Sur la recherche des racines simples d'un polynôme

(L1, cc2-gr4 Python)

jpr, Nice, 01/12/2016

François Godi (ENS-Rennes) vous a proposé un intéressant problème : celui de la **recherche de la liste complète des racines simples réelles d'un polynôme réel p, situées à l'intérieur d'un intervalle $[a, b]$ donné, avec une précision de h.** Si $p(X)$ est un polynôme, il est usuel de dire que le réel α est une *racine* (ou un *zéro*) de p si $p(\alpha) = 0$. Lorsque α est racine, on peut factoriser $p(X)$ sous la forme $p(X) = (X - \alpha)q(X)$ et on dit que α est une racine **simple** de $p(X)$ si $q(\alpha) \neq 0$. Par exemple si $p(X) = (X + 2)(X - 3)(X^2 + 1)(X - 5)^2$, les racines simples dans $\mathbb{R}$ de $p(X)$ sont -2 et 3. La racine 5 n'est pas *simple*, elle est *multiplicité* 2. Quant au facteur $X^2 + 1$, il a bien deux racines simples mais elles sont complexes. Le but final est de programmer en Python une fonction `racines(p,a,b,h)` où h est la précision souhaitée. Et l'on souhaite que :

```
>>> racines(p,-3,10,0.001)     # les racines de p dans [-3,10] avec une précision ≤ 10^-3
[-1.9999997082008265, 3.0000042055084783]
```

Euh, il y a des étudiants qui ont cru qu'il s'agissait d'utiliser la fonction `sqrt`. Non, rien avec à voir avec la **racine carrée** ! Il s'agissait du mot *racine* au sens de la racine d'une équation. En cas de doute sur le vocabulaire, posez la question à l'enseignant, n'allez pas arracher des pissenlits !

1 Présentation de la technique

Vous savez trouver une racine de l'équation $f(x) = 0$ sous de bonnes conditions[1] par la **méthode des tangentes de Newton** étudiée dans l'exercice de TD 2.3, qui consiste à partir d'une approximation et de converger vers une racine. Cette technique est très efficace mais a l'inconvénient de ne donner qu'une seule racine, et n'importe laquelle s'il en existe plusieurs. Il faut donc trouver une autre technique.

Or une autre technique vous a déjà été présentée ! Dans la page Web du cours Python, cherchez la première apparition du mot *exemple* et vous avez déjà 25% de votre cc2 qui était résolu là-bas dans le fichier `hoering.py`. Lisez-vous les compléments de cours ? Cette autre technique est la **recherche par dichotomie**, que nous avons d'ailleurs étudiée dans le cours 6 pp. 12–14 et dans l'exercice de TD 6.4 (correction sur le Web dans `sol6.py`). Mais lisez-vous les corrections ? Je vous taquine un peu, mais j'insiste : vous n'êtes définitivement plus au lycée mais à l'université, il va falloir vous habituer à un travail personnel *intense*. C'est ça ou la galère...

Fuyons la galère. La dichotomie consiste à couper en deux une donnée, un problème, pour ici en abandonner chaque fois la moitié. Donc très très efficace, presque miraculeux pour un programmeur qui se doit de maîtriser cette technique. Rapidement : soit f une fonction continue et strictement monotone[2] sur $[a, b]$ telle que $f(a)f(b) \neq 0$. Alors :

- ou bien elle n'a aucune racine[3] dans $[a, b]$. C'est le cas si $f(a)f(b) > 0$.
- ou bien elle n'en a qu'une seule. Pour la trouver, on regarde si le milieu m de $[a, b]$ est racine. Si oui, gagné. Sinon, on continue sur l'une des moitiés $[a, m]$ ou $[m, b]$ suivant le signe du produit $f(a)f(m)$. Voir le fichier `hoering.py` cité plus haut.

[1] En particulier, il faut que la racine existe, que la fonction f soit dérivable et à dérivée presque partout non nulle, pour éviter de tomber sur des sommets où la tangente serait horizontale.

[2] *Monotone* signifie croissante ou décroissante.

[3] *Racine* : solution de $f(x) = 0$.

Mais encore le même problème : on ne trouve qu'une seule racine et à condition d'être en possession d'un intervalle sur lequel le polynôme est strictement monotone. La GROSSE astuce (qui vous était donnée) consiste à travailler entre deux sommets consécutifs du polynôme, et là il sera strictement croissant ou décroissant (d'accord ?). Tout le problème revient donc à détecter les sommets situés dans l'intervalle $[a, b]$. Mais ceci est facile si l'on suppose que l'on sait calculer les racines de la dérivée p' du polynôme p. Bref, nous sommes ramenés à programmer essentiellement quatre fonctions :

- une fonction `dichotomie(p,a,b,h)` chargée de vérifier s'il existe une racine dans $[a, b]$ sur lequel p est supposé strictement monotone.
- une fonction `derive(p)` retournant le polynôme dérivé de p. Simple petit traitement de liste.
- une fonction `racines(p,a,b,h)` cherchant la liste de toutes les racines de p dans $[a, b]$, en faisant appel à `extrema`.
- une fonction `extrema(p,a,b,h)` cherchant la liste de tous les extrema de p dans $[a, b]$, en faisant appel à la fonction `racines`.

Ces deux fonctions `racines` et `extrema` qui s'invoquent mutuellement forment ce que l'on nomme une **co-récurrence** ou **récurrence mutuelle**. Il s'agit d'un phénomène qui n'est pas rare, ni en programmation ni en mathématique. Deux exemples amusants :

```
def est_pair(n) :
    if n == 0 : return True
    return est_impair(n-1)

def est_impair(n) :
    if n == 0 : return False
    return est_pair(n-1)
```

$$U_0 = 1$$
$$U_n = U_{n-1} + V_{n-1}$$
$$V_0 = 2$$
$$V_n = V_{n-1} - U_{n-1}$$

Que vaut V_3 ?

2 Implémentation de la technique

Nous programmons en Python. Je ne suivrai pas exactement l'ordre des questions du cc2. Vous trouverez la correction dans `cc2-gr4.py`. Nous travaillons sur des **polynômes denses**[4] représentés suivant les puissances croissantes (cf cours 7 page 11) par des listes contenant *tous* les coefficients, même nuls. Si le polynôme est de degré d, la liste sera de longueur $d + 1$. Donc :

```
def degre(p) :           # p est un polynôme dense
    return len(p) - 1

>>> p1 = [-2,0,6,-3]     # P_1 = -2 + 6x^2 - 3x^3
>>> degre(p1)
3
```

Le **polynôme dérivé** s'obtient en s'appuyant (sans la faire tomber) sur la formule $(ax^n)' = nax^{n-1}$ en évitant le cas $n = 0$, ce qui diminuera de 1 la longueur de la liste `p`. Le même travail se faisant sur tous les monômes, une compréhension de liste s'impose :

```
def derive(p) :          # le polynôme dérivé
    return [i*p[i] for i in range(1,len(p))]

>>> derive(p1)
[0,12,-9]                # 12x - 9x^2
```

[4] On aurait aussi pu utiliser le module `polycreux`.

Nous aurons besoin de la **valeur d'un polynôme** `p` en un point `x`. Nous sommes pressés, utilisons le *mauvais algorithme* immédiat à programmer, consistant à calculer toutes les puissances :

```
19 def valeur(p,x) :                # valeur de p en x
20     return sum(p[i]*x**i for i in range(len(p)))

21 >>> valeur(p1,-1))
22 7
```

Vous voudrez sans doute vous documenter sur le *schéma de Hörner* qui est l'algorithme optimal pour calculer la valeur d'un polynôme dense en un point, et que nous avons utilisé dans l'exercice de TD 3.2d pour passer du binaire au décimal par exemple. Il donnerait ici sans aucun calcul de puissances :

```
23 def valeur(p,x) :                 # avec un schéma de Horner
24     res = 0
25     for i in range(1,len(p)+1) :   # parcours droite -> gauche de p
26         res = res * x + p[-i]
27     return res
```

Ah, la fameuse **dichotomie** ! On se donne un polynôme `p` strictement monotone dans un intervalle $[a, b]$, et une précision h. Nous voulons retourner `False` s'il n'a pas de racine, et une racine approchée s'il en a une (et alors il n'en a qu'une).

```
1  def dichotomie(p,a,b,h) :         # p est strictement monotone sur [a,b]
2      ya = valeur(p,a) ; yb = valeur(p,b)
3      # il y a une racine ssi ya et yb sont de signes contraires !
4      if ya * yb < 0 :    # il y a une seule racine dans ]a,b[. Cherchons-la.
5          while True :    # recherche par dichotomie ! Voir hoering.py
6              m = (a + b) / 2
7              if abs(valeur(p,m)) < h : return m
8              if valeur(p,a) * valeur(p,m) < 0 : b = m
9              else : a = m
10     else : return False

11 >>> dichotomie(p1,-1,0,0.01)      # recherche d'une racine simple dans entre -1 et 0
12 -0.515625
13 >>> dichotomie(p1,0,1,0.01)       # entre 0 et 1. Voir la figure 1 ci-dessous
14 0.55859375
15 >>> dichotomie(p1,1.5,2,0.01)     # entre 1.5 et 2
16 False
17 >>> dichotomie(p1,-2,-0.7,0.01)   # entre -2 et -0.7
18 -1.942236328125
```

Maintenant, visons le cœur du problème et supposons déjà rédigée la fonction `racines(p,a,b,h)` retournant la liste des racines de `p` dans $[a, b]$. Programmons alors la fonction `extrema(p,a,b,h)` en calculant les racines du polynôme dérivé `p'`. On considère les extrémités `a` et `b` comme des extrema. Trop facile :

```
19 def extrema(p,a,b,h) :
20     if degre(p) < 1 : return []   # oups, p est constant !
21     return [a] + racines(derive(p),a,b,h) + [b]

22 # je ne peux pas tester extrema avant d'avoir défini la fonction racines...
```

Il ne reste plus qu'à porter l'estocade, et programmer la fonction `racines` qui va calculer les extrema $a = x_0, x_1, x_2, \ldots, x_{n-1}, x_n = b$, puis chercher par dichotomie s'il y a une racine dans chacun des sous-intervalles $]x_i, x_{i+1}[$ sur lesquels `p` est strictement monotone.

```
def racines(p,a,b,h) :
    if degre(p) == 0 :  return []            # p est constant non nul, pas de racine
    L = extrema(p,a,b,h)                     # la liste des extrema dans [a,b]
    res = []
    for i in range(len(L)-1) :
        essai = dichotomie(p,L[i],L[i+1],h)  # tentative entre deux extrema consecutifs
        if essai : res.append(essai)         # on a trouvé une racine !
    return res

>>> p2 = [-150,35,-121,24,30,-11,1]          # P_2 = (x+2)(x-3)(x^2+1)(x-5)^2
>>> extrema(p2,-4,6,0.01)                   # Voir la figure 2 ci-dessous
[-4,-1.3358803183113452, 0.153247393176247, 1.4679058622925836,
 3.881322276516486, 5.000013249671447,6]     # 5 sommets
>>> racines(p2,-10,10,0.001)
[-2.0000005476001776, 2.9999975003469297]    # deux racines simples entre -10 et 10
>>> racines(p2,4,10,0.001)
[]                                           # aucune racine simple entre 4 et 10
```

```
In[93]:= p2 = Expand[(x + 2) (x - 3) (x^2 + 1) (x - 5)^2]
Out[93]= -150 + 35 x - 121 x^2 + 24 x^3 + 30 x^4 - 11 x^5 + x^6
```

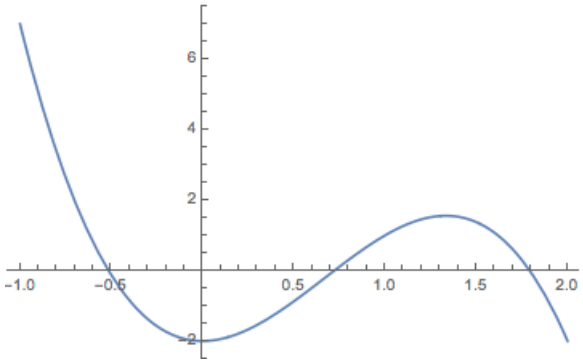

Figure 1

```
Plot[p2, {x, -5, 7}, PlotRange → {-350, 900}]
```

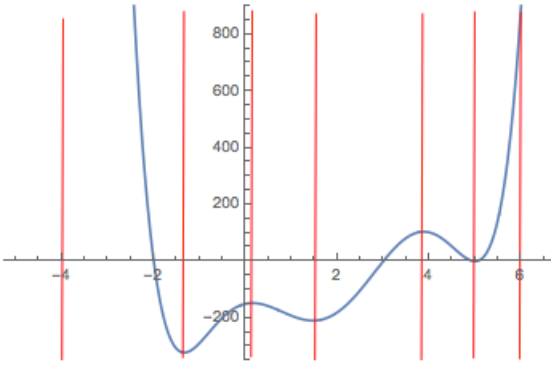

Figure 2

N.B. Les graphiques sont réalisés avec l'excellent logiciel **Mathematica®** *de Wolfram Research*[5] *dont vous pouvez obtenir la licence chaque année avec votre carte d'étudiant. Mathematica contient un langage de programmation sophistiqué spécialement conçu pour les mathématiques. La mise en page de ce document est faite en LaTeX, le standard de toute la communauté scientifique, ici via le logiciel* **TeXShop** *distribué gratuitement par Richard Koch*[6].

[5] https://www.wolfram.com/mathematica
[6] http://pages.uoregon.edu/koch/texshop