Résolution de l'équation $f(x) = 0$ par dichotomie

B. Lacolle, P. Lafourcade, Groupe Algorithmique, IREM, Université Joseph Fourier

1 Introduction

La résolution de l'équation $f(x) = 0$ par dichotomie est un problème naturel entre algorithmique et mathématique. Le principe de l'algorithme est de localiser une solution de l'équation dans un intervalle de plus en plus petit. Il a été traité à d'assez nombreuses reprises, par exemple [2] ou le site Planète MATHS de l'Académie de Grenoble [5]. Pourtant sa réalisation se révèle délicate. Ceci est dû à plusieurs facteurs :

- l'approximation dans les calculs liée à la représentation des nombres en machine.
- le choix des hypothèses permettant d'obtenir un algorithme correct n'est pas facile, même si l'on considère des calculs exacts.

Ce texte est destiné aux professeurs et commence par une étude en supposant que les calculs sont faits de façon exacte. Nous évoquerons ensuite quelques questions liées à l'utilisation des calculs approchés. Les implémentations des algorithmes réalisées dans le langage Python 3 confronteront directement le lecteur à la question de la réalisation des opérations sur un ordinateur. Enfin pour les élèves, il nous semble préférable de commencer l'activité en considérant une fonction croissante (paragraphe 2.2).

2 Calcul exact

Nous supposons que les calculs effectués par la machine sont exacts.

2.1 Hypothèse simple mais dangeureuse

Hypothèse 1 : Considérons une fonction f et un intervalle $[a, b]$ tels que :

$$f(a) \times f(b) \leq 0 \tag{1}$$

En ajoutant l'hypothèse de la continuité de f sur $[a; b]$, cela constitue une condition suffisante pour qu'il existe (au moins) une solution sur $[a; b]$ de l'équation $f(x) = 0$.

Quel algorithme est correct ? Soit $\epsilon > 0$ et les deux algorithmes suivants :

Algorithme A

```
tantque b-a > ϵ
  m ← le milieu de [a;b]
  si f(a)*f(m) ≤ 0 :
    alors b ← m
    sinon a ← m
```

Algorithme B

```
tantque b-a > ϵ
  m ← le milieu de [a;b]
  si f(a)*f(m) < 0:
    alors b ← m
    sinon a ← m
```

Remarquons d'abord que les algorithmes s'arrêtent. À chaque étape l'intervalle $[a; b]$ est réduit de moitié, ainsi il existe un nombre d'itérations (éventuellement nul) après lesquelles $\mid b - a \mid \leq \epsilon$.

Preuve de l'algorithme A : Nous montrons qu'une fois l'algorithme fini nous avons un intervalle $[a;b]$ tel que :

$$\mid b-a \mid \leq \epsilon \text{ et } f(a) \times f(b) \leq 0.$$

Nous considérons l'invariant $f(a) \times f(b) \leq 0$ et montrons qu'il reste vrai lors du déroulement de l'algorithme.

```
tantque b-a > ϵ
   { Invariant : f(a)*f(b) ≤ 0 }
   m ← le milieu de [a;b]
   si f(a)*f(m) ≤ 0 :
      alors b ← m
      sinon a ← m
```

L'invariant est vérifié au début de l'algorithme d'après l'hypothèse (1).

Examinons ce qui se produit lors du déroulement d'une itération :

- dans le cas où $f(a) \times f(m) \leq 0$ le nouvel intervalle $[a';b'] = [a;m]$. Il vérifie bien cette propriété.
- dans le cas où $f(a) \times f(m) > 0$, nous avons également $f(a) \times f(b) \leq 0$ et par conséquent : $f(a)^2 \times f(b) \times f(m) \leq 0$. Or $f(a)^2 > 0$ (quantité non nulle, car $f(a) \times f(m) > 0$), on conclut donc que $f(b) \times f(m) \leq 0$ et que le nouvel intervalle $[a';b'] = [m;b]$ satisfait l'invariant.

□

Le résultat fourni par l'algorithme est donc un intervalle $[a;b]$ tel que :

$$\mid b-a \mid \leq \epsilon \text{ et } f(a) \times f(b) \leq 0.$$

En considérant le fait que f est continue, il résulte que la fonction f s'annule au moins une fois sur l'intervalle $[a;b]$ final.

Par contre, on remarquera que la continuité de la fonction ne joue aucun rôle dans la preuve de l'invariant de l'algorithme, mais est une hyptothèse nécessaire pour obtenir une solution à l'équation. On illustre ceci en appliquant l'algorithme à la fonction discontinue suivante définie sur $[-1;1]$ par :

$$f(x) = \begin{cases} -1, & x \in [-1;0[\\ 1, & x \in [0;1] \end{cases}$$

```
def f(x):
   if x < 0:
      return -1
   else:
      return 1
a = -1
b = 1
eps = 0.001
while (b-a) > eps:
   m = (a+b)/2.
   if f(a)*f(m) <= 0:
      b = m
   else:
      a = m
   print (a,b)
```

L'implémentation donne la suite des valeurs suivantes de $[a;b]$ sur un PC :

-1 0,0
-0,5 0,0
-0,25 0,0
-0,125 0,0
-0,0625 0,0
-0,03125 0,0
-0,015625 0,0
-0,0078125 0,0
-0,00390625 0,0
-0,001953125 0,0
-0,0009765625 0,0

L'algorithme B n'est pas correct Nous montrons d'abord que il n'est pas possible de faire la même preuve de l'invariant $f(a) \times f(b) \leq 0$ que celle menée pour l'algorithme A. Ceci n'est pas une preuve que l'algorithme B n'est pas correct car il est possible que que nous n'ayons pas choisi le bon invariant. Par contre cela nous donne une idée pour construire un contre-exemple. Nous démontrons donc que l'algorithme B n'est pas correct en donnant un contre-exemple simple.

Soit l'algorithme B, nous montrons que la preuve de l'invariant par la même méthode que pour l'algorithme A ne fonctionne pas.

```
tantque b-a > ϵ
   { Invariant : f(a)*f(b) ≤ 0 }
   m ← le milieu de [a;b]
   si f(a)*f(m) < 0:
      alors b← m
      sinon a ← m
```

- Si $f(a) \times f(m) < 0$ le nouvel intervalle vaut $[a'; b'] = [a; m]$ et l'invariant est vérifié.
- Dans le cas contraire on a $f(a) \times f(m) \geq 0$ et également $f(a) \times f(b) \leq 0$ ce qui implique que : $f(a)^2 \times f(b) \times f(m) \leq 0$, mais $f(a)^2 \geq 0$ et on peut avoir $f(a) = 0$, et donc un signe quelconque pour $f(m) \times f(b)$.

La preuve n'est pas possible quand une des bornes de l'intervalle annule la fonction f. Nous construisons donc un contre-exemple simple possédant cette propriété et observons le résultat de l'algorithme. Soit la fonction $f(x) = x - 1,5$ sur l'intervalle $[a; b] = [0; 2]$.

- À la première itération, m prend la valeur 1 et on a donc $f(m) = -0,5$. La condition $f(a) \times f(m) < 0$ n'est donc pas vérifiée et le nouvel intervalle devient $[1; 2]$.
- À la seconde itération m prend la valeur $1,5$ et on a donc $f(m) = 0$. La condition $f(1) \times f(1,5) < 0$ est non vérifiée et nous obtenons l'intervalle $[1,5; 2]$.
- Dans la suite du déroulement de l'algorithme la condition $f(a) \times f(b) < 0$ n'est plus jamais vérifié et la suite des intervalles proposés sont donnés par le résultat de l'implémentation suivante en Python.

```
def f(x):
   return x-1.5
a = 0
b = 2
eps = 0.001
while (b-a) > eps:
   m = (a+b)/2.
   if f(a)*f(m) < 0:
      b = m
   else:
      a = m
   print (a,b)
```

Résultats affichés par l'algorithme :
1,0 2
1,5 2
1,75 2
1,875 2
1,9375 2
1,96875 2
1,984375 2
1,9921875 2
1,99609375 2
1,998046875 2
1,9990234375 2

2.2 Hypothèse forte mais prudente

Afin d'éviter certains problèmes, en particulier devant des élèves, il nous paraît judicieux d'examiner le cas plus simple d'une fonction croissante.

Hypothèses 2 : Considérons une fonction f et un intervalle $[a; b]$ tels que :

– f croissante sur $[a;b]$;
– $f(a) \leq 0$ et $f(b) \geq 0$;
– f est continue sur $[a;b]$.

Quel algorithme est correct ?

Algorithme C

```
tantque b-a > epsilon
  m ← le milieu de [a;b]
  si f(m) ≤ 0 :
    alors a ← m
    sinon b ← m
```

Algorithme D

```
tantque b-a > epsilon
  m ← le milieu de [a;b]
  si f(m) < 0 :
    alors a ← m
    sinon b ← m
```

Contrairement au cas du paragraphe 2.1, l'inégalité stricte peut être changée en une inégalité large sans affecter la validité de la preuve. Nous considérons l'invariant suivant : $\{f(a) \leq 0 \text{ et } f(b) \geq 0\}$

La preuve de l'algorithme C :
– L'invariant est vérifié au début de l'algorithme d'après les hypothèses 2.
– Supposons l'invariant vrai au début d'une itération et montrons qu'il l'est encore après une itération de boucle. Nous distinguons deux cas :
  – $f(m) \leq 0$, le nouvel intervalle $[a';b'] = [m;b]$ convient,
  – $f(m) > 0$, donc $f(m) \geq 0$ et l'intervalle $[a';b'] = [a;m]$ convient.

Exemple d'exécution de l'algorithme C : Soit la fonction $f(x) = x - 1$ sur $[0;2]$.

```
def f(x):
  return x-1
a = 0
b = 2
eps = 0.001
while (b-a) > eps:
  m = (a+b)/2.
  if f(m) <= 0:
    a = m
  else:
    b = m
  print (a,b)
```

Résultats :
0 1,0
0,5 1,0
0,75 1,0
0,875 1,0
0,9375 1,0
0,96875 1,0
0,984375 1,0
0,9921875 1,0
0,99609375 1,0
0,998046875 1,0
0,9990234375 1,0

La preuve de l'algorithme D :
– L'invariant est vérifié au début de l'algorithme d'après les hypothèses 2.
– Supposons l'invariant vrai au début d'une itération et montrons qu'il l'est encore après une itération de boucle. Nous distinguons deux cas :
  – $f(m) < 0$, on a $f(m) \leq 0$ et le nouvel intervalle $[a';b'] = [m;b]$ convient,
  – $f(m) \geq 0$, ainsi l'intervalle $[a';b'] = [a;m]$ convient.

Exemple d'exécution de l'algorithme D : Soit la fonction $f(x) = x - 1$ sur $[0; 2]$.

```
def f(x):
   return x-1
a = 0
b = 2
eps = 0.001
while (b-a) > eps:
   m = (a+b)/2.
   if f(m) < 0:
      a = m
   else:
      b = m
   print (a,b)
```

Résultats :
1,0 2
1,0 1,5
1,0 1,25
1,0 1,125
1,0 1,0625
1,0 1,03125
1,0 1,015625
1,0 1,0078125
1,0 1,00390625
1,0 1,001953125
1,0 1,0009765625

Comment trouver la solution exacte ? Le professeur et les élèves regretteront que si la valeur exacte d'une solution est rencontrée lors de l'algorithme, elle ne soit pas détectée. Cette idée est tout à fait naturelle, par contre elle introduit deux difficultés :

- une difficulté algorithmique non négligeable avec l'ajout d'un test d'arrêt plus complexe,
- une difficulté liée au fait qu'il est très difficile de savoir dans quel cas les calculs sont effectués de façon exacte. Nous y reviendrons plus en détail dans la prochaine section.

Dans l'hypothèse que **les calculs se font de façon exacte**, pour prendre en compte la possibilité de trouver exactement une solution, nous rajouterons donc un "test logique" dans la variable `trouve` (prenant les valeurs `vrai` ou `faux`) qui nous indiquera si nous avons trouvé la solution exacte. Les hypothèses deviennent donc :

- f croissante sur $[a; b]$,
- $f(a) < 0$ et $f(b) > 0$ (la solution recherchée est dans l'intervalle $]a; b[$, ce qui signifie que l'on a testé auparavant les valeurs extrêmes a et b).

La seconde condition implique que la valeur de `trouve` sera `faux` au début de l'algorithme.

Sachant que la fonction $x \to -x^2 + 4x - 3 = (x - 1)(3 - x)$ est bien croissante sur $[0, 2]$ et que $f(0) = -3 < 0$ et $f(2) = 1 > 0$, nous donnons l'algorithme suivant qui trouve bien la solution exacte $x = 1$.

```
def f(x):
   return -x*x+4*x-3
a = 0
b = 2
eps = 0.001
trouve = 0
while ( ((b-a) > eps) and (trouve == 0) ):
   m = (a+b)/2.
   if f(m) == 0:
      trouve = 1
      print("Solution trouvee")
   else:
      if f(m) < 0:
         a = m
      else:
      b = m
   print(a,b)
if trouve==1:
   print("La solution exacte est : ",m)
else:
   print("la solution est dans l'intervalle : ")
   print(a,b)
```

3 Calculs approximatifs

Nous n'allons évidemment pas évoquer tous les aspects de cette question délicate. Notons que la question de l'arithmétique machine est une question complexe, mais elle n'est pas pour autant "aléatoire" et "sans justificatif". On pourra se référer à l'ouvrage [1] ou pour les plus curieux aux articles de vulgarisation [4, 3]. Les nombres "flottants" répondent aux normes IEEE 754 qui fixent les formats des données, les modes d'arrondi,...

3.1 Les nombres flottants : le minimum pour comprendre

Pour nos implémentations nous choisissons le langage Python exécuté sur un PC standard qui manipule des nombres "flottants" répondant aux normes IEEE 754. Brièvement et pour éviter tout détail technique retenons simplement que les nombres manipulés seront de la forme $\pm\ m\ 2^e$ où m est un entier positif et e un entier de signe quelconque. Evidemment il existe des limitations sur m et e ainsi que des conventions de codage différentes. Notons également que si ce type de représentation des nombres est assez universel sur les ordinateurs usuels, **il n'en n'est pas de même sur la plupart des calculettes**.

3.2 Des exemples à faire en classe en tout sécurité

Utiliser des *entiers*, effectuer des *additions, soustractions, multiplications*, des *divisions par des puissances de 2* n'exposeront pas à de gros problèmes. **Il n'en sera pas de même pour les divisions générales**. Par exemple, pour les algorithmes de dichotomie, considérer des polynômes à coefficients entiers sur un intervalle à bornes entières ne posera donc pas beaucoup de problèmes. Ainsi les exemples proposés jusqu'à présent correspondent à de telles hypothèses.

3.3 Les ennuis arrivent là où on ne les attend pas

Que le résultat d'un calcul complexe soit entaché d'une erreur ne surprend personne. Par contre, que des nombres simples n'aient pas toujours de représentation exacte en machine est plus étonnant : cela peut être le cas de nombres décimaux.

Nous illustrons ce phénomène en reprenant l'algorithme qui calcule une solution exacte proposé à la fin de la section précédente. Pour la fonction $f(x) = x - 1$, sur l'intervalle $[0; 2]$ le résultat du programme est tout-à-fait celui que l'on attend et vaut 1.

Par contre, en partant de l'intervalle $[-0,3; 2,3]$, nous obtenons que le milieu calculé n'est pas exactement le point $m = 1$, puisque les nombres $-0,3$ et $2,3$ ont fait l'objet d'une approximation dûe à leur représentation binaire en machine.

```
def f(x):
   return x-1
a = 0
b = 2
eps = 0.001
trouve = 0
while (((b-a)>eps) and (trouve == 0)):
   m = (a+b)/2.
   if f(m) == 0:
      trouve = 1
      print("Solution trouvee")
   else:
      if f(m) < 0:
         a = m
      else:
         b = m
   print(a,b)
if trouve==1:
   print("La solution exacte est",m)
else:
   print("la solution est dans :")
   print(a,b)
```

Le résultat sur l'intervalle $[-0,3; 2,3]$ est :
0,9999999999999999 2,3
0,9999999999999999 1,65
0,9999999999999999 1,325
0,9999999999999999 1,1624999999999999
0,9999999999999999 1,0812499999999998
0,9999999999999999 1,040625
0,9999999999999999 1,0203125
0,9999999999999999 1,0101562499999999
0,9999999999999999 1,0050781249999998
0,9999999999999999 1,0025390625
0,9999999999999999 1,00126953125
0,9999999999999999 1,0006347656249999
la solution est dans l'intervalle :
0,9999999999999999 1,0006347656249999

3.4 Un algorithme faux et des calculs approchés donnent un résultat juste !

Reprenons l'algorithme B donné au paragraphe 2.1 avec la fonction $f(x) = cos(x \times \frac{\pi}{3})$.
Sur l'intervalle $[0; 2]$ nous obtenons la suite des valeurs suivantes :

1,0 2
1,5 2
1,5 1,75
1,5 1,625
1,5 1,5625
1,5 1,53125
1,5 1,515625
1,5 1,5078125
1,5 1,50390625

1,5 1,501953125
1,5 1,5009765625

Ce programme devrait donner le même résultat faux que celui du paragraphe 2.1, sauf que l'évaluation de `cos(pi/3)` ne donne pas $0,5$ mais de façon approchée $0,5000000000000001$. [1]

Ainsi le programme évite l'erreur fatale ! Dans ce cas particulier on vérifie que par chance le programme donne un résultat correct.

3.5 Preuve d'algorithme avec des calculs approchés

Il est possible de faire des preuves y compris avec une certaine approximation des résultats. Nous faisons les **hypothèses raisonnables suivantes** pour l'évaluation approchée d'une quantité ξ :

- si la valeur exacte $\xi < 0$, alors sa valeur approchée est strictement négative ;
- si la valeur exacte $\xi > 0$, alors sa valeur approchée est strictement positive ;
- si la valeur exacte $\xi = 0$, alors sa valeur approchée est soit nulle, soit strictement positive, soit strictement négative ; en fait on ne peut rien conclure.

Reprise de la preuve de l'algorithme C Nous sommes donc confrontés à l'interprétation de la quantité $\xi = f(m)$. Si le calcul (approché) donne $f(m) \leq 0$ cela signifie soit que **la valeur exacte de** $f(m)$ est réellement strictement positive ou nulle, soit qu'il s'agit d'une approximation dans l'évaluation de $f(m) = 0$. Dans les deux cas la valeur exacte de $f(m)$ est effectivement négative ou nulle.

Dans le cas contraire, le calcul (approché) a donné $f(m) > 0$ ce qui implique soit que **la valeur exacte de** $f(m)$ est réellement strictement positive, soit qu'il s'agit d'une approximation dans l'évaluation de $f(m) = 0$. Dans les deux cas la valeur exacte de $f(m)$ est effectivement positive ou nulle.

Le même raisonnement fonctionne pour l'algorithme D.

3.6 Conclusion :

Considérer le cas d'une fonction croissante évite bien des écueils et semble adapté à une première introduction pour les élèves.

Références

[1] J. Claude Bajard and J.M. Muller (coordinateurs), *Calcul et arithmétique des ordinateurs*, Hermes (traité IC2), 2004.

[2] P. Bouttier, A. Crumière, F. Didier,J-M. Fillia, M. Quatrini, H. Roland, *Algorithmes et logique au Lycée*, Publication de l'IREM de l'Académie d'Aix-Marseille, N° 36, octobre 2009.

[3] V. Lefèvre and J.M. Muller, *L'erreur en Arithmétique des Ordinateurs*, Le Temps des Savoirs, No 2, Octobre 2000.

[4] J.M. Muller, *Ordinateurs en quete d'arithmetique*, La Recherche, No 278, Vol. 26, Juillet-Août. 1995.

[5] Site Planète MATHS de l'Académie de Grenoble, http ://www.ac-grenoble.fr/disciplines/maths/

1. Visible en exécutant sous Python la commande `print(repr(cos(pi/3.)))`