Modélisation et Évaluation des Systèmes Informatiques

Christophe GUYEUX et Jean-François COUCHOT
guyeux[arobase]univ-fcomte[point]fr
couchot[arobase]univ-fcomte[point]fr

12 septembre 2013

Table des matières

1 Des erreurs de problèmes numériques **2**
1.1 Erreurs de calcul en machine sur les entiers 2
1.2 Erreurs de calcul en machine sur les flottants 3
1.3 Adaptation de la méthode pour contourner les approximations 4
1.4 Erreurs sur les données 4

2 Introduction à la complexité **6**
2.1 Quelques définitions 6
2.2 Mesurer la complexité 7
2.3 Exemples d'étude de complexité 7
2.3.1 Un premier algo 7
2.3.2 Un second algo 7
2.4 Classes de complexité de problèmes 8
2.4.1 La classe *P* 8
2.4.2 La classe *NP* 8
2.4.3 La classe *NP-complet* 8

3 Interpolation polynomiale **9**
3.1 Formalisation du problème 9
3.2 Détermination efficace du polynôme d'interpolation 9

4 Résolution d'équations **12**
4.1 Motivation 12
4.2 Méthodes classiques 12
4.2.1 Méthode par dichotomie 12
4.3 Convergence des méthodes de Lagrange et Newton 14
4.4 Méthode de point fixe 14
4.4.1 Exemples de points fixe 14
4.4.2 Algorithme du point fixe 15
4.4.3 Conditions suffisantes de convergence 15
4.4.4 Vitesse de convergence 15

Chapitre 1

Des erreurs de problèmes numériques

Ce chapitre s'inspire de [Jed05] et de [BM03]. On y pointe quelques erreurs classiques en calcul numérique. On peut classer ces erreurs en plusieurs groupes :

- les erreurs de calcul en machine : elles sont dues aux arrondis de calcul pour les nombres flottants, par exemple.
- les erreurs de méthode : elles sont dues à l'algorithme utilisé. Par exemple, approximation d'une somme infinie par une somme finie, d'une limite d'une suite par un terme de grand indice, d'une intégrale par une somme finie.
- les erreurs sur les données (imprécision des mesures physiques, résultats d'un calcul approché). Ces données ne peuvent pas être modifiées, mais on peut étudier l'influence de ces erreurs sur le résultat final.

1.1 Erreurs de calcul en machine sur les entiers

On donne à la figure 1.1 les codes java et python permettant d'évaluer la fonction factorielle.

En Java, on a :

$$\begin{aligned}
5! &= 120 \\
&\vdots \\
12! &= 479001600 \\
13! &= 1932053504 \\
&\vdots \\
15! &= 2004310016 \\
16! &= 2004189184 \\
17! &= -288522240 \\
&\vdots \\
34! &= 0 \\
35! &= 0
\end{aligned}$$

On remarque que

- le résultat donné pour 13 ! est différent de 13 fois le résultat de 12 !

```java
class TestFactorielle{
  public static int  factorielle(int n){
    int r = 1;
    for (int i=2; i<=n ; i++){
      r = r * i;}
    return r;}

  public static void main(String args[]){
    for (int j=1; j< 36; j++){
      System.out.println(j+' '+
                         factorielle(j));
    }}}
```

```python
def factorielle(n):
  r=1
  i=2
  while i<=n :
    r=r*i
    i+=1
    return r

for j in range(36):
    print(str(j) + " "
          + str(factorielle(j)))
```

FIGURE 1.1 – Factorielle en Java, en Python

-2147483648	-2147483647	...	-2	-1
$100\ldots000$	$100\ldots001$	...	$111\ldots110$	$111\ldots111$

0	1	2	...	2147483647
$000\ldots000$	$000\ldots001$	$000\ldots010$	...	$011\ldots111$

TABLE 1.1 – Correspondance entiers-binaires

Nombre	Représentation	Valeur approchée	Erreur
$\frac{1}{7}$	$0,\overline{142857}\ldots$	0.14285714285714285	$7.10^{-18} + \frac{10^{-19}}{7}$
$\ln 2$	0.693147180559945309417232121458...	0.6931471805599453	$\approx 10^{-17}$
$\sqrt{2}$	1.414213562373095048801688724209...	1.4142135623730951	$> 10^{-17}$
π	3.141592653589793238462643383279...	3.141592653589793	$> 10^{-17}$

TABLE 1.2 – Interprétation erronée de nombres réels particuliers

- le résultat donné pour 16 ! est plus petit que celui donné pour 15 !
- le résultat donné pour 17 ! est négatif
- tous les résultats donnés à partir de 34 ! sont nuls !

Par contre en Python 2.7 on a des résultats cohérents :

$$\begin{aligned}
12! &= 479001600 \\
13! &= 6227020800 \\
&\vdots \\
17! &= 355687428096000 \\
&\vdots \\
34! &= 295232799039604140847618609643520000000 \\
35! &= 10333147966386144929666651337523200000000
\end{aligned}$$

Les deux langages travaillent pourtant avec des entiers et ne sont donc pas exposés aux erreurs d'arrondis.

Expliquons l'erreur d'interprétation du langage java. Celui-ci code chaque entier avec 32 bits. Le bit le plus à gauche est celui de signe. Il reste donc 31 bits. Cela permet de couvrir tous les entiers de l'intervalle

$$[\![-2147483648, 2147483647]\!].$$

Le tableau 1.1 donne la correspondance entre certains entiers et le version binaire.

Multiplier par un facteur revient à effectuer des opérations sur les bits. Tant que le résultat est inférieur à la valeur maximale des entiers, tout se passe bien. Par contre dès que le résultat est supérieur, l'interpréteur fait n'importe quoi. C'est le cas à partir de 13 !. De plus lorsqu'on a dépassé les capacités, on peut ateindre 0 sans que cela ait du sens (comme à partir de 34 !)

Si le langage python réussit (au moins à partir de 2.7), c'est parce qu'il stocke les entiers sous 64 bits et les convertit en long si besoin, dont le nombre de bits n'est pas borné. Python 3, dont le type int équivaut au long de la version 2.7 n'a pas un nombre borné de bits pour les entiers. Il ne faillit pas dans ce calcul.

1.2 Erreurs de calcul en machine sur les flottants

Un ordinateur représente chaque nombre réel sur un nombre fini de bits. Ceci ne permet la représentation exacte que d'un petit sous-ensemble des réels. La plupart des calculs sur les réels conduisent ainsi à des résultats approchés qui résultent de la discrétisation de la représentation.

Le tableau 1.2 donne des exemples de nombres réels et leur représentation par des flottants en java et en python.

On constate que les deux langages utilisent 64 bits dont 1 pour le signe, 53 pour le contenu et 11 pour l'exposant de la puissance de 10. Ils peuvent donc mémoriser au plus 17 chiffres significatifs.

1.3 Adaptation de la méthode pour contourner les approximations

Exercice 1.1. *Soit l'équation $x^2 + bx + c = 0$ avec b et c strictement positifs. On suppose que le discriminant Δ est strictement positif et proche numériquement de b^2.*

1. *Exprimer les deux racines x_1 et x_2 ($x_1 < x_2$).*
2. *Que dire du signe du numérateur de x_2 en théorie ?*
3. *En pratique quelle va être sa valeur si l'interpréteur fait des approximations.*
4. *Cette erreur d'arrondi est-elle effectuée dans le calcul de x_1 ?*
5. *Montrer qu'on pourrait calculer la racine x_2 avec $x_2 = \frac{c}{x_1}$.*
6. *Cette nouvelle méthode permet-elle de trouver le signe de x_2 ? Est-elle plus précise ?*

Travaux pratiques 1.1. *Soit l'équation $x^2 + 1.5.10^9 x + 1 = 0$. Donner une réponse pratique aux questions précédentes en effectuant les calculs en java.*

1.4 Erreurs sur les données

Les données provenant de mesures physiques sont souvent entachées d'erreurs. Par exemple, un traceur GPS ne peut avoir une précision inférieure à 8m

Ainsi, lorsqu'une méthode de calcul s'applique à des données physiques, on doit étudier l'influence des erreurs sur le résultats numérique calculé. Si une petite erreur sur les données provoque un changement radical de la solution calculée, le problème est dit *mal conditionné*.

On cherche par exemple à résoudre le problème à deux équations et deux inconnues suivant :

$$\left\{ \begin{array}{lllll} 1,2969x & + & 0,8648y & = & 0,8642 \quad L_1 \\ 0,2161x & + & 0,1441y & = & 0,1440 \quad L_2. \end{array} \right.$$

Ce système est équivalent à

$$\left\{ \begin{array}{llllll} 1,2969x & + & 0,8648y & = & 0,8642 & L_1 \\ & + & 10^{-8}y & = & -2 \times 10^{-8} & 1,2969.L_2 - 0,2161.L1 \end{array} \right.$$

qui a pour unique solution $\begin{pmatrix} 2 \\ -2 \end{pmatrix}$. Si on considère maintenant le système légèrement modifié suivant :

$$\left\{ \begin{array}{lllll} 1,2969x & + & 0,8648y & = & 0,8642 \quad L_1 \\ 0,2161x & + & 0,144y & = & 0,1440 \quad L_2 \end{array} \right.$$

Une valeur approchée à 10^{-5} près de l'unique solution de ce système serait $\begin{pmatrix} 0.66626 \\ 0.00015 \end{pmatrix}$.

On constate qu'une infime modification du système initial a eu de grandes répercutions sur les solutions du système.

Travaux pratiques 1.2. DÉFINITION 1.1 (CONDITIONNEMENT). Soit A une matrice inversible. Le *conditionnement* de A, noté *cond*(A) est défini par

$$cond(A) = ||A||\, ||A^{-1}||.$$

Une matrice A est dite bien conditionnée si son conditionnement *cond*(A) est proche de 1.

On considère les matrices

$$A = \begin{pmatrix} 4 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 4 \end{pmatrix}, A' = \begin{pmatrix} 4 & 1 & 0,1 & 0,2 \\ 1,08 & 4,04 & 1 & 0 \\ 0 & 0,98 & 3,89 & 1 \\ -0,01 & -0,01 & 1 & 3,98 \end{pmatrix}, B = \begin{pmatrix} 5 \\ 6 \\ 6 \\ 5 \end{pmatrix} \text{ et } B' = \begin{pmatrix} 5,1 \\ 5,9 \\ 6,1 \\ 4,9 \end{pmatrix}.$$

1. *Que dire de A et A', B et B'.*

2. *Résoudre à l'aide de numpy le système* $AX_1 = B$.
3. *Résoudre à l'aide de numpy le système* $AX_2 = B'$.
4. *Résoudre à l'aide de numpy le système* $A'X_3 = B$.
5. *Que dire des différents vecteurs* X_1, X_2 *et* X_3 *?*
6. *Calculer le conditionnement de* A.
7. *Reprendre les questions précédentes avec*

$$A = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}, A' = \begin{pmatrix} 10 & 7 & 8,1 & 7,2 \\ 7,08 & 5,04 & 6 & 5 \\ 8 & 5,98 & 9,98 & 9 \\ 6,99 & 4,99 & 9 & 9,98 \end{pmatrix}, B = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix} \text{ et } B' = \begin{pmatrix} 32,1 \\ 22,9 \\ 33,1 \\ 30,9 \end{pmatrix}.$$

8. *Que peut-on en conclure ?*

Chapitre 2

Introduction à la complexité

2.1 Quelques définitions

Soit $\mathcal{F}$ l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{R}_+^*$.

DÉFINITION 2.1 (FONCTIONS ASYMPTOTIQUEMENT DOMINÉES). Soit $f \in \mathcal{F}$. L'ensemble $\mathcal{O}(f)$ des fonctions *asymptotiquement dominées par* f est défini par

$$\mathcal{O}(f) = \{g \in \mathcal{F} \text{ t. q. } \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}_+, \forall n \geq n_0, g(n) \leq cf(n)\}.$$

Ainsi, à partir d'un rang n_0, $g(n)$ est majorée par $cf(n)$.

PROPOSITION 2.1. Soit f et g deux fonctions de $\mathcal{F}$. Si $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = l$ alors on a $\mathcal{O}(f) = \mathcal{O}(g)$

On confond dans ce qui suit la fonction $n \mapsto f(n)$ avec $f(n)$.

Exercice 2.1. *Que dire de :*

1. 2^{n+1} *et* $\mathcal{O}(2^n)$ *?*
2. $(n+1)!$ *et* $\mathcal{O}(n!)$ *?*

PROPOSITION 2.2. Soit deux réels ϵ et c vérifiant $0 < \epsilon < 1 < c$. Alors on a
$\mathcal{O}(1) \subset \mathcal{O}(\ln(n)) \subset \mathcal{O}(n^\epsilon) \subset \mathcal{O}(n) \subset \mathcal{O}(n\ln(n)) \subset \mathcal{O}(n^c) \subset \mathcal{O}(c^n) \subset \mathcal{O}(n!)$

DÉFINITION 2.2 (FONCTIONS DOMINANT ASYMPTOTIQUEMENT). Soit $f \in \mathcal{F}$. L'ensemble $\Omega(f)$ des fonctions *dominant asymptotiquement* f est défini par

$$\Omega(f) = \{g \in \mathcal{F} \text{ t. q. } \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}_+, \forall n \geq n_0, g(n) \geq cf(n)\}.$$

Ainsi, à partir d'un rang n_0, $g(n)$ majore $cf(n)$.

PROPOSITION 2.3. Soit f et g deux fonctions de $\mathcal{F}$. Si $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = l$ alors on a $\Omega(f) = \Omega(g)$

PROPOSITION 2.4. Soit deux réels ϵ et c vérifiant $0 < \epsilon < 1 < c$. Alors on a
$\Omega(1) \supset \Omega(\ln(n)) \supset \Omega(n^\epsilon) \supset \Omega(n) \supset \Omega(n\ln(n)) \supset \Omega(n^c) \supset \Omega(c^n) \supset \Omega(n!)$

DÉFINITION 2.3 (FONCTIONS ASYMPTOTIQUEMENT ÉQUIVALENTES). Soit $f \in \mathcal{F}$. L'ensemble $\Theta(f)$ des fonctions *asymptotiquement équivalentes* à f est défini par

$$\Theta(f) = \{g \in \mathcal{F} \text{ t. q. } \exists n_0 \in \mathbb{N}, \exists c, c' \in \mathbb{R}_+, \forall n \geq n_0, cf(n) \leq g(n) \leq c'f(n)\}.$$

PROPOSITION 2.5. Soit f et g deux fonctions de $\mathcal{F}$. Si $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = l$ alors on a $f \in \Theta(g)$ et $g \in \Theta(f)$.

PROPOSITION 2.6. Soit f, f_1, et g des fonctions de $\mathcal{F}$. Si $f \in \Theta(g)$ et si $\lim_{n\to+\infty} \frac{f_1(n)}{g(n)} = 0$ alors on a $f + f_1 \in \Theta(g)$.

2.2 Mesurer la complexité

Soit $\mathcal{A}(n)$ un algorithme résolvant un problème sur des données de taille n, $n \in \mathbb{N}$. On suppose que l'exécution de $\mathcal{A}$ coûte $T(n)$ étapes informatiques élémentaires ou unités de temps.

DÉFINITION 2.4 (PIRE DES CAS). L'algorithme $\mathcal{A}(n)$ a une *complexité du pire des cas* dans $\mathcal{O}(f(n))$ si $T(n) \in \mathcal{O}(f(n))$. On dit que $\mathcal{A}$ est en $\mathcal{O}(f(n))$.

Attention, la majoration proposée doit être la plus fine possible. De plus, celle-ci doit être valide au delà d'une valeur n supérieure à un seuil n_0.

DÉFINITION 2.5 (MEILLEUR DES CAS). L'algorithme $\mathcal{A}(n)$ a une *complexité du meilleur des cas* dans $\Omega(f(n))$ si $T(n) \in \Omega(f(n))$. On dit que $\mathcal{A}$ est en $\Omega(f(n))$.

DÉFINITION 2.6 (ORDRE DE GRANDEUR D'UN TEMPS D'EXÉCUTION). L'algorithme $\mathcal{A}(n)$ a une *complexité* en $\Theta(f(n))$ si $T(n) \in \Theta(f(n))$. On dit que $\mathcal{A}$ est en $\Theta(f(n))$.

On note que si $T(n) \in \Theta(f(n))$, alors $T(n) \in \mathcal{O}(f(n))$ et $T(n) \in \Omega(f(n))$. Lorsque $T(n) \in \Theta(n)$, on dit que l'algorithme est *linéaire en* n et lorsque $T(n) \in \Theta(n^2)$, on dit que l'algorithme est *quadratique en* n.

2.3 Exemples d'étude de complexité

2.3.1 Un premier algo

Soit p une fonction polynôme de degré n définie par $p(x) = \sum_{i=0}^{n} a_i x^i$. On considère tout d'abord l'algorithme suivant :

Data : n : degré, $(a_i)_{0 \leq i \leq n}$: coefficients, t : réel en lequel on évalue
Result : *val* : réel tel que *val* $= p(t)$

```
1 a' = a ;
2 for i = n − 1 to 0 do
3 |   a' = a_i + t × a' ;
4 end
5 val = a' ;
```

Algorithme 1: Évaluation du polynôme p en t

On calcule les coûts d'exécution des lignes de l'algorithme comme suit :

- les lignes 1 et 4 cumulées coûtent une constante de temps A indépendante de n ;
- la ligne 2 nécessite un temps d'exécution B indépendant de n ;
- la boucle for (ligne 2) nécessite donc un temps nB.

Donc $T(n) = A + nB$ et donc $\lim_{n \to +\infty} \frac{T(n)}{n} = B$. Ainsi cet algorithme a une complexité linéaire en n.

2.3.2 Un second algo

On considère l'algorithme suivant :

Data : n : entier naturel, $(x_i)_{0 \leq i \leq n}$: abscisses réelles, $(y_i)_{0 \leq i \leq n}$: ordonnées réelles.
Result : d : vecteur de $n + 1$ réels

```
1 for i = 0 to n do
2 |   d'_i = y_i ;
3 end
4 for i = 1 to n do
5 |   for j = n to i do
6 |   |   d_j = (d_j − d_{j−1})/(x_j − x_{j−1}) ;
7 |   end
8 end
```

Algorithme 2: Polynôme d'approximation d'une fonction

On calcule les coûts d'exécution des lignes de l'algo comme suit :

– la boucle for (lignes 2-2) nécessite un temps $(n+1)A$.
– la seconde boucle for (lignes 3-7) nécessite un temps $\sum_{i=1}^{n} B(n-i+1)$ soit encore $B\sum_{i=1}^{n}(n-i+1) = B\sum_{i=1}^{n} i = B\frac{n.(n+1)}{2}$

Ainsi $T(n) = (n+1)A + B\frac{n.(n+1)}{2}$ et donc l'algorithme proposé est quadratique en n.

Exercice 2.2. *Dans un cadre industriel, on est amené à effectuer un contrôle (P) sur les données. La durée de cette phase est limitée. Il existe deux méthodes $A_1(n)$ et $A_2(n)$ de traitement de (P). On a établi que le temps d'exécution $T_1(n)$ de $A_1(n)$ est en $\Theta(n^3)$ et que le temps d'exécution $T_2(n)$ de $A_2(n)$ est en $\Theta(n^2)$.*

1. *Lors d'une évaluation pratique des deux méthodes, on a obtenu les mesures suivantes*

n	$T_1(n)$	$T_2(n)$
200	10400	6800

.

   (a) *En déduire quel sera le temps nécessaire au traitement de données de taille $n' = 10^4$.*

   (b) *Même question avec $n' = 200\lambda$ où λ réel de $[1, +\infty[$.*

   (c) *Déterminer la taille maximale des données que peuvent traiter A_1 et A_2 si le temps disponible est $t = 10^5$.*

2. *En fait, une étude statistique approfondie des deux algorithmes a permis d'établir que pour n grand, on obtient : $T_1(n) \approx 0,001n^3$ et $T_2(n) \approx 0,19n^2$.*

   (a) *Quel est à priori l'algorithme le plus performant au vu de l'étude effectuée ?*

   (b) *Montrer qu'il existe un seuil n_0 en dessous duquel l'algorithme réputé le meilleur est en fait le moins performant. Déterminer cette valeur.*

   (c) *Pourquoi un algorithme performant sur des données de grandes tailles peut-il être lent pour des données de petite taille ?*

2.4 Classes de complexité de problèmes

2.4.1 La classe *P*

Les problèmes de cette classe sont ceux qui admettent une solution algorithmique *déterministe* et en temps *polynomial* : lorsque la taille du problème est n, le nombre d'étapes de l'algorithme qui le résout reste plus petit qu'une certaine puissance de n et ne contient pas d'indéterminisme. La complexité de l'algorithme est alors en $\Theta(n^k)$ pour un certain k.

2.4.2 La classe *NP*

Les problèmes de cette classe sont ceux qui admettent une solution algorithmique *non déterministe* en temps *polynomial*. De façon équivalente, c'est la classe des problèmes pour lesquels si une solution est proposée, on peut vérifier sa validité à l'aide d'un un algorithme en temps polynomial. On a *P* ⊂ *NP*.

2.4.3 La classe *NP-complet*

Les problèmes de cette classe sont ceux de la classe *NP* tels que tous les problèmes de la classe NP peuvent se réduire à celui-ci. Cela signifie que le problème est au moins aussi difficile que tous les autres problèmes de la classe NP.

Le problème du voyageur de commerce (qui consiste à trouver le chemin le plus court reliant une série de villes), le problème du sac à dos (étant donné un sous-ensemble S de l'ensemble des entiers naturels et m un nombre positif, peut-on trouver une partie A de S telle que la somme de ses éléments soit égale à l'entier m) sont des exemples de problèmes NP-complets.

Chapitre 3

Interpolation polynomiale

Soit $f : I \to \mathbb{R}$ une fonction connue seulement en $n+1$ points $x_0, x_1, \ldots, x_n$, tous dans I. Ce sont par exemples les points en lesquels la fonction f a pu être évaluée. Peut-on déterminer un polynôme p de degré au plus égal à n prenant les mêmes valeurs que la fonction f en $x_0, x_1, \ldots, x_n$. Si un tel polynôme p existe, on dit alors que p interpole f aux points (au nœuds) $x_0, x_1, \ldots, x_n$.

3.1 Formalisation du problème

DÉFINITION 3.1 (INTERPOLATION). On dit qu'un polynôme p de degré n interpole f aux points $x_0, x_1, \ldots, x_n$ si pour tout i, $0 \leq i \leq n$, on a $p(x_i) = f(x_i)$.

PROPOSITION 3.1 (UNICITÉ DU POLYNOME D'INTERPOLATION). Si p de degré inférieur ou égal à n interpole f en $x_0, x_1, \ldots, x_n$, alors p est unique.

DÉFINITION 3.2 (BASE DE LAGRANGE). On appelle base de Lagrange associée au support $\{x_0, x_1, \ldots, x_n\}$ l'ensemble des polynômes l_i, $0 \leq i \leq n$ définis par

$$l_i = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

PROPOSITION 3.2 (EXISTENCE DU POLYNOME D'INTERPOLATION). Le polynôme p défini par $p(x) = \sum_{i=0}^{n} f(x_i) l_i(x)$, où les l_i forment la base de Lagrange, interpole f en $x_0, x_1, \ldots, x_n$. On parle d'*interpolation de Lagrange* dans ce cas.

Exercice 3.1. *Montrer les résultats suivants :*

1. *Le polynôme de degré 0 qui interpole f en x_0 est donné par $p(x) = f(x_0)$.*
2. *Soit x_0 et x_1 deux réels distincts. Le polynôme de degré inférieur 1 qui interpole f en x_0 et x_1 est donné par $p(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$.*

3.2 Détermination efficace du polynôme d'interpolation

On peut écrire le polynôme p qui interpole f en $x_0, x_1, \ldots, x_n$ sous sa forme « de Newton »

$$p(x) = \sum_{i=0}^{n} d_i \left[\prod_{j=0}^{i-1} (x - x_j) \right],$$

où les d_i sont à déterminer.

On peut alors démontrer le résultat suivant.

PROPOSITION 3.3 (CONSTRUCTION ITERRATIVE SELON LA FORME DE NEWTON). Pour tout $k \in \{1, \ldots, n\}$, on a :

$$p_i(x) = p_{i-1}(x) + d_i \left[\prod_{j=0}^{i-1} (x - x_j) \right]$$

.

Ainsi pour obtenir p sur x_0, x_1, ..., x_n, il suffit :
- de calculer d_0, pour définir p_0 qui interpole p sur x_0,
- de calculer d_1, pour définir p_1 qui interpole p sur x_0 et x_1,
- ...
- de calculer d_n, pour définir p_n qui interpole p sur x_0, x_1, ..., x_n.

Le coefficient d_i est le coefficient dominant de p_i : il dépend de f et de x_0, x_1, ..., x_i. On appelle ce nombre la « différence divisée » et on le note $d_i = f[x_0, \ldots, x_i]$. On admettra la méthode suivante pour calculer la table des différences divisées.

PROPOSITION 3.4 (CALCUL DES DIFFÉRENCES DIVISÉES). On a :

1. $f[x_i] = f(x_i)$ pour tout $i \in \{0, \ldots, n\}$;
2. $f[x_i, x_{i+1}, \ldots, x_{i+j}] = \dfrac{f[x_{i+1}, \ldots, x_{i+j}] - f[x_i, x_{i+1}, \ldots, x_{i+j-1}]}{x_{i+j} - x_i}$

Ainsi on obtient :

x_0	$\mathbf{f[x_0]}$	$\mathbf{f[x_0, x_1]}$	$\mathbf{f[x_0, x_1, x_2]}$	$\ldots$	$\mathbf{f[x_0, x_1, x_2, x_3 \ldots, x_n]}$
x_1	$f[x_1]$	$f[x_1, x_2]$	$\ldots$	$f[x_1, x_2, x_3, \ldots, x_n]$	
$\ldots$					
x_{n-1}	$f[x_{n-1}]$	$f[x_{n-1}, x_n]$			
x_n	$f[x_n]$				

Les coefficients sur la première ligne fournissent les coefficients de la forme de Newton de p_n relative aux centres $x_0, \ldots x_{n-1}$: $p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \ldots + f[x_0, \ldots, x_n](x - x_0)(x - x_1) \ldots (x - x_{n-1})$

Exercice 3.2. *On donne trois valeurs d'une fonction f définie sur $[1, 6]$: $f(1) = 1,5709$, $f(4) = 1,5727$ et $f(6) = 1,5751$.*

1. *En utilisant les polynômes de Lagrange relatifs au support $\{1, 4\}$, fournir une valeur approchée de $f(3,5)$ grâce au polynôme d'interpolation de f de degré un.*
2. *En utilisant les polynômes de Lagrange relatifs au support $\{1, 4, 6\}$, fournir une valeur approchée de $f(3,5)$ grâce au polynôme d'interpolation de f de degré deux.*
3. *Traiter de nouveau ces deux questions en utilisant une forme de Newton.*
4. *Comparer ces deux méthodes et conclure.*

Travaux pratiques 3.1 (Première interpolation). 1. *Construire l'algorithme qui :*
   - *prend en entrée $x_0, \ldots, x_n$ et $f(x_0), \ldots, f(x_n)$;*
   - *retourne la matrice* `diff_div` *de taille $(n+1) \times (n+1)$ telle que* `diff_div[i][j]`$= f[x_i, \ldots, x_{i+j}]$ *pour $i \in \{0, \ldots, n\}$ et $j \in \{0, \ldots, n-i\}$ comme définie dans la proposition précédente.*

2. *Dans le cadre d'un processus industriel, on a noté le temps d'exécution $T(n)$ d'un algorithme sur des données de taille n dans le tableau suivant :*

n	10	25	60
$T(n)$	$2,3$	$8,0$	$24,6$

   (a) *Déterminer le polynôme p d'interpolation de T sur le support $\{10, 25, 60\}$ en utilisant la méthode développée à la première partie.*

   (b) *En déduire le temps $T(n)$ pour un traitement de données de taille n, avec $n \in \{15, 40, 100\}$.*

   (c) *Représenter le tracé de p.*

Travaux pratiques 3.2 (Choix du support). *L'objectif du TP est de montrer que des points régulièrement répartis sur un support contraint peu le polynôme sur les bords de celui-ci. Ainsi l'approximation peu s'en trouver dégradée sur les bords.*

On se place sur $[a, b] = [-1, 1]$ *et on considère deux interpolations de* $f(x) = e^x$ *correspondant à deux choix de supports.*

1. *Premier choix : Pour tout* $i \in \{0, \ldots, 8\}$*, on construit* $x_i = -1 + 0,25i$*.*
   (a) *Déterminer le polynôme* p_1 *d'interpolation de* f *sur* $\{x_0, \ldots, x_8\}$ *à l'aide de la méthode développée au TP précédent.*
   (b) *Représenter* p_1 *en faisant apparaître les points d'abscisse* $x_0, \ldots, x_8$*.*
   (c) *Représenter la fonction* $e_1(x) = |f(x) - p_1(x)|$ *pour tout* $x \in [-1, 1]$*.*
2. *Second choix : Pour tout* $i \in \{0, \ldots, 8\}$*, on construit* $t_i = \cos(\frac{\pi}{2} \frac{2i+1}{9})$*.*
   (a) *Déterminer le polynôme* p_2 *d'interpolation de* f *sur* $\{t_0, \ldots, t_8\}$ *à l'aide de la méthode développée au TP précédent.*
   (b) *Représenter* p_2 *en faisant apparaître les points d'abscisse* $t_0, \ldots, t_8$*.*
   (c) *Représenter la fonction* $e_2(x) = |f(x) - p_2(x)|$ *pour tout* $x \in [-1, 1]$*.*
3. *Conclure*

Chapitre 4

Résolution d'équations

4.1 Motivation

Bien qu'il puisse être posé simplement (trouver tous les x tel que $P(x) = 0$), résoudre algébriquement des équations est un problème difficile. Depuis l'antiquité, l'homme a cherché des algorithmes donnant les valeurs des racines d'un polynôme en fonction de ses coefficients. On connaît une solution pour les polynômes de degré 2. C'est Cardan qui donna en 1545 les formules de résolution de certaines équations cubiques de la forme $x^3 + px + q = 0$ avec p et q non nuls. Calculons le discriminant $\Delta = \frac{4}{27}p^3 + p^2$ et discutons de son signe :

– si Δ est nul, l'équation possède deux solutions réelles, une simple et une double :

$$\begin{cases} x_0 = 2\sqrt[3]{\frac{-q}{2}} = \frac{3q}{p} \\ x_1 = x_2 = -\sqrt[3]{\frac{-q}{2}} = \frac{-3q}{2p} \end{cases}$$

– Si Δ est positif, l'équation possède une solution réelle et deux complexes. On pose alors $u = \sqrt[3]{\frac{-q+\sqrt{\Delta}}{2}}$ et $v = \sqrt[3]{\frac{-q-\sqrt{\Delta}}{2}}$. La seule solution réelle est alors $x_0 = u + v$. Il existe également deux solutions complexes conjuguées l'une de l'autre :

$$\begin{cases} x_1 = ju + \bar{j}v \\ x_2 = j^2u + \overline{j^2} \end{cases} \qquad \text{où} \qquad j = -\frac{1}{2} + i\frac{\sqrt{3}}{2} = e^{i\frac{2\pi}{3}}$$

– si Δ est négatif, l'équation possède trois solutions réelles :

$$x_k = 2\sqrt{\frac{-p}{3}}\cos\left(\frac{1}{3}\,\text{Arccos}\left(\frac{-q}{2}\sqrt{\frac{27}{-p^3}}\right) + \frac{2k\pi}{3}\right) \qquad \text{avec} \qquad k \in \{0, 1, 2\}.$$

Donnée à titre d'exemple, ce travail montre que rapidement on obtient des algorithmes compliqués. De plus, Abel a montrée en 1824 qu'il n'est pas toujours possible d'exprimer les racines de l'équation générale de degré supérieur ou égal à 5 à partir des coefficients du polynôme, des quatre opérations et des racines nièmes... On s'intéresse donc aux méthodes numériques.

4.2 Méthodes classiques

On considère pour l'ensemble du chapitre l'équation $f(x) = 0$, où x décrit l'intervalle $[a, b]$ de $\mathbb{R}$ et f désigne une fonction définie et continue sur $[a, b]$ et à valeur dans $\mathbb{R}$. On suppose de plus la condition $f(a)f(b) \leq 0$ qui garantit l'existence d'(au moins) une solution sur $[a, b]$. On présente la méthode par dichotomie et on ferra ensuite des exercices sur d'autres méthodes.

4.2.1 Méthode par dichotomie

Construisons trois suites $(x_n)_{n\in\mathbb{N}}$, $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ définies par :

– $a_0 = a$, $b_0 = b$ et pour tout $n \in \mathbb{N}$, $x_n = (a_n + b_n)/2$, soit le milieu de $[a_n, b_n]$;
– si $f(a_n)f(x_n) \leq 0$, alors $a_{n+1} = a_n$ et $b_{n+1} = x_n$;
– si $f(a_n)f(x_n) > 0$, alors $a_{n+1} = x_n$ et $b_{n+1} = b_n$.

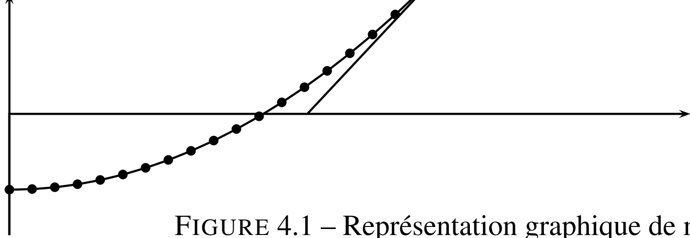

FIGURE 4.1 – Représentation graphique de méthodes itératives de construction de racine

Par récurrence montrons que $f(a_n)f(b_n) \leq 0$ pour tout $n \in \mathbb{N}$. C'est vrai au rang 0. Supposons que ce le soit jusqu'au rang n. Évaluons $f(a_{n+1})f(b_{n+1})$:
- si $f(a_n)f(x_n) \leq 0$ alors comme $a_{n+1} = a_n$ et $b_{n+1} = x_n$, le résultat est établi, c.-à-d. $f(a_{n+1})f(b_{n+1}) \leq 0$;
- sinon, $f(a_n)f(x_n) > 0$. Ainsi $f(a_{n+1})f(b_{n+1})$ a le même signe que $f(a_{n+1})f(b_{n+1})f(a_n)f(x_n)$ c.-à-d. le signe de $f(x_n)f(b_n)f(a_n)f(x_n)$ soit encore le signe de $f(a_n)f(b_n)$. Ce dernier est positif d'après l'hypothèse de récurence.

A chaque itération, l'intervalle $[a_n, b_n]$ est découpé en deux. On a donc

$$|a_{n+1} - b_{n+1}| \leq \tfrac{1}{2}|a_n - b_n|,\ a \leq a_n \leq x_n \leq b_n \leq b,\ a_{n+1} \geq a_n \text{ et } b_{n+1} \leq b_n$$

On obtient alors par récurrence que

$$|a_n - b_n| \leq \frac{1}{2^n}|a - b| \tag{4.1}$$

La suite (a_n) est croissante, majorée. Elle converge vers une limite α quand n tend vers l'infini. De même, la suite (b_n) est décroissante, minorée. Elle converge vers une limite α' quand n tend vers l'infini. D'après l'inégalité précédente, $\alpha = \alpha'$. La suite (x_n) est encadrée par (a_n) et (b_n). Elle converge donc aussi vers α. Enfin, comme f est continue et comme $f(a_n)f(b_n) \leq 0$, on a $f(\alpha)f(\alpha) \leq 0$ et donc $f(\alpha) = 0$ et donc α est une racine. On a donc trouvé une méthode algorithmique simple qui converge vers une des racines.

Essayons maintenant de déterminer une majoration de l'erreur commise à chaque itération. Tout d'abord, comme (a_n) est croissante et converge vers α on a $a_n \leq \alpha$. De même $\alpha \leq b_n$ et donc $a_n \leq \{\alpha, x_n\} \leq b_n$. Ainsi, d'après l'équation (4.1), on a

$$|x_n - \alpha| \leq |a_n - b_n| \leq \frac{1}{2^n}|a - b|.$$

Ainsi pour un ϵ donné positif représentant l'amplitude maximale de l'erreur, posons

$$n_0 = \lfloor \frac{\ln(\frac{b-a}{\epsilon})}{\ln(2)} \rfloor + 1 \tag{4.2}$$

Si $n \geq n_0$, alors $2^n \geq 2^{\frac{\ln(\frac{b-a}{\epsilon})}{\ln(2)}} \geq e^{\ln(\frac{b-a}{\epsilon})} \geq \frac{b-a}{\epsilon}$. Ainsi $\frac{1}{2^n}(b-a) \leq \epsilon$ et donc $|x_n - \alpha| \leq \epsilon$.
Pour un ϵ donné, on sait calculer un rang n_0 à partir duquel l'erreur avec la limite est inférieure à cet ϵ.

Exercice 4.1. *Soit f une fonction de $\mathbb{R}$ dans $\mathbb{R}$. On suppose que f possède une racine α dans l'intervalle $[a, b]$ et on cherche une valeur approchée de cette racine. L'idée commune des méthodes de Lagrange et Newton est d'écrire*

$$0 = f(\alpha) = f(x) + (\alpha - x)f'(\zeta),\ \textit{où}\ \zeta \in [\alpha, x] \tag{4.3}$$

On construit de façon itérative une suite $(x_n)_{n\in\mathbb{N}}$ censée converger vers α en remplaçant l'équation précédente par

$$0 = f(x_n) + (x_{n+1} - x_n)q_n \tag{4.4}$$

où q_n est une approximation de $f'(x_n)$.

Dans cet exercice, on suppose en plus que :
- *$f(x_n) \neq 0$: sinon, la racine est trouvée ;*
- *f est injective sur $[a, b]$ (pour tout x, y de $[a, b]$, si $f(x) = f(y) \Rightarrow x = y$) ;*
- *q_n n'est jamais nul.*

1. *Méthode globale*
   (a) *Montrer que l'on a $x_{n+1} \neq x_n$.*
   (b) *Montrer que (4.4) est équivalente à*

$$x_{n+1} = x_n - \frac{f(x_n)}{q_n} \tag{4.5}$$

   (c) *Dans la représentation graphique donnée figure 4.1 :*

i. donner l'équation de la droite tracée ;

ii. montrer que l'abscisse du point d'intersection de cette droite avec l'axe des abscisses est x_{n+1} ;

iii. Construire quelques x_i supplémentaires.

2. *Dans la méthode de la corde, q_n est constante et égale à*

$$q_n = \frac{f(b) - f(a)}{b - a} \tag{4.6}$$

3. *Dans la méthode de Lagrange on a*

$$q_n = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \tag{4.7}$$

4. *Dans la méthode de Newton on a $q_n = f'(x_n)$*

4.3 Convergence des méthodes de Lagrange et Newton

PROPOSITION 4.1. Soit l'équation $f(x) = 0$ et la suite (x_n) des itérés de Newton définie par x_0 et $\forall n \in N$, $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Sous les hypothèses suivantes :

1. f de classe C^2 sur $[a, b]$,
2. $f(a)f(b) < 0$,
3. $\forall x \in [a, b]$, $f'(x) \neq 0$,
4. f'' de signe constant sur $[a, b]$,
5. $\frac{|f(a)|}{|f'(a)|} < b - a$ et $\frac{|f(b)|}{|f'(b)|} < b - a$,

la suite des itérés de Newton converge pour tout choix de x_0 dans $[a, b]$ vers l'unique solution α de cet intervalle.

DÉFINITION 4.1 (ERREUR ET ORDRE). Soit (x_n) une suite convergeant vers une valeurs α. On appelle *erreur de rang n* le réel $e_n = \alpha - x_n$. On dit que la convergence est d'ordre $p \in \mathbb{R}_+^*$ si $\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^p} = c$, où c est un réel positif non nul. On remarque que plus l'ordre est élevé, plus la méthode converge vite.

PROPOSITION 4.2 (VITESSE DE CONVERGENCE). Si la suite des itérés de Lagrange converge, alors la convergence est d'ordre $(1 + \sqrt{5})/2$. Si la suite des itérés de Newton converge, alors la convergence est au moins d'ordre 2.

4.4 Méthode de point fixe

Soit f une application d'un ensemble E dans lui-même. On appelle *point fixe* de l'application f tout élément u dans E tel que $g(u) = u$. On voit que résoudre ce type d'équation est un cas particulier des équations numériques $h(u) = g(u) - u = 0$. Cependant, on peut les étudier pour les algorithmes spécifiques qui les résolvent.

4.4.1 Exemples de points fixe

L'équation $x^4 + 6x^2 - 60x + 36 = 0$ dite de Ferrari admet deux racines réelles dans l'intervalle $[0, 4]$. Pour résoudre numériquement ce problème, on peut transformer l'équation en une équation du point fixe $g_i(x) = x$ avec

- $g_1(x) = \frac{1}{60}(x^4 + 6x^2 + 36)$;
- $g_2(x) = -\frac{36}{x^3+6x-60}$;
- $g_3(x) = (-6x^2 + 60x - 36)^{\frac{1}{4}}$.

4.4.2 Algorithme du point fixe

L'algorithme du point fixe donné ci dessous (Algorithme 3) est une version constructive de la suite (x_n) définie par x_0 et pour tout $n \in \mathbb{N}$, $x_{n+1} = g(x_n)$.

Data : x_0 : terme initial, g : fonction définissant les itérés
Result : n : nombre d'itérations effectuées, $[x_1, \dots, x_n]$: vecteurs des itérés

1. $n = 0$;
2. initialisation du booléen d'*arrêt*;
3. **while** *non(arrêt)* **do**
4. $\quad x_{n+1} = g(x_n)$;
5. $\quad n = n + 1$;
6. **end**

Algorithme 3: Algorithme du point fixe

4.4.3 Conditions suffisantes de convergence

PROPOSITION 4.3. Supposons qu'il existe un intervalle $[a, b]$ de $\mathbb{R}$ sur lequel g vérifie :

1. g est définie sur $[a, b]$ et $g([a, b]) \subset [a, b]$;
2. g est dérivable sur $[a, b]$ et il existe un réel $k \in [0, 1[$ tel que pour tout $x \in [a, b]$ on a $|g'(x)| < k$;

alors :

1. g admet un point fixe l dans $[a, b]$;
2. pour tout x_0 de $[a, b]$, la suite x_n converge vers l, unique solution dans $[a, b]$.

4.4.4 Vitesse de convergence

PROPOSITION 4.4 (VITESSE DE CONVERGENCE DE LA MÉTHODE DU POINT FIXE). Sous les hypothèses de la proposition précédente, on peut dire qu'en général la convergence de la méthode du point fixe est linéaire.

Exercice 4.2. *Pour chacune des équations suivantes, étudier la convergence de la suite des itérés du point fixe pour un x_0 choisi dans l'intervalle proposé.*

1. *fonction g_3 de Ferrari sur $[2; 4]$;*
2. *fonction g_2 de Ferrari sur $[3; 3{,}1]$;*

Exercice 4.3. *On verra dans cet exercice que les conditions de la proposition 4.3 sont suffisantes, pas nécessaires.*

1. *Quels sont les points fixes de $g(x) = x - x^3$ sur $[-1; 1]$?*
2. *La fonction g vérifie-t-elle les hypothèses de la proposition 4.3 ?*
3. *Montrer que pour tout $x_0 \in [-1; 1]$, la suite des itérés converge. On pourra se restreindre à étudier x_0 sur $[0; 1]$ comme la fonction est paire.*
4. *Mêmes questions avec $g(x) = x + x^3$.*

Travaux pratiques 4.1. *Tout le code suivant est à faire en python.*

1. *Écrire la fonction* `[n,X] = iteration_dichotomie(a,b,m,epsilon,f)` *où*
   - `a`, `b` *sont les bornes de l'intervalle,* `m` *est le nombre maximal d'itérations,* `epsilon` *est la précision souhaitée (voir équation (4.2)) et* `f` *la fonction à itérer ;*
   - `n` *est le nombre d'itérations réalisées pour que* $f(x_n)=0$ *ou que* $|x_n - x_{n-1}| \leq$ `epsilon`*,* `n` *étant inférieur à* `m` *et* `X` *est le vecteur contenant les valeurs* $x_0, \dots, x_n$.
2. *Écrire la fonction* `[n,X] = iteration_corde(a,b,x0,m,epsilon,f)` *où*
   - x_0 *est le premier terme de la suite ;*
3. *Écrire la fonction* `[n,X] = iteration_Newton(x0,m,epsilon,f)`.

Travaux pratiques 4.2. *L'objectif du TP est de mesurer l'ordre de grandeur de la convergence d'une méthode. On suppose que la suite* (x_n) *converge vers* l *avec l'ordre* $p \geq 1$. *On note* $e_n = l - x_n$. *On a* $|e_{n+1}| \approx c|e_n|^p$ *et donc* $\ln(|e_{n+1}|) \approx p\ln(|e_n|) + \ln(c)$. *En posant* $y = \ln(|e_{n+1}|)$, $x = \ln(|e_n|)$ *et* $k = \ln(c)$ *on a* $y = px + k$ *soit l'équation d'une droite de pente* p. *Pour estimer* p, *on peut donc tracer l'ensemble de points* $(\ln(|e_n|), \ln(|e_{n+1}|))$, *contruire la droite de regression linéaire et prendre son coefficient directeur.*

1. *Construire la méthode* `p=ordre_convergence(X,l)` *telle que*
   - *X est le vecteur contenant les valeurs des itérés* `x0`, ..., `xn` *et* `l` *est la limite présumée de la suite ;*
   - *cette fonction exploite la fonction* `scipy.stats.linregress(x, y=None)` *;*
   - `p` *est l'ordre de convergence calculé numériquement.*
2. *Tester les méthodes du TP précédent (dichotomie, corde, Lagrange, Newton) pour la fonction* f *définie par* $f(x) = \cos(x) - x$ *sur* $[0, \pi/2]$. *Calculer l'ordre à l'aide de la fonction développée à la première question.*
3. *Comparer avec la fonction* `scipy.optimize.newton`.

Bibliographie

[BM03] J. Bastien and J.N. Martin. *Introduction à l'analyse numérique : Applications sous Matlab*. Sciences sup. Dunod, 2003.

[Jed05] F. Jedrzejewski. *Introduction aux méthodes numériques*. Springer, 2005.