On a $24 = 16 + 8$, donc

$$\overline{3}^{24} = \overline{3}^{16}.\overline{3}^{8} = (((\overline{3}^2)^2)^2)^2.((\overline{3}^2)^2)^2.$$

On a $\overline{3}^2 = \overline{9}$, $\overline{3}^4 = \overline{9}^2 = \overline{81} = \overline{3.25 + 6} = \overline{6}$, $\overline{3}^8 = \overline{6}^2 = \overline{25 + 11} = \overline{11}$ et $\overline{3}^8 = \overline{11}^2 = \overline{21} = \overline{-4}$, donc $\overline{3}^{24} = \overline{-4}.\overline{11} = \overline{-44} = \overline{-6} \neq \overline{1}$.

Par contre, en prenant $a = 7$, on obtient $\overline{7}^2 = \overline{49} = \overline{-1}$, donc $\overline{7}^4 = \overline{-1}^2 = \overline{1}$, donc $\overline{7}^8 = \overline{1}$ et $\overline{7}^{24} = \overline{1}^3 = \overline{1}$.

4.3 PGCD et algorithme d'Euclide

Définition 4.8. *Soient $m, n \in \mathbb{Z}$. Alors il existe un unique $d \in \mathbb{N}$ tel que d est un diviseur commun à m et à n et tel que tout diviseur commun à m et n divise d. On pose $m \wedge n = d$ et $m \wedge n$ s'appelle le PGCD de m et de n.*

Lemme 4.9. *(admis) Soient $m, n \in \mathbb{Z}$, $m \neq 0$ et r le reste dans la division euclidienne de m par n. Alors $m \wedge n = m \wedge r$.*

On en déduit l'algorithme suivant pour calculer $m \wedge n$ (algorithme d'Euclide). Quitte à remplacer m, n par $-m, -n$, on peut supposer que m, n sont positifs et quitte à échanger m et n, on peut supposer que $m \leq n$. Tant que m et n sont non nuls, on remplace m, n par r, m. Le PGCD de m et n est alors le dernier reste non nul.

Exemple. *Calculons le PGCD de 123 et 51. On a $123 = 51.2 + 21$, $51 = 21.2 + 9$, $21 = 9.2 + 3$, $9 = 3.3 + 0$, donc $121 \wedge 51 = 3$.*

4.3.1 Forme étendue de l'algorithme d'Euclide

Nous savons que si $d = PGCD(a, b)$, alors $\exists u, v \in \mathbb{Z} : d = au + bv$ $(*)$.

Question. *Peut-on trouver u et v ?*

Réponse. *Oui, grâce à une forme étendue de l'algorithme d'Euclide*

Soient $a, b \in \mathbb{N}$ et $d = a \wedge b$. Quitte à échanger a et b, on peut supposer que $b \leq a$. On veut trouver $u, v \in \mathbb{Z}$ tels que $au + bv = d$. Soient $r = a \bmod b$ et $q \in \mathbb{Z}$ tels que $a = bq + r$. Supposons que l'on connaisse $u', v' \in \mathbb{Z}$ tels que $bu' + rv' = d$. On a alors

$$d = bu' + rv' = bu' + (a - bq)v' = av' + b(u' - qv').$$

On en déduit donc u, v tels que $au + bv = d$. Comme dans le cas de l'algorithme d'Euclide, on en déduit que l'algorithme suivant, qui prend en entrée deux entiers $a > 0$ et $b \geq 0$ et renvoie un triplet d'entiers relatifs (d, u, v) vérifiant $d = a \wedge b$ et $au + bv = d$ est valide.

Algorithme Euclide-étendu(a, b)
si $b = 0$
  | renvoyer $(a, 1, 0)$
sinon
  | $(d', u', v') \leftarrow$ Euclide-étendu$(b, a \bmod b)$
  | $(d, u, v) \leftarrow (d', v', u' - \lfloor \frac{a}{b} \rfloor v')$
  | renvoyer (d, u, v)

L'exemple suivant sert à comprendre la procédure d'Euclide-étendu.

Exemple. *Étudions Euclide-étendu(105,78).*

a	b	$\lfloor \frac{a}{b} \rfloor$	d	u	v
105	78	1	3	3	$-1 - 1 \times 3 = -4$
78	27	2	3	-1	$1 - 2 \times (-1) = 3$
27	24	1	3	1	$0 - 1 \times 1 = -1$
24	3	8	3	0	$1 - 8 \times 0 = 1$
3	0		3	1	0

Nous pouvons trouver u et v également de la façon suivante :

$105 = 78 \times 1 + 27$ *donc* $3 = 27 - 24$
$78 = 27 \times 2 + 24$ $= 27 - (78 - 27 \times 2) = 27 \times 3 - 78$
$27 = 24 \times 1 + 3$ $= (105 - 78) \times 3 - 78 = 105 \times 3 - 78 \times 4$
$24 = 3 \times 8 + 0$

Les lignes 2 et 3 de l'algorithme déterminent l'arrêt du processus : si $b = 0$, alors $a = a \wedge b = a \times 1 + b \times 0$, et $(a, 1, 0)$ est alors renvoyé.

Si $b \neq 0$, Euclide-étendu calcule d'abord (d', u', v') tel que $d' = b \wedge (a \bmod b)$ et ainsi, $d' = bu' + (a \bmod b)v'$. On a alors $d = a \wedge b = b \wedge (a \bmod b) = d'$, et $d = d' = bu' + (a \bmod b)v' = bu' + (a - \lfloor \frac{a}{b} \rfloor b)v' = av' + b(u' - \lfloor \frac{a}{b} \rfloor v')$. Donc le choix $u = v'$ et $v = u' - \lfloor \frac{a}{b} \rfloor v'$ donne une solution à l'équation $d = au + bv$. L'algorithme est donc valable.

4.4 Application en cryptographie : le système RSA

Détails

Alice veut envoyer un message à Bob. Didier, un importun, aimerait bien parvenir à lire ce message.

But. *Pour Alice et Bob : envoyer des messages entre eux sans que personne d'autre ne puisse les lire.*

Alice chiffre alors son message et l'envoie à Bob de manière à ce que seul Bob sache le déchiffrer. Il faut donc que :

- Bob sache lire le message d'Alice ;

- Didier ne sache pas le lire.

Notations.
- M=$\{messages\}$
- F_A *: fonction de chiffrement d'Alice.*
- F_B *: fonction de déchiffrement de Bob.*
- m *: message d'origine (non chiffré).*
- $c = F_A(m)$ *: message codé.*

On a donc $m = F_B(c)$, donc F_B est la fonction inverse de F_A, i.e. $F_B \circ F_A = Id$.

$$c \longrightarrow m$$

Une première idée serait que Alice et Bob partagent un secret qui leur permet de chiffrer et de déchiffrer les messages : on parle dans ce cas de **chiffrement symétrique**.

Le problème de ce chiffrement est qu'Alice et Bob doivent être en mesure d'échanger ce secret en toute sécurité.

Exemple : le masque jetable ou chiffrement de Vernam
On choisit une clé (une suite de caractères, par exemple des entiers de 0 à 25) $c = (c_0, \ldots, c_k)$ de manière « aléatoire » au moins aussi longue que le message $m = (m_0, \ldots, m_k)$ à chiffrer. Le message envoyé est alors $m' = (m_0 + c_0\%26, \ldots, m_k + c_k\%26)$. Pour le destinataire du message, il suffit alors de calculer $(m' - c)\%26$. En théorie, ce chiffrement est incassable. En pratique, il est peu utilisé, car il nécessite de pouvoir échanger une clé secrète longue avant de pouvoir communiquer.

Une seconde idée est de faire un **chiffrement asymétrique**.

Principe.
- *Bob possède une clé secrète. Il ne la confie à personne, pas même à Alice.*
- *Il fabrique, à partir de cette clé secrète, une clé qu'il rend publique, de manière à ce qu'Alice puisse la consulter.*

Cette clé publique doit être construite de manière à ce que :
- Alice sache chiffrer des messages à l'aide de la clé publique.
- Bob sache déchiffrer des messages à l'aide de sa clé secrète.
- Didier ne sache pas déchiffrer les messages sans la clé ni deviner la clé secrète à partir de la clé publique et du message chiffré c.

4.5 Système RSA

Ce système a été mis en place par Rivest, Schamir et Adleman en 1977 ; il s'agit d'un système asymétrique ou encore d'un système à clé publique.

4.5.1 Construction des clés publique et privée

Principe.

- *Bob choisit deux nombres premiers distincts p et q de taille comparable.*
- *Bob calcule $n = pq$ et $\varphi = (p-1)(q-1)$.*
- *Bob choisit un entier e qui vérifie $1 < e < \varphi$ et $e \wedge \varphi = 1$.*
- *Bob cherche l'unique entier $2 \leq d \leq \varphi - 1$ tel que $de \equiv 1[\varphi]$.*
- *On obtient les clés publique et privée souhaitées :*
  - ⋆ *clé publique : (n, e) ;*
  - ⋆ *clé secrète : (d, φ).*

Plusieurs questions se posent alors :
1) Comment calculer d ?
2) Peut-on calculer d à partir de (n, e) ?

Réponses. 1) *On applique Euclide-étendu(e, φ) qui renvoie $(1, x, y)$; alors $d = x$.*
2) *Pour calculer d, il suffit de connaître $\varphi(n) = (p-1)(q-1)$. C'est facile si on connaît p et q. On ne connaît pas de méthode (rapide) si on ne connaît que n.*

4.5.2 Chiffrement

Principe.

- *Alice récupère la clé publique de Bob.*
- *Alice écrit le message sous la forme d'un entier $0 \leq m \leq n-1$.*
- *Alice calcule $C = m^e \mod n$.*
- *Alice envoie C à Bob.*

Remarques.

- *Le calcul de C s'effectue à l'aide de l'algorithme d'exponentiation modulaire rapide.*
- *Pour retrouver m à l'aide de C et de la clé publique (n, e) , il faut soit retrouver d (ce qui peut se faire avec l'algorithme d'Euclide étendu), soit calculer une racine e-ième de C mod n, i.e. trouver m_1 tel que $m_1^e \equiv C[n]$, ce qui est, de façon calculatoire, très difficile si on ne connait pas la décomposition en produit de facteurs premiers de n. Si on connaît cette décomposition, on calcule d pour obtenir la racine.*

4.5.3 Déchiffrement

Principe. *Bob utilise la clé privée (d, φ) pour retrouver $m = C^d \mod n$. Ceci se base sur le résultat suivant :*

Lemme 4.10. *Soit $m \in \mathbb{Z}/n\mathbb{Z}$. Alors $m^{ed} = m$.*

4.6 Générateurs congruentiels linéaires

En mathématiques, informatique et dans la vie courante, il est fréquent d'avoir besoin de générer des nombres « aléatoires », par exemple pour certains tests de primalité, dans la méthode de Monte-Carlo, pour tester une conjecture sur des exemples, pour tester un programme, pour jouer à un jeu de carte sur ordinateur, etc. Pour ce faire, on peut utiliser des données physiques, par exemple la dernière décimale du temps en millisecondes depuis lequel le processeur de l'ordinateur est allumé, mais ce n'est pas pratique si l'on a besoin de « beaucoup » de nombres.

En pratique, on peut utiliser des algorithmes qui engendrent des suites, à partir d'une donnée initiale, que l'on appelle « graine ». La suite renvoyée n'est par définition pas « aléatoire », puisque dès que l'on rentre la même graine, la suite obtenue est la même, mais elle en a « l'apparence » (ce qui n'est pas évident à définir). On présente ici une introduction au problème complexe de la génération de nombres pseudo-aléatoires. On pourra consulter [Knu14] pour plus d'informations sur le sujet (le livre est assez technique mais l'introduction sur le sujet est très intéressante et accessible). Un des avantages dûs au fait que pour la même graine, on obtient la même suite est que cela permet la reproductibilité des expériences.

Les générateurs congruentiels linéaires ont été inventés en 1948 par Lehmer et sont encore utilisés de nos jours par certains programmes. Le principe est assez simple. On choisit un **module** $m \in \mathbb{N}$, un **multiplicateur** $a \in [\![0, m-1]\!]$ et un **incrément** $c \in [\![0, m-1]\!]$. On choisit une graine $X_0 \in \mathbb{N}$. On définit alors $(X_n) \in [\![0, m-1]\!]^{\mathbb{N}}$ par la relation suivante :

$$X_{n+1} \equiv aX_n + c[m], n \in \mathbb{N}.$$

Autrement dit, X_{n+1} est le reste dans la division euclidienne de $aX_n + c$ par m. Il faut alors bien choisir a, c, m.

Par exemple, si on choisit $a = 2$, $c = 1$ et $m = 10$, on obtient :

- Si $X_0 = 2$, $X_1 = 5$, $X_2 = 1$, $X_3 = 3$, $X_4 = 7$, $X_5 = 5$, $X_6 = 1$, etc.
- Si $X_0 = 9$, $X_1 = 9$, etc.
- Si $X_0 = 0$, $X_1 = 1$, $X_2 = 3$, $X_3 = 7$, $X_4 = 5$, $X_5 = 3$, $X_6 = 7$, etc.

On remarque si $X_i = X_j$, pour deux entiers distincts i et j, alors $X_{i+1} = X_{j+1}$, $X_{i+2} = X_{j+2}$ etc. Par récurrence, on en déduit que $X_{i+n} = X_{j+n}$, pour tout $n \in \mathbb{N}$. On peut en déduire que si $i' \geq i$ et $k \in \mathbb{N}$, alors $X_{i'+k(j-i)} = X_{i'}$: la suite est ultimement périodique, de période $j - i$. Comme on peut choisir j tel que $j - i \leq m$, la période est au plus m. Pour que la suite soit utile, il faut que sa période soit relativement longue.

Par exemple si on choisit $a = 25$, $c = 16$, $m = 256$, on obtient :

- avec $X_0 = 125$, on obtient : $125, 69, 205, 21, 29, 229, 109, 181, 189, 133, 13, 85, 93, 37, 173, 245$,
- avec $X_0 = 96$, on obtient : $96, 112, 0, 16, 160, 176, 64, 80, 224, 240, 128, 144, 32, 48, 192, 208$,
- avec $X_0 = 50$, on obtient : $50, 242, 178, 114, 50, 242, 178, 114, 50, 242, 178, 114, 50, 242, 178, 114$,
- avec $X_0 = 10$, on obtient : $10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10$.

Lemme 4.11. *Soit d un diviseur commun à a et m. Alors la période de (X_n) est inférieure ou égale à m/d.*

Démonstration. Soit $n \in \mathbb{N}$. Écrivons $a = a'd$ et $m = m'd$, où $a', m' \in \mathbb{Z}$. Alors $X_{n+1} = a'dX_n + c + \lambda m'd$, pour un certain $\lambda \in \mathbb{Z}$ et donc $X_{n+1} \equiv c[d]$. On en déduit que pour tout $n \in \mathbb{N}^*$, $X_n \equiv c[m']$. Soit $c' \in [\![0, d-1]\!]$ le reste de la division euclidienne de c par d. Alors Comme $\{x \in [\![0, m-1]\!] | x \equiv c[d]\} = \{c', c'+d, c'+2d, \ldots, c'+(m'-1)d\}$, on a

$$|\{x \in [\![0, m-1]\!] | x \equiv c[m']\}| = m'.$$

□

D'après le lemme 4.11, il est préférable de se placer dans le cas où $a \wedge m = 1$, ce que l'on fera dans la suite.

Il est naturel de se demander si on peut obtenir une suite de période m et si oui, à quelles condition. On a alors le théorème suivant :

Théorème 4.12. *(Knuth, voir [Knu14, 3.2.1.2 Theorem A]) La suite (X_n) a pour période m si et seulement si :*

1. *c et m sont premiers entre eux,*
2. *$b := a - 1$ est un multiple de p, pour tout diviseur premier p de m,*
3. *b est un multiple de 4, si m l'est.*

Néanmoins, choisir a, c et m comme dans le théorème n'est absolument pas une garantie que la suite produite est « aléatoire ». Par exemple si on prend $a = 1$ et $c = 0$, on obtient la suite 0, 1, 2, …, $m-1$, 0, …, qui est bien périodique de période m mais n'est pas intéressante.

En pratique, trouver de « bonnes » valeurs de a, c et m est un problème difficile et on ne peut jamais être sûr que les choix que l'on a faits ne se révèleront pas mauvais a posteriori. Il faut tester les suites que l'on obtient à l'aide de tests statistiques, par exemple le test du χ^2.

Bibliographie

[Cha94] Jean-Luc Chabert. Histoire d'algorithmes : du caillou à la puce. Belin, 1994.

[Dem06] Jean-Pierre Demailly. Analyse numérique et équations différentielles. EDP sciences, 2006.

[Fil13] Francis Filbet. Analyse numérique-Algorithme et étude mathématique-2e édition : Cours et exercices corrigés. Dunod, 2013.

[Knu14] Donald E Knuth. Art of computer programming, volume 2 : Seminumerical algorithms. Addison-Wesley Professional, 2014.

[Swi12] Gérard Swinnen. Apprendre à programmer avec Python 3. Editions Eyrolles, 2012.