

TP d'informatique n°13

(Python Maths)

Utilisation de Python en arithmétique des polynômes

L'objectif du TP est de manipuler les listes Python pour application aux polynômes.

I Ecriture d'un polynôme, opérations sur les polynômes

Pour des raisons pratiques, un polynôme non nul $P = \sum_{k=0}^p a_k X^k$ donc avec $a_p \neq 0$, sera mémorisé dans Python par la liste des coefficients $[a_0, a_1, \dots, a_p]$. Le polynôme nul sera symbolisé par la liste $[0]$.

1. Ecrire une fonction `coeff` prenant comme argument une liste `LP` associée à un polynôme P et un entier naturel k , et qui renvoie le coefficient de degré k du polynôme. Attention si $k > \deg(P)$, `coeff(LP, k)` doit répondre 0.
2. Ecrire une fonction `normalise` prenant comme argument une liste `L` de flottants et qui rend la liste $[0]$ si `L` est la liste vide ou n'est constituée que de 0, et la liste `L` débarrassée de ces 0 finaux sinon. La liste retournée est alors une liste associée à un polynôme.
3. Ecrire une fonction `degre` prenant comme argument une liste `L` de flottants et qui rend -1 si le degré du polynôme associé à la liste `L` est nul et le degré de ce polynôme sinon.
4. Ecrire une fonction `SommePoly` prenant comme argument deux listes associées aux polynômes P et Q et qui rend la liste (normalisée) associée au polynôme $P + Q$. Calculer la complexité de cet algorithme en fonction du degré des polynômes.
5. Ecrire une fonction `MultiScalaire` prenant comme argument un flottant `a` et une liste `LP` associée au polynôme P et qui rend la liste (normalisée) associée au polynôme aP . (Attention à la multiplication par 0...)
6. Ecrire une fonction `MultiXk` prenant comme argument un entier naturel `n` et une liste `LP` associée au polynôme P et qui rend la liste (normalisée) associée au polynôme $X^n P$.
7. Ecrire une fonction `ProduitPoly` prenant comme argument deux listes associées aux polynômes P et Q et qui rend la liste (normalisée) associée au polynôme $P \times Q$. Calculer la complexité de cet algorithme en fonction du degré des polynômes. (Attention cette complexité devra tenir également compte de la complexité des fonctions écrites précédemment.)

II Arithmétique des polynômes

On va maintenant s'intéresser à l'arithmétique des polynômes. Pour ceux qui n'ont pas encore écrit les fonctions de la partie I, ils peuvent les télécharger sur la page "mathsmpsimarceau.jimbo.com"

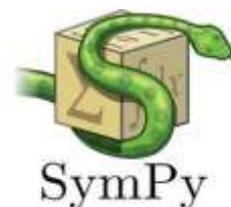
8. Ecrire une fonction `diveuclide` prenant comme argument deux listes LA et LB associées à deux polynômes A et B, et qui renvoie le booléen `False` si le polynôme B est nul et, sinon, le tuple (LQ, LR) où LQ et LR sont les listes normalisées des polynômes Q et R quotient et reste de la division euclidienne de A par B. Cette fonction utilisera les fonctions `coeff`, `SommePoly` etc... écrites dans la partie précédente.
9. Ecrire une fonction `PGCD` prenant comme argument deux listes LA et LB associées à deux polynômes A et B, et qui renvoie la liste LD associée au PGCD de A et de B.
10. On rappelle l'algorithme d'Euclide étendu :

```
Données :      a et b deux éléments d'un "anneau euclidien A" (comme  $\mathbb{Z}$  ou  $\mathbb{K}[X]$ )
Initialisation : u, v, r, u1, v1, r1  $\leftarrow$  1A, 0A, a, 0A, 1A, b
  Tant que r1  $\neq$  0A Faire
    q  $\leftarrow$  Quotient de la division euclidienne de r par r1
    u, v, r, u1, v1, r1  $\leftarrow$  u1, v1, r1, u - q * u1, v - q * v1, r - q * r1 (les - et * étant les
    opérations dans l'anneau A et les affectations sont, ici, simultanées)
  Fin Faire
Retour :      (r, u, v)
```

Ecrire une fonction `PGCD_etendu` prenant comme argument deux listes LA et LB associées à deux polynômes A et B, et qui renvoie le tuple (LD, LU, LV) où les listes LD, LU et LV sont les listes associées respectivement au PGCD de A et de B, et aux polynômes U et V de Bézout de degré minimal vérifiant $AU + BV = \text{PGCD}(A,B)$.

III Amélioration : utilisation de SymPy

Des développeurs ont déjà répondu à cette recherche des PGCD et des polynômes de Bezout.



11. Après avoir chargé le module `sympy`, exécuter dans l'interpréteur :

```
>>> x = symbols('x')
>>> f, g = 2 * x**2 - 2*x - 4, 2*x - 4
```

Tester sur f et g les fonctions du module `sympy` suivantes : `quo`, `gcd`, `rem`, `div`, `gcdex`.

Que font ces fonctions ?

Effectuer les tests avec les polynômes $(2X - 3, 2X - 3.00000001)$, $(12X^2 + 4X, 16X^2)$, 3 autres couples avec des polynômes de degré de plus en plus grand...

12. La fonction `clock` du module `time` donne "la valeur de l'horloge interne" au moment de l'appel. Comparer les durées d'exécution des fonctions `PGCD_etendu` et `gcdex` sur plusieurs exemples.