TP S3 : autour de la méthode de Newton dérivation numérique et formelle, méthodes de Newton et de la sécante

1 Avant la méthode de Newton : comment dériver ?

1.1 Taux de variations bêtes et calculs numériques étranges

a) Ecrire en PYTHON et en SCILAB une fonction qu'on appellera `derive` qui prend en argument une fonction `f` un point `x0` et un nombre `h` et qui renvoie le taux de variation $(f(x_0+h)-f(x_0))/h$.

b) On pense bien sûr que pour avoir une bonne valeur approchée de $f'(x_0)$, il vaut mieux prendre h petit. Essayons d'appliquer la fonction `dérive` du a) à la fonction $f : x \mapsto x^2$, pour $x0 = 7$. Le nombre dérivé exact vaut $f'(x_0) = 14$. Appliquer votre fonction avec $h = 10^{-i}$ pour i variant de 1 à 16.

c) Que pensez des résultats obtenus ? Le plus facile à expliquer est peut-être le résultat pour $h = 10^{-16}$. Pour quelle valeur de h le résultat est-il optimal ?

1.2 Taux de variations symétriques

a) **Maths :** justifier que pour une fonction de classe $\mathcal{C}^3$, on a :

$$\frac{f(x_0+h)-f(x_0-h)}{2h} \underset{h\to 0}{=} f'(x_0)+O(h^2),$$

alors qu'on a seulement :

$$\frac{f(x_0+h)-f(x_0)}{h} \underset{h\to 0}{=} f'(x_0)+O(h).$$

On appelle ici *taux de variations symétriques* les expressions $\frac{f(x_0+h)-f(x_0-h)}{2h}$. Les formules précédentes permettent d'espérer que les taux de variations symétriques donnent une meilleure approximation du nombre dérivé.

b) **Info. :** Ecrire une fonction `derive2` qui prend les mêmes arguments que `derive` et renvoie le taux de variation symétrique.

c) Faire le même test avec `derive2` qu'avec `dérive`.

On étudiera plus précisément les raisons des phénomènes observés aux questions c) en cours.

1.3 Excursions pour les fonctions polynomiales : gestion formelle de la dérivation des polynômes en SCILAB

L'essentiel : la donnée d'une fonction polynomiale $f : \mathbb{R} \to \mathbb{R}$, $x \mapsto \sum_{k=0}^{n} a_k x^k$ est équivalente à la donnée de la suite de ses coefficients, i.e. un tableau $[a_0, \ldots, a_n]$.

1.3.1 Déclaration d'un polynôme formel en SCILAB

On peut déclarer le polynôme formel p associé à la suite $[a_0, \ldots, a_n]$ avec la commande `poly` avec la syntaxe :

```
-->p=poly([2,3,1],'x','c')
```

Explication : Le `'x'` désigne le nom de l'indéterminée, le `'c'` est pour coefficients. En effet, on peut aussi définir un polynôme à partir de la valeur de ses racines `'r'` (argument par défaut).

```
 -->q=poly([2,3],'x') // fabrique le polynôme unitaire dont les racines sont 2 et 3
// autrement dit (x-2)(x-3)
```

1.3.2 Opérations sur les polynômes formels

On peut les dériver avec la commande `derivat`. Comme ce ne sont pas des fonctions, il faut une commande spéciale pour les évaluer : la commande `horner` (qui effectue bien sûr l'algorithme de Horner cf. chap. S1).

1.3.3 Généralisation aux quotients de polynômes

Tout ce qui précède s'applique encore aux quotients $x \mapsto f(x) = p(x)/q(x)$ avec p et q polynômes. Si `p` et `q` sont des polynômes formels, SciLab peut gérer le quotient `p/q` comme un objet formel, qu'il sait dériver, évaluer.

Conséquence : Si on veut appliquer la méthode de Newton à une fonction polynomiale ou rationnelle mieux vaudra utiliser ces commandes pour la dérivation, pour manipuler une valeur exacte de la fonction dérivée.

1.4 Si on veut calculer des dérivées de manière formelle pour les fonctions usuelles

On passe dans le monde du calcul formel :
• SciLab ne fait pas de calcul formel (à part ce qu'on vient de dire pour les polynômes formels), pas plus que Matlab,
• il existe des logiciels de calculs formels comme Maple, Mathematica, mais aussi en Python, le module `sympy` pour *SYMbolic computation with Python*

Exemple :

```
 >>> from sympy import * # oui je sais c'est dangereux de tout importer comme ça.
 # mais ici le import sympy as sp demanderait trop de préfixes après...
>>> x=symbols('x')
>>> diff(sin(x),x)
cos(x)
```

Vous pouvez ainsi jouer à vérifier que `sympy` calcule des dérivées formelles des fonctions usuelles, comme vos calculatrices sans doute.... mais le numérique revient quand même dans le processus de l'évaluation.

2 La méthode de Newton

2.1 On fournit la fonction et sa dérivée

a) Ecrire une fonction Python et la même fonction en SciLab qu'on appellera `newton1`
- qui prend quatre arguments : une fonction `f`, sa dérivée `fp` que l'utilisateur a l'amabilité d'avoir défini pour la machine, un nombre `x0` et un nombre `epsilon`,
- applique la méthode de Newton à la fonction `f` en partant de `x0`,
- qui s'arrête quand l'écart entre deux termes successifs $|x_{n+1} - x_n|$ est plus petit que `epsilon`.
- et qui renvoie la dernière valeur de la suite calculée (x_{n+1} avec les notations du point précédent) ainsi que le nombre d'itérations de la méthode i.e. $n + 1$.

Question : avec ce test d'arrêt, a-t-on une estimation a priori de la différence entre la valeur approchée (x_{n+1}) et la racine r qu'on cherche ?

b) i) Appliquer votre fonction `newton1` à la fonction `sin` : en PYTHON, on utilisera le module `math`.

Tester `newton1(sin,cos,x0,epsilon)` pour `x0` de `0.1` à `2` avec un pas de `0.1` et `epsilon=10^(-7)`.

ii) En déduire une valeur approximative de la borne supérieure *bassin d'attraction* de 0 à droite de 0. On notera b cette borne supérieure.

iii) Que se passe-t-il pour x_0 à droite de b : la suite converge-t-elle forcément vers le zéro qui est à droite i.e. π ?

c) **A propos du test d'arrêt**

i) **Maths :** Justifier que si le zéro r recherché pour f vérifie $f'(r) \neq 0$ et si f est de classe $\mathcal{C}^2$ (hyp. du cours) alors :

$$x_{n+1} - x_n \underset{n\to+\infty}{\sim} x_n - r.$$

Ainsi la condition d'arrêt $|x_{n+1} - x_n| < \varepsilon$ « n'est pas loin » de donner $|x_n - r| < \varepsilon$.

N.B. Cependant, comme vu dans l'exemple du cours pour les fonctions convexes (où la suite (x_n) est décroissante), il n'y a pas de raison que r soit inclus dans $[x_{n+1}, x_n]$.

ii) **Maths :** Toujours dans l'hypothèse où $f'(r) \neq 0$, justifier qu'il existe une constante $k \neq 0$ telle que $f(x_n) \underset{n\to+\infty}{\sim} k(x_n - r)$ et donc qu'on peut aussi utiliser comme test d'arrêt la condition $|f(x_n)| < \varepsilon$.

iii) **Info :** Réécrire le code (plus simple) d'une fonction `newton2` qui utilise le test d'arrêt du (ii)

d) On peut comparer notre fonction à celle implémentée dans `scipy` ou plus précisément dans le sous-module `scipy.optimize` : cette fonction s'obtient via `scipy.optimize.newton`.

i) Lire la documentation de cette fonction de `scipy`.

ii) En déduire comment appliquer la méthode de Newton à `f=sin` avec cette fonction, en rentrant la précision voulue.

N.B. En SCILAB comme en `scipy.optimize`, la commande `fsolve` n'utilise pas exactement Newton comme indiqué par le `help fsolve` qui parle de *méthode hybride de type Powell*.

2.2 Version pour les polynômes en SCILAB

a) Avec les informations données au § 1.3 écrire une fonction `newtonpoly` qui prend en argument un polynôme de SCILAB, un `x0` et un `epsilon` et lui applique la méthode de Newton. On utilisera la commande `horner` pour l'évaluation.

b) Tester cette méthode sur $f : x \mapsto (x-a)(x-b)$ pour a et b de votre choix. Oui je sais, on connaît les racines ! Mais dans ce cas, quels sont les *bassins d'attraction* de a et de b ? Il y a-t-il des valeurs de `x0` qui posent problèmes ici ? Chercher *numériquement* la réponse à ces questions.

c) Tester cette méthode sur $f : x \mapsto x^3 - x$ dont, là encore, on connaît bien les racines. Essayez pour les valeurs de `x0` suivantes :

i) $x_0 = -0,5$: commentez le résultat ?

ii) Expliquer par une étude théorique les problèmes posées par les valeurs $x_0 = \pm\dfrac{1}{\sqrt{3}}$ et $x_0 = \pm\dfrac{1}{\sqrt{5}}$. Pour voir le problème posé par cette valeur de x_0 on fera afficher les valeurs successives de la suite (x_n).

iii) Tester expérimentalement le comportement de la méthode pour les x_0 tels que $|x_0| < \dfrac{1}{\sqrt{3}}$, $|x_0| > \dfrac{1}{\sqrt{5}}$ (pour gagner du temps, on pourra ne considérer que les $x_0 > 0$.)

iv) Tester enfin ce qui se passe pour $x_0 \in [\dfrac{1}{\sqrt{3}}, \dfrac{1}{\sqrt{5}}]$ (ou son symétrique dans les négatifs).

d) Etudions maintenant d'un point de vue différent la méthode de Newton appliquée à un polynôme $f_\alpha : x \mapsto x^3 + \alpha x + 1$ en appliquant toujours la méthode à partir de $x_0 = 0$. On va procéder comme on l'a fait au T.P. S1 pour l'étude de l'itération de $f : x \mapsto x^2 + c$.

i) Pour chaque valeur de α dans $[-2, -1]$ avec un certain pas, on calcule les itérés de la méthode de Newton pour f_α à partir de $x_0 = 0$, qu'on note (x_n). On affiche les points (x_n, α) pour $n \in [\![20, 120]\!]$.

(Autrement dit, sur chaque droite horizontale d'ordonnée α, on place les points d'abscisses x_n).

ii) Le polynôme f_α possède une unique racine si $\alpha > \alpha_0 = -(3/2)\sqrt[3]{2} = -1.889\ldots$ et trois racines si $\alpha < \alpha_0$.

Ceci peut aider à interpréter certaines zones de la figure obtenue au (i)

2.3 Version utilisant la dérivation formelle vs dérivée numérique

a) Modifier votre programme PYTHON du 2.1. pour que l'utilisateur n'ait pas à entrer la dérivée, mais que celle-ci soit calculée formellement par `sympy`.

b) Modifier votre programme PYTHON du 2.1. pour que l'utilisateur n'ait pas à entrer la dérivée, mais que celle-ci soit calculée numériquement par un taux de variation symétrique à 10^{-7} près.

c) Voit-on une différence d'efficacité des programmes des a) et b) (dérivation formelle ou numérique) sur l'équation $\sin(x) = 0$?