

# Résolution numérique d'équations

Skander Zannad et Judicaël Courant

2014-02-08

Cadre :  $f : I \rightarrow \mathbb{R}$ , avec  $I$  intervalle.

On cherche une approximation d'un  $x$  tel que  $f(x) = 0$ .

## 1 Méthode dichotomique

### 1.1 Principe

Cadre :  $f$  continue.

- On part de  $a$  et  $b$  ( $a < b$ ) tels que  $f(a)$  et  $f(b)$  de signe contraire.
- On calcule  $m = \frac{1}{2}(a + b)$ .
- On itère en repartant de  $a$  et  $m$  ou de  $m$  et  $b$  en fonction du signe de  $f(m)$ .

Pour savoir si on repart de  $a$  et  $m$  ou de  $m$  et  $b$  :

- On regarde le signe de  $f(a)f(m)$ .
- Si  $f(a)f(m) > 0$  :  $f(a)$  et  $f(m)$  de même, donc  $f(m)$  et  $f(a)$  de signes opposés.
- Si  $f(a)f(m) < 0$  :  $f(a)$  et  $f(m)$  de signes opposés.
- Si  $f(a)f(m) = 0$  :  $f(a)$  ou  $f(m)$  nul, on repart de  $a$  et  $m$ .

Invariant de l'algorithme :  $f(a)$  et  $f(b)$  de signes opposés (donc d'après le TVI ...)

Plus précisément :  $f(a)f(b) \leq 0$ .

Terminaison :

- pour un résultat à une précision  $\epsilon > 0$ , on s'arrête quand  $b - a < 2\epsilon$  et on rend  $\frac{a+b}{2}$ .
- L'écart  $b - a$  suit une progression géométrique de raison  $\frac{1}{2}$  donc tend vers 0, donc l'algorithme s'arrête.

## 1.2 Implantation

```
def dichotomie(f, a, b, epsilon):
    assert f(a) * f(b) <= 0 and epsilon > 0
    c, d = a, b
    fc, fd = f(c), f(d)
    while d - c > 2 * epsilon:
        m = (c + d) / 2.
        fm = f(m)
        if fc * fm <= 0:
            d, fd = m, fm
        else:
            c, fc = m, fm
    return (c + d) / 2.
```

## 1.3 Démonstration

- Terminaison
- Correction

Problème lié au produit de deux valeurs trop petites : on peut avoir  $f(a) \times f(b)$  nul avec  $f(a) \neq 0$  et  $f(b) \neq 0$ .

## 1.4 Complexité

Nombre de tours de boucle :

$$\left\lceil \log_2 \left( \frac{b-a}{\epsilon} \right) \right\rceil - 1$$

Conséquence : pour avoir  $p$  bits significatifs,  $\Theta(p)$  tours de boucle.

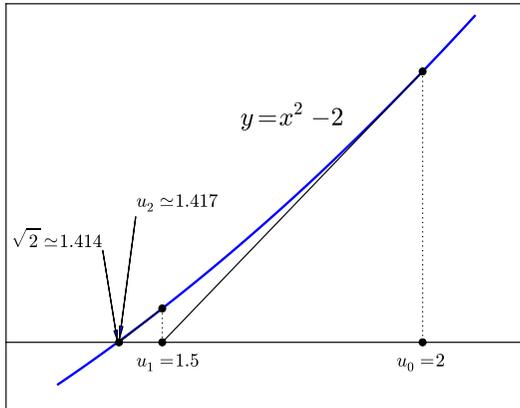
NB : on a évité de recalculer les valeurs de  $f$

- Ne change rien à la complexité asymptotique.
- Parfois important en pratique (?)

## 2 Méthode de Newton

### 2.1 Un exemple

Principe sur l'exemple de  $f : x \mapsto x^2 - 2$ .

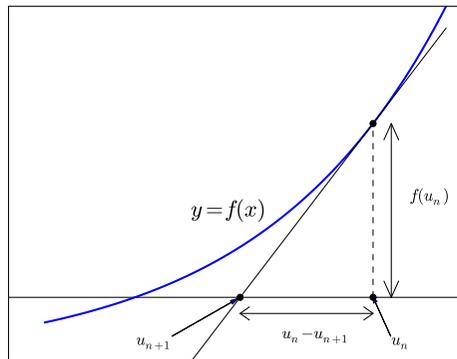


## 2.2 Définition

Construction d'une suite  $(u_n)_{n \in \mathbb{N}}$  où

- Pour  $u_0$ , on choisit une approximation d'un zéro de  $f$ .
- On pose  $u_{n+1} = F(u_n)$  pour  $n \in \mathbb{N}$ , où ...

$$F : x \mapsto x - \frac{f(x)}{f'(x)}$$



## 2.3 Implantation

```
def newton(f, fp, x0, epsilon):
    u = x0
    v = u - f(u)/fp(u)
    while abs(v-u) > epsilon:
        u, v = v, v - f(v)/fp(v)
    return v
```

## 2.4 Correction, terminaison, complexité

Problèmes :

- Division par zéro ? (Bonne définition de  $(u_n)_{n \in \mathbb{N}}$  ?)
- Terminaison ? (Convergence ?)
- Résultat proche d'un zéro de  $f$  ?

### 2.4.1 Mathématiquement

Soit  $x_0$  tel que  $f(x_0) = 0$ .

Si

- $f$  est de classe  $C^2$
- et  $f'(x_0) \neq 0$

Alors pour  $u_0$  suffisamment proche de  $x_0$ , il existe  $C > 0$  tel que

$$\forall n \in \mathbb{N} \quad |u_n - x_0| \leq C2^{-2^n}$$

Le nombre de chiffres corrects *double* à chaque itération !

Impact des erreurs de calcul : généralement faible (intérêt d'une méthode itérative).

### 2.4.2 Démonstration

Pour  $h$  au voisinage de 0 :

$$\begin{aligned} f(x_0 + h) &= f(x_0) + hf'(x_0) + \frac{1}{2}f''(x_0)h^2 + o(h^2) \\ &= hf'(x_0) + O(h^2) \end{aligned}$$

$$\begin{aligned} f'(x_0 + h) &= f'(x_0) + hf''(x_0) + o(h) \\ &= f'(x_0) + O(h) \end{aligned}$$

$$\begin{aligned} F(x_0 + h) &= x_0 + h - \frac{hf'(x_0) + O(h^2)}{f'(x_0) + O(h)} \\ &= x_0 + h - \frac{hf'(x_0)}{f'(x_0)} \frac{1 + O(h)}{1 + O(h)} \\ &= x_0 + O(h^2) \end{aligned}$$

$$F(x) = x_0 + O((x - x_0)^2)$$

Donc il existe  $\alpha > 0$  et  $K > 0$  vérifiant

$$\forall x \in [x_0 - \alpha, x_0 + \alpha] \quad |F(x) - x_0| \leq K|x - x_0|^2$$

On peut supposer  $K\alpha < \frac{1}{2}$  (quitte à diminuer  $\alpha$ ).

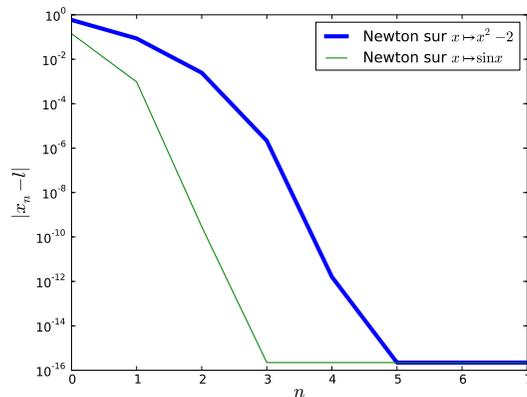
Alors :

(i)  $[x_0 - \alpha, x_0 + \alpha]$  stable par  $F$ .

(ii)  $\forall n \in \mathbb{N} \quad K|u_{n+1} - x_0| \leq (K|u_n - x_0|)^2$

$$\forall n \in \mathbb{N} \quad K|u_n - x_0| \leq (K|u_0 - x_0|)^{2^n} \leq 2^{-2^n}$$

### 2.4.3 En pratique



## 2.5 Applications

### 2.5.1 Au calcul de $1/a$ ( $a > 0$ )

Calcul du rapport/de l'inverse : opérations les plus difficiles à calculer sur un processeur.

Méthode fréquente : combinaison matérielle/(micro)logiciel.

On calcule une première approximation  $u_0$  de l'inverse de  $a$ .

On applique la méthode de Newton à la fonction  $f : x \mapsto \frac{1}{x} - a$ . On pose :

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)} = u_n + u_n - au_n^2 = u_n(2 - au_n)$$

### 2.5.2 Au calcul de la racine carrée (version mathématique)

On applique la méthode de Newton à la fonction  $f : x \mapsto x^2 - a$ , où  $a > 0$  est le nombre dont on veut la racine carrée :

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

Bien sur le papier mais nécessite un calcul d'inverse à chaque étape (plus deux multiplications, une addition et une division par deux).

### 2.5.3 Au calcul de la racine carrée (version informatique)

On applique la méthode de Newton à la fonction  $f : x \mapsto \frac{1}{x^2} - a$ , où  $a > 0$  est le nombre dont on veut la racine carrée :

$$u_{n+1} = u_n - \frac{\frac{1}{u_n^2} - a}{-\frac{2}{u_n^3}} = \frac{1}{2} u_n (3 - au_n^2)$$

On multiplie l'approximation de  $\frac{1}{\sqrt{a}}$  obtenue par  $a$  pour obtenir une approximation de  $\sqrt{a}$ .

## 2.6 Calcul approché de la dérivée

Remarque : pour la méthode de Newton, on a besoin de  $f$ .

Un moyen de la calculer numériquement :

$$\frac{f(x_0 + h) - f(x_0)}{h} \approx f'(x_0) \quad (\text{pour } h \text{ petit})$$

Erreur commise :  $\frac{h}{2} f''(x_0) + o(h)$  (si  $f$  de classe  $\mathcal{C}^2$ ).

Une approximation plus précise et pas plus dure à calculer :

$$\frac{f(x_0 + h) - f(x_0 - h)}{2h} = f'(x_0) + \frac{h^2}{6} f'''(x_0) + o(h^2) \quad (\text{si } f \text{ de classe } \mathcal{C}^3)$$

Une autre méthode, celle de la sécante : on choisit deux approximations initiales  $u_0$  et  $u_1$  et on approche  $f'(u_n)$  par  $\frac{f(u_n) - f(u_{n-1})}{u_n - u_{n-1}}$  :

$$\forall n \in \mathbb{N}^* \quad u_{n+1} = u_n - f(u_n) \frac{u_n - u_{n-1}}{f(u_n) - f(u_{n-1})}$$

On peut montrer que si  $u_0$  et  $u_1$  sont suffisamment proche du zéro cherché et  $f$  de classe  $\mathcal{C}^2$ , il existe  $C > 0$  tel que

$$\forall n \in \mathbb{N} \quad |u_n - x_0| \leq C2^{-2^n}$$

(voir livre, chap. 8)

### 3 Méthodes proposées par numpy et scipy

Intérêt de l'utilisation de numpy et scipy : ne pas réinventer l'eau chaude.

#### 3.1 Racines d'un polynôme

Pour calculer les racines de

$$P = \sum_{k=0}^n a_k X^k$$

on utilise

```
numpy.roots([a0, a1, ..., an])
```

Exemple :

```
>>> numpy.roots([2, 0, 1])
array([ 0.+0.70710678j,  0.-0.70710678j])
```

Quelle est la méthode employée ?

La plupart des utilisateurs utilisent `numpy.roots` sans se poser de questions.

Mais il y a des cas où nous avons besoin d'avoir une idée de l'erreur :

(i) On peut essayer de tester expérimentalement (mais comment être exhaustif?) ;

(ii) On peut enquêter pour trouver la réponse.

Enquêtons...

La documentation (`help(numpy.roots)`) nous dit notamment :

the algorithm relies on computing the eigenvalues of the companion matrix [1].

[1] r. a. horn & c. r. johnson, *matrix analysis*. cambridge, uk : cambridge university press, 1999, pp. 146-7.

(consulter la bibliothèque universitaire la plus proche)

Mais par quelle méthode calcule t-on ces valeurs propres ?

Le code nous renseigne :

```
a = diag(nx.ones((n-2,)), p.dtype), -1)
a[0, :] = -p[1:] / p[0]
roots = eigvals(a)
```

on construit une matrice  $a$  nulle à l'exception de la sous-diagonale (où on met des 1), puis on remplace la première ligne par les coefficients  $-\frac{a_1}{a_0}, \dots, -\frac{a_n}{a_0}$  et on calcule les valeurs propres (*eigenvalues*) de la matrice obtenue.

Allons donc voir `help(numpy.linalg.eigvals)` :

This is a simple interface to the LAPACK routines `dgeev` and `zgeev` that sets those routines' flags to return only the eigenvalues of general real and complex arrays, respectively.

LAPACK ?

Wikipédia, *Linear Algebra Package* :

LAPACK (pour Linear Algebra PACKage) est une bibliothèque logicielle écrite en Fortran, dédiée comme son nom l'indique à l'algèbre linéaire numérique.

Ok, mais quelle est la *méthode* employée ?

En utilisant un moteur de recherche («LAPACK dgeev»), on trouve un fichier `dgeev.f` (fichier FORTRAN) dont la doc évoque une décomposition QR... sans vraiment faire référence à un algorithme précis.

Conclusion : il s'agit plus d'une quête que d'une enquête

- Les algorithmes utilisés sont souvent subtils et font appels à des mathématiques relativement évoluées ;
- Difficile de savoir exactement ce qu'il se passe sous le capot ;
- On peut quand même aller lire le code de `dgeev` ;
- Heureusement, il est public... ;
- Wikipédia est en général un bon point d'entrée pour débiter la quête (ici *QR algorithm* sur <http://en.wikipedia.org>).

#### 3.2 Méthode de Newton/de la sécante

Recherche d'un zéro d'une fonction :

```
scipy.optimize.newton(func, x0, fprime)
```

La documentation est claire :

Find a zero using the Newton-Raphson or secant method.

Find a zero of the function 'func' given a nearby starting point 'x0'. The Newton-Raphson method is used if the derivative 'fprime' of 'func' is provided, otherwise the secant method is used.

#### 3.3 Méthode de Brent

```
scipy.optimize.brentq(f, a, b).
```

La qualité de la documentation est remarquable.

Ça commence par dire *ce que* ça fait :

Find a root of a function in given interval.

Return float, a zero of 'f' between 'a' and 'b'. 'f' must be a continuous function, and [a,b] must be a sign changing interval.

Puis ça continue par *comment* ça le fait :

Description : Uses the classic Brent (1973) method to find a zero of the function 'f' on the sign changing interval [a, b]. Generally considered the best of the rootfinding routines here. It is a safe version of the secant

method that uses inverse quadratic extrapolation. Brent's method combines root bracketing, interval bisection, and inverse quadratic interpolation. It is sometimes known as the van Wijngaarden-Deker-Brent method. Brent (1973) claims convergence is guaranteed for functions computable within  $[a,b]$ .

Puis *comment trouver plus d'information* (la référence historique et des références plus faciles d'accès) :

[Brent1973] provides the classic description of the algorithm. Another description can be found in a recent edition of Numerical Recipes, including [PressEtal1992]. Another description is at <http://mathworld.wolfram.com/BrentsMethod.html>. It should be easy to understand the algorithm just by reading our code. Our code diverges a bit from standard presentations : we choose a different formula for the extrapolation step.

[Brent1973] Brent, R. P., *Algorithms for Minimization Without Derivatives*. Englewood Cliffs, NJ : Prentice-Hall, 1973. Ch. 3-4.

[PressEtal1992] Press, W. H. ; Flannery, B. P. ; Teukolsky, S. A. ; and Vetterling, W. T. *Numerical Recipes in FORTRAN : The Art of Scientific Computing*, 2nd ed. Cambridge, England : Cambridge University Press, pp. 352-355, 1992. Section 9.3 : "Van Wijngaarden-Dekker-Brent Method."

## 4 Choisir une méthode

Dépend de différents facteurs :

- (i) Peut-on utiliser un programme existant ou doit-on programmer une méthode soi-même ?
- (ii) La méthode doit-elle s'appliquer à une large classe de fonctions (par exemple pas nécessairement dérivables) ou à quelques fonctions régulières et bien identifiées ?
- (iii) Dispose t-on d'une expression de la dérivée ?
- (iv) Cherche t-on une solution dans un intervalle bien précis ou n'importe quelle solution ?
- (v) A t-on besoin d'une convergence rapide ?

Suivant les cas, on choisira la méthode :

- de dichotomie ;
- des sécantes ;
- de Brent ;
- ou de Newton.

Lesquelles dans quels cas ?