Polytech'Lille — GIS 3 — Calcul Numérique — Devoir à la Maison

---

Ce document comporte une présentation théorique, suivie de calculs effectués en Maple. Les questions sont données dans la partie Maple.

1 Méthode de Newton en une variable

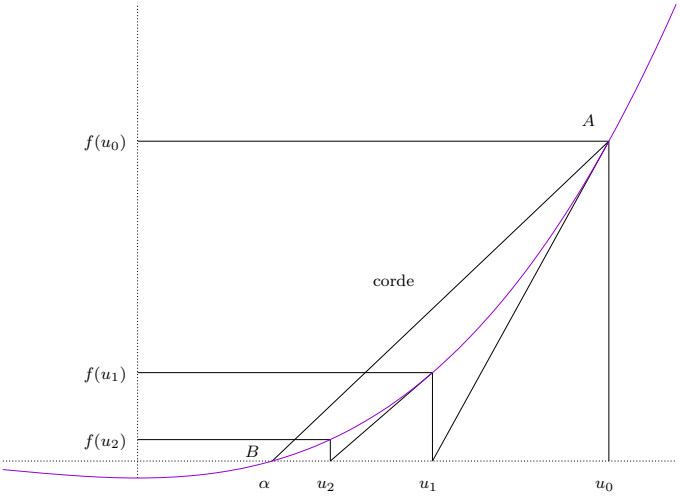

FIGURE 1 – Interprétation graphique de la méthode de Newton. À chaque itération, le terme u_{n+1} est donné par l'intersection avec l'axe des x de la tangente au graphe, prise en $(u_n, f(u_n))$.

La méthode de Newton est une méthode très utile de résolution d'équations numériques. On cherche à résoudre $f(x) = 0$ où f est une fonction suffisamment dérivable (au moins deux fois). On suppose qu'on connaît une approximation u_0 d'une racine α de f. La recette consiste à calculer les premiers termes de la suite définie par

$$u_{n+1} \quad = \quad u_n - \frac{f(u_n)}{f'(u_n)}. \tag{1}$$

Prenons par exemple $f(x) = x^2 - 2$ (la racine recherchée est $\alpha = \sqrt{2}$). La formule (1) générale donne la formule ci-dessous. Si on part de $u_0 = 3$, on obtient une approximation de α en très peu d'itérations.

$$u_{n+1} \quad = \quad \frac{u_n}{2} + \frac{1}{u_n}.$$

Remarques :

— on espère que la suite converge vers α mais, sans étude supplémentaire, on n'en est pas sûr : la suite peut diverger, répéter indéfiniment la même séquence de termes ou converger vers une autre racine que α ;
— à quel moment faut-il arrêter les calculs ? la réponse n'est pas évidente non plus : on peut fixer un nombre maximal d'itérations, tester si $|u_{n+1} - u_n| < \varepsilon$ (où ε est une précision fixée par l'utilisateur), ou si $|f(u_n)| < \varepsilon$ … ou une combinaison de ces trois critères.

Mais alors, pourquoi l'utilise-t-on (après tout, on dispose de la méthode dichotomique qui ne présente pas toutes ces difficultés) ? Parce qu'elle converge *beaucoup* plus vite que la méthode dichotomique.

1.1 Vitesse de convergence

Pour pouvoir analyser la vitesse de convergence de la méthode de Newton, il faut commencer par supposer qu'elle converge vers la racine α. Supposons donc que la suite (1) converge vers α et notons

$$e_n \quad = \quad |u_n - \alpha|$$

l'erreur à la n-ème itération. On peut montrer que, sous réserve que α soit une racine simple de f (c'est-à-dire que $f'(\alpha) \neq 0$), la vitesse de convergence est (au minimum) *quadratique*. Cela signifie qu'il existe une constante $c > 0$ telle que

$$\lim_{n \to +\infty} \frac{e_{n+1}}{e_n^2} \quad = \quad c\,. \tag{2}$$

Pour fixer les idées, oublions la limite et prenons $c = 1$. On a donc $e_{n+1} = e_n^2$ et donc, si $e_n = 10^{-p}$ (c'est-à-dire si on a p décimales exactes à l'itération n) alors $e_{n+1} = 10^{-2\,p}$ (on a $2\,p$ décimales exactes à l'itération suivante). Dit autrement, si u_n est suffisamment proche de α, le nombre de décimales exactes double (approximativement) à chaque itération.

1.2 Justification de la méthode

On s'appuie sur la figure 1. Notons G le graphe de la fonction $y = f(x)$. D'après le théorème des accroissements finis, il existe $\xi \in [\alpha, u_0]$ tel que la tangente au graphe G en $(\xi, f(\xi))$ soit parallèle à la corde reliant les points $A = (u_0, f(u_0))$ et $B = (\alpha, 0)$. Dit autrement, il existe $\xi \in [\alpha, u_0]$ tel que la pente $f'(\xi)$ de la tangente est égale à la pente de la corde :

$$\text{pente de la corde} \quad = \quad \frac{\Delta y}{\Delta x} \quad = \quad \frac{f(\alpha) - f(u_0)}{\alpha - u_0}.$$

Posons que la pente de la corde est égale à $f'(\xi)$ et tirons α. On trouve :

$$\alpha \quad = \quad u_0 - \frac{f(u_0)}{f'(\xi)}.$$

Si on connaissait ξ on aurait α immédiatement. Comme on ne connaît pas ξ, on remplace $f'(\xi)$ qui est inconnu par $f'(u_0)$, qu'on peut calculer. On obtient alors une approximation $u_1 \simeq \alpha$. Plus généralement, on obtient la suite (1).

1.3 Reformulation de la méthode

Ce paragraphe prépare la section 2.

En multipliant les membres gauche et droit de la formule (1) par $f'(u_n)$ et regroupant dans le membre gauche tous les termes qui dépendent de $f'(u_n)$, on voit que les termes u_n à calculer dans la méthode de Newton vérifient

$$f'(u_n)\,(u_{n+1} - u_n) \;=\; -f(u_n)\,.$$

On peut donc calculer u_{n+1} à partir de u_n de la façon suivante :

$$\begin{array}{rrcl} \text{résoudre} & f'(u_n)\,z & = & -f(u_n)\,, \\ & u_{n+1} & = & u_n + z\,. \end{array}$$

2 Méthode de Newton en plusieurs variables

On considère un vecteur f de deux fonctions de deux variables réelles x et y. Par exemple

$$f(x,y) \;=\; \begin{pmatrix} f_1(x,y) \\ f_2(x,y) \end{pmatrix} \;=\; \begin{pmatrix} x^2 + 2\,x\,y - 1 \\ x^2\,y^2 - y - 3 \end{pmatrix}.$$

Les solutions (réelles) de l'équation $f(x,y) = 0$ sont des vecteurs $(x\ y)^T$ de réels tels que $f_1(x,y) = f_2(x,y) = 0$. Sur l'exemple, il y a trois solutions réelles. L'une d'elles $(x\ y)^T = (2\ \ -\frac{3}{4})^T$ a des coordonnées rationnelles (les autres non).

La méthode de Newton s'applique pour résoudre de telles équations. On suppose qu'on connaît une approximation u_0 (un vecteur de dimension 2) d'une solution α. L'idée consiste à appliquer la méthode de la section 1.3 en remplaçant $f'(u_n)$ par la matrice jacobienne de f, évaluée en u_n.

$$J \;=\; \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix} \;=\; \begin{pmatrix} 2\,x + 2\,y & 2\,x \\ 2\,x\,y^2 & 2\,x^2\,y - 1 \end{pmatrix}.$$

La recette consiste donc à calculer les premiers termes de la suite ci-dessous. Il faut résoudre un système de deux équations linéaires en deux inconnues à chaque itération.

$$\begin{array}{rrcl} \text{résoudre} & J(u_n)\,z & = & -f(u_n)\,, \\ & u_{n+1} & = & u_n + z\,. \end{array} \tag{3}$$

Dans le cas de notre exemple, partons de $u_0 = (3\ \ -1)^T$. Le système linéaire à résoudre est défini par

$$J(u_0) = \begin{pmatrix} 4 & 6 \\ 6 & -19 \end{pmatrix}, \quad -f(u_0) = \begin{pmatrix} -2 \\ -7 \end{pmatrix}.$$

En appliquant (3), on trouve $z = (-\frac{5}{7}\ \frac{1}{7})^T$ et $u_1 = (\frac{16}{7}\ -\frac{6}{7})^T$.

L'analyse de la vitesse de convergence reste valable dans le cas de plusieurs variables. Supposons que la suite (3) converge vers une solution α et que cette solution soit "simple" (c'est-à-dire que $\det J(\alpha) \neq 0$). Alors, en définissant l'erreur à la n-ème itération par [1]

$$e_n \ = \ \frac{\|u_n - \alpha\|}{\|\alpha\|},$$

la vitesse de convergence est quadratique (cf. formule (2)).

Références

[1] François Boulier. Un Algorithme d'Isolation de Racines Réelles. Technical report, Université de Lille, 2017. Support du cours de calcul scientifique en PEIP 2, accessible sur `https://pro.univ-lille.fr/fileadmin/user_upload/pages_pros/francois_boulier/PEIP/deux-cercles.pdf`.

[2] Wikipedia. Newton Fractal. `https://en.wikipedia.org/wiki/Newton_fractal`.

1. le raisonnement reste vrai aussi bien pour l'erreur relative que pour l'erreur absolue et pour une norme quelconque.

Pour observer la vitesse de convergence, on calcule avec 40 décimales

```
> restart;
  Digits := 40:
```

Méthode de Newton en une variable

```
> f := x -> x^2 - 4;
  df := D(f);
```

$$f := x \mapsto x^2 - 4$$
$$df := x \mapsto 2\,x \tag{1.1}$$

```
> u[0] := 3.;
  for n from 0 to 5 do
      u[n+1] := u[n] - f(u[n])/df(u[n])
  end do;
```

$$u_0 := 3.$$
$$u_1 := 2.1666666666666666666666666666666666666667$$
$$u_2 := 2.0064102564102564102564102564102564102256$$
$$u_3 := 2.0000102400262144671090357991316457770 13$$
$$u_4 := 2.0000000000262144000001717986918411259 00$$
$$u_5 := 2.0000000000000000000001717986918400000000$$
$$u_6 := 2.0000000000000000000000000000000000000000 \tag{1.2}$$

Questions.

Répondre aux questions suivantes en Python.

Qu'observe-t-on si on part de u[0] < 0 ?
Changer la fonction f pour (x - 2)^2. Qu'observe-t-on ? Pourquoi ?
Changer la fonction f dans le but de calculer sqrt(3).

Méthode de Newton en deux variables

```
> with (LinearAlgebra):
> f1 := x^2 + 2*x*y - 1;
  f2 := x^2*y^2 - y - 3;
```

$$f1 := x^2 + 2\,x\,y - 1$$
$$f2 := x^2\,y^2 - y - 3 \tag{2.1}$$

On vérifie que (2, -3/4) est solution du système

```
> eval ([f1,f2],{x=2,y=-3/4});
```

$$[0, 0] \tag{2.2}$$

```
> J := VectorCalculus:-Jacobian ([f1,f2],[x,y]);
```

(2.3)

$$J := \begin{bmatrix} 2x+2y & 2x \\ 2xy^2 & 2x^2y-1 \end{bmatrix} \tag{2.3}$$

```
> u[0] := <3.,-1.>:
  for n from 0 to 5 do
      Jn := eval (J, {x=u[n][1], y = u[n][2]}):
      mfn := eval (< -f1, -f2 >, {x=u[n][1], y = u[n][2]}):
      z := LinearSolve (Jn, mfn):
      u[n+1] := u[n] + z
  end do:
```

```
> seq (u[n], n = 0 .. 6);
```

$$\begin{bmatrix} 3. \\ -1. \end{bmatrix}, \begin{bmatrix} 2.285714285714285714285714285714285714286 \\ -0.857142857142857142857142857142857142857 \end{bmatrix}, \tag{2.4}$$

$$\begin{bmatrix} 2.039164678841754647163342154023375734227 \\ -0.770013638561810940191374560550324119606 5 \end{bmatrix},$$

$$\begin{bmatrix} 2.001042689436719867021260440607729764935 \\ -0.750657240726826713942723453356187724135 4 \end{bmatrix},$$

$$\begin{bmatrix} 2.000000898146717819495997946024393095882 \\ -0.750000632033952081635014409854405016466 1 \end{bmatrix},$$

$$\begin{bmatrix} 2.000000000000736970653854223736514858123 \\ -0.750000000000542769241098152311621187768 4 \end{bmatrix},$$

$$\begin{bmatrix} 2.000000000000000000000000518397673520354 \\ -0.750000000000000000000000388219611041580 8 \end{bmatrix}$$

Questions.

Répondre en Python.
Remplacer l'appel à LinearSolve par des appels à des fonctions de plus bas niveau.

Optionnel: visualiser graphiquement les « bassins d'attraction » des trois solutions du système (voir section suivante pour les coordonnées des trois solutions).
Source d'inspiration [2].

Résolution exacte

Dans cette section, on utilise des méthodes « avancées » pour résoudre le système de façon exacte. Ces méthodes ne sont pas au programme du cours. Elles ne fonctionnent que pour des systèmes polynomiaux de « petite » taille.

```
> B := Groebner:-Basis ([f1,f2], plex(y,x));
```

$$B := [x^5 - 2x^3 + 2x^2 - 11x - 2,\ -x^4 + 2x^2 + 4y + 11] \tag{3.1}$$

Le système B ci-dessous a mêmes solutions que le système [f1, f2].
Le premier élément de B est un polynôme en une variable. On peut le résoudre exactement en utilisant des méthodes « par intervalles ». Voir [1].

```
> factor (B[1]);
```

$$(x-2)\,(x^4+2\,x^3+2\,x^2+6\,x+1) \tag{3.2}$$

```
> readlib(realroot):
  solutions_intervalles := realroot (B[1], 10^(-Digits));
```

$$solutions\_intervalles := \left[\left[-\frac{193674855196085699862653686702124445843987}{871122859317602466466238995025326621327 36}, -\frac{968374275980428499313268433510622229219 93}{435561429658801233233119497512663310663 68}\right], \left[-\frac{152680810192164653264793723998548055157 83}{871122859317602466466238995025326621327 36}, -\frac{610723240768658613059174895994192220631 31}{348449143727040986586495598010130648530944}\right], [2, 2]\right] \tag{3.3}$$

```
> abscisses := [seq (evalf (solutions_intervalles[i][1]), i = 1..3)];
```

$$abscisses := [-2.22327830253245406581777339776416775769, -0.17526897447251903038089533155099983408 80, 2.] \tag{3.4}$$

Les ordonnées s'obtiennent en reportant les trois abscisses dans B[2].

```
> ordonnees := [seq (solve (eval (B[2], x = abscisses[i])), i=1..3)];
```

$$ordonnees := [0.88674602860566380196123783777523312267 00, -2.76512368918815130246382958626394204797 8, -0.75000000000000000000000000000000000000 00] \tag{3.5}$$

```
> solutions := [seq (<abscisses[i], ordonnees[i]>, i=1..3)];
```

$$solutions := \left[\left[\begin{array}{c} -2.22327830253245406581777339776416775769 \\ 0.88674602860566380196123783777523312267 00 \end{array}\right], \left[\begin{array}{c} -0.17526897447251903038089533155099983408 80 \\ -2.76512368918815130246382958626394204797 8 \end{array}\right], \left[\begin{array}{c} 2. \\ -0.75000000000000000000000000000000000000 00 \end{array}\right]\right] \tag{3.6}$$

Vérification

```
> seq (eval ([f1,f2], {x=abscisses[i], y=ordonnees[i]}), i=1..3);
```

$$[1.5\,10^{-38}, -3.1\,10^{-38}], [0., 1.2\,10^{-39}], [0., 0.] \tag{3.7}$$