DS 2

Informatique

Racines de polynômes à coefficients entiers

Simon Dauguet
simon.dauguet@gmail.com

Mercredi 26 Novembre 2025

Le devoir dure 2h.

La qualité de la rédaction et de la présentation seront prises en compte dans la notation. On prendra bien garde à la justesse et la précision des justifications.

Si un candidat repère ce qui lui semble être une erreur d'énoncé, il l'identifiera clairement sur la copie et explicitera les décisions qu'il sera amené à prendre.

La calculatrice n'est pas autorisée.

Le sujet comporte 3 pages.

Vous pouvez utiliser toutes fonctions préalablement définies ainsi que les fonctions natives python telles que `len()`, `max()`, `min()`, etc.

Dans ce sujet, on propose une étude numérique des polynômes à coefficients entiers.

Ainsi le polynôme $P(X) = a_0 + a_1X + a_2X^2 + a_3X^3$ sera noté par le tableau `[a0, a1, a2, a3]`. Attention, le dernier coefficient pourra être nul mais le tableau ne sera jamais vide (même pour le polynôme nul).

1. Déterminer le nombre 173 en binaire et déterminer l'écriture décimale de $\overline{0110\ 1101}^2$.
2. Quel polynôme est défini par `[1, -2, 1, 0]` ? Définir sous ce format le polynôme $P(X) = X^3 - 2X + 4$.
3. On considère la fonction

```
def evaluation(P : list, x : float) -> float :
    som = P[0]
    for i in range( 1, len(P) ) :
        som += P[i] * x**i
    return som
```

   Que renvoie l'appel `evaluation([1, -2, 1, 0], 1)` ?

4. Proposer une fonction `estNul(P : list) -> bool`, qui précise si P est le polynôme nul.
5. Proposer une fonction `degre(P : list) -> int or float`, qui renvoie le degré du polynôme `P`.
   Rappel : en python : l'infini s'écrit `float('inf')`.
6. Proposer une fonction `simplifier(P : list) -> list`, qui renvoie une copie de `P` sans les 0 inutiles des coefficients a_i nuls où $i > degre(P)$. Le tableau `[0]` sera renvoyé pour le polynôme nul.
   Remarque : Alors, pour tout polynôme simplifié non nul, `len(P) = degre(P) + 1`.

7. Proposer une fonction `oppose(P : list) -> list`, qui renvoie l'opposé de P. Le polynôme renvoyé aura été simplifié.
8. Proposer une fonction `somme(P : list, Q : list) -> list`, qui effectue la somme des polynômes P et Q. Le polynôme renvoyé aura été simplifié.

   Par exemple, `somme([1,1,1], [-1,1,-1])` renverra `[0,2]`, tandis que `somme([1,0,0], [-1,1])` renverra `[0,1]`.
9. Rappel : le produit des polynômes $P(X) = \sum_{i=0}^{d} a_i X^i$ et $Q(X) = \sum_{i=0}^{e} b_i X^i$ s'obtient par $P(X) \times Q(X) = \sum_{i=0}^{d+e} \left(\sum_{j=0}^{i} a_j b_{i-j} \right) X^i$.

   En déduire un algorithme (on pourra penser à faire une copie des polynômes P et Q en les complétant d'autant de 0 que nécessaire) pour proposer une fonction `produit(P : list, Q : list) -> list`, qui effectue le produit (simplifié) des polynômes P et Q.

   Par exemple, `produit([0,0], [-1,1,-1])` renverra `[0]`, tandis que `produit([0,2], [1,3,0])` renverra `[0,2,6]`.
10. Proposer une fonction `deriver(P : list) -> list`, qui renvoie le polynôme dérivé de P.

On souhaite maintenant un encadrement d'une racine du polynôme P entre -10 et 10 si elle existe.

11. Naïvement, on décide de calculer $P(x)$ pour tout $x \in [\![-10, 10[\![$, si deux images sont de signes opposés on saura que le polynôme s'annule entre ces deux valeurs. Critiquer cette démarche.
12. Proposer une fonction `racinePossibles(P : list) -> list`, qui renvoie la liste de tous les tuples (a, b) de sorte que $(a, b) \in [\![-10, 10[\![^2$ avec $b = a$ si $P(a) = 0$ et $b = a+1$ si $P(a)P(a+1) < 0$. Par conséquent, on aura toujours une racine dans $[a, b[$.

    Par exemple, pour $P = (X+1)(X-2)$, `racinePossibles([-2,-1,1])` renverra `[(-1,-1),(2,2)]` tandis que pour $P = 32\left(X + \frac{5}{8}\right)\left(X - \frac{1}{4}\right)$, `racinePossibles([-5,12,32])` renverra `[(-1,0),(0,1)]`.
13. On suppose que $a = b$ ou $P(a)P(b) < 0$, et on considère le code suivant qui renvoie par dichotomie un intervalle $[a, b[$ dans lequel une racine existe :

```
def racine(P : list, a : float, b : float, epsilon : float = 2**-3 ) -> tuple :
    while b-a > epsilon :
        c = ( ..... ) / 2
        Pc = evaluation( P, c )
        if Pc == 0 :
            return ...
        if evaluation( P, a ) * Pc < 0 :
            ...
        else :
            ...
    return a, b
```

    Compléter le code.
14. En déduire une fonction `racines10(P, epsilon : float) -> list`, qui renvoie une liste de tuple `(a,b)` de longueur inférieure à `epsilon` sur lesquels la fonction polynomiale s'annule ou change de signe.
15. On souhaite tester la robustesse de notre code sur le polynôme $Q(X) = X - 1 - 4^{-k}$. Proposer un polynôme P ayant les mêmes racines que Q mais à coefficients entiers.
16. Souhaitant gagner en précision, on décide de faire l'appel `racine( P, 1, 2, 4**-k )` et on obtient (avec des flottant 32bits) le tableau ci-contre :

    Écrire le nombre `1+4**-k` en flottant 32bits. Expliquer alors le phénomène.
    On rappelle : en 32bits, on a $flt = (-1)^s(1+f) \times 2^{e-127}$

k	racine(P, 1, 2, 4**-k)
10	[1.000001, 1.000001]
11	[1.0000002, 1.0000002]
12	[1.0, 1.0]
13	[1.0, 1.0]

17. La fonction **`evaluation()`**, précédemment codée, a pu l'être de manière non optimale.

    En effet, pour le polynôme $P(X) = X^3 + 2X^2 - X + 1 = 1 - X + 2 \cdot X \cdot X + X \cdot X \cdot X$, on remarque que le calcul naïf demande 7 opérations tandis que $P(X) = 1 + X \cdot (-1 + X \cdot (2 + X \cdot (1)))$ requiert 6 opérations (en effet, en factorisant on ne calcule qu'une seule fois X^2, il s'agit de la méthode de Horner).

    Proposer une fonction d'évaluation plus optimisée, **`evaluationOpti(P : list, x : float) -> float`**, qui met en œuvre cette remarque.

18. Enfin on souhaite effectuer la division euclidienne d'un polynôme P par un polynôme D (*i.e.* on souhaite obtenir deux polynômes Q et R de sorte que $P(X) = D(X)Q(X) + R(X)$ et tels que $\deg(R) < \deg(D)$). Compléter la fonction **`divisionEuclidenne(P : list, D : list) -> tuple`**, qui renvoie un tuple contenant le polynôme quotient ainsi que le polynôme reste de cette division, à coefficients rationnels.

```
1  def divisionEuclidienne(P : list, D : list) -> tuple :
2      d, e = degre(P), degre(D)    # D != [0]
3      R = simplifier( P )          # polynôme reste
4      Q = max(1, d-e+1 ) * [ 0 ]   # polynôme quotient
5      if d < e :
6          return ...
7      cD = simplifier( D )         # copie de D sans 0 inutile
8      while d >= ... :
9          Q[ d-e ] = ...
10         nQ = produit( cD, ( d-e ) * [0] + [ -Q[ d-e ] ] )
11         R = somme( R, ... )
12         d = ...
13         return ...
```

Par exemple, pour $D(X) = X + 1$ on a $P(X) = 3X^2 + 3X + 3 = (X + 1) \times 3X + 3$, ainsi l'appel de **`divisionEuclide([3,3,3],[1,1])`** renverra **`([0,3.0],[3])`**.

Le résultat obtenu sera-t-il toujours exact ? Expliquer.