DS 2
Informatique
Racines de polynômes à coefficients entiers
Correction

Simon Dauguet
simon.dauguet@gmail.com

Mercredi 26 Novembre 2025

Si le contexte le permet, on nommera indifféremment le polynôme que la représentation associée sous forme de tableau python.

1. On décompose $173 = 128+32+8+4+1 = \overline{1010\ 1101}^2$ tandis que $\overline{0110\ 1101}^2 = 2^6+2^5+2^3+2^2+2^0 = 109$.
2. `[1, -2, 1, 0]` définit le polynôme $1-2X+X^2$.

Le polynôme $P = X^3-2X+4$ est, quant à lui, défini par `[4,-2,0,1]`.

3. L'appel `evaluation( [1, -2, 1, 0], 1)` renvoie l'image de 1 par le polynôme $1-2X+X^2$, soit 0.
4. On met en œuvre un algorithme de type recherche du premier coefficient non-nul.

```
1  def estNul(P : list) -> bool :
2      """
3  >>> estNul([0,0,0])
4  True
5  >>> estNul([1,2,0])
6  False
7      """
8      for i in range( len(P) ) :
9          if P[i] != 0 :
10             return False
11     return True
```

5. Une fois le polynôme nul traité, il s'agit d'un algorithme de type recherche du premier coefficient non-nul, en partant de la fin cette fois.

```
1  def degre(P : list) -> int or float :
2      """
3  >>> degre([0,0,0,0])
4  -inf
5  >>> degre([0,0,1,0])
6  2
7  >>> degre([1])
8  0
9  >>> degre([1,0])
10 0
11     """
12     if estNul(P) : return -float('inf')
13     for i in range( len(P)-1, -1, -1) :
14         if P[i] != 0 :
15             return i
16     return 0
```

Ou bien directement,

```
def degre(P : list) -> int or float :
    for i in range( len(P)-1, -1, -1) :
        if P[i] != 0 :
            return i
    return -float('inf')
```

6. Excepté pour le polynôme nul, on ne garde que les coefficients d'indices inférieurs au degré de P.

```
def simplifier(P : list) -> list :
    """
>>> simplifier( [0,0,2,3,0,0,0] )
[0, 0, 2, 3]
>>> simplifier( [0,0,0,0,0] )
[0]
    """
    d = degre( P )
    if d == -float('inf') :
        return [0]
    return P[:d+1]
```

7. On définit par compréhension l'opposé de P puis on le simplifie.

```
def oppose(P : list) -> list :
    """
>>> oppose( [-1,-2,1,0] )
[1, 2, -1]
    """
    return simplifier( [ -ai for ai in P ] )
```

8. Le polynôme résultant de la somme est de degré au plus le maximum des degrés de P et Q.

Ainsi, on effectue la somme des coefficients jusqu'au plus petit des degrés des polynômes, puis on complète avec les coefficients associés aux monômes de plus grand degré. On simplifie ensuite le polynôme résultant.

```
def somme(P : list, Q : list) -> list :
    """
>>> somme([1,0,0], [-1,1])
[0, 1]
>>> somme([1,1,1], [-1,-1,-1])
[0]
>>> somme([1,1,1], [-1,1,-1])
[0, 2]
>>> somme([1,0,1,1], [1,2,-1,1])
[2, 2, 0, 2]
    """
    S = [ P[i]+Q[i] for i in range( min( len(P), len(Q) ) ) ]
    if len(P) > len(Q) :
        S += P[ len(Q): ]
    else :
        S += Q[ len(P): ]
    return simplifier( S )
```

9. On utilise la remarque faite à la question 6, à savoir $len(P) = degre(P) + 1$. Ainsi si $S = PQ$, puisque $deg(S) = deg(P) + deg(Q)$, on définit la longueur du polynôme résultant **`lenS`**, et on complète les polynômes par autant de zéros que nécessaires.

On effectue alors le produit avant de simplifier le polynôme résultant.

```python
def produit(P : list, Q : list) -> list :
    """
>>> produit([1,1,1,0,0,0], [0,0])
[0]
>>> produit([1,1,1,0,1,0], [3,0])
[3, 3, 3, 0, 3]
>>> produit([1,1,1,0,0,0], [0,1,1,0])
[0, 1, 2, 2, 1]
    """
    if estNul(P) or estNul(Q) : return [0]
    lenS = degre(P) + degre(Q) + 1
    cP = P[:] + [0] * (lenS - len(P))
    cQ = Q[:] + [0] * (lenS - len(Q))
    S = lenS * [0]
    for i in range( lenS ) :
        for k in range( i+1 ) :
            S[i] += cP[i-k] * cQ[k]
    return S
```

10. Il s'agit d'un algorithme de type somme (ici l'énoncé suppose que P est donné simplifié, sa dérivé le sera donc aussi).

```python
def deriver(P : list) -> list :
    """
>>> deriver([0])
[0]
>>> deriver([5,0])
[0]
>>> deriver([1,1,1,1])
[1, 2, 3]
    """
    if degre(P) < 1 :
        return [0]
    return [ i*P[i] for i in range(1,len(P)) ]
```

11. Puisque la fonction polynomiale P est continue sur $\mathbb{R}$, $P(a)P(b) \leq 0$ nous assure l'existence d'une racine dans $[a, b]$ mais il peut en exister plusieurs. De plus, on peut très bien avoir $P(a)P(b) > 0$ et avoir une racine dans $[a, b]$.

12. On parcourt de -10 à 10 exclu l'intervalle et on vérifie si `i` est une racine (`P(i)=0`) ou si le produit est strictement négatif.

```python
def racinePossibles(P : list) -> list :
    """
>>> racinePossibles([-2,-1,1])
[(-1, -1), (2, 2)]
>>> racinePossibles([-5,48,32])
[(-1, 0), (0, 1)]
    """
    intervalles = []
    for i in range( -10, 10 ) :
        Pi = evaluation(P, i)
        if Pi == 0 :
            intervalles += [ (i, i) ]
        if  Pi * evaluation(P, i+1) < 0 :    # inefficace
            intervalles += [ (i, i+1) ]
    return intervalles
```

Remarque : on calcule deux fois les évaluations pour $i \in [\![-9, 9]\!]$.

13. On complète l'algorithme de recherche dichotomique proposé.

```python
def racine(P : list, a : float, b : float, epsilon  : float = 2**-3) -> tuple :
    """
>>> racine([-2,-1,1], 2, 2)
[2, 2]
>>> racine([-5,12,32], 0, 1)
[0.25, 0.25]
>>> racine([-5,12,32], -1, 0)
[-0.625, -0.625]
    """
    while b-a > epsilon :
        c = ( a + b ) / 2                          # complété
        Pc = evaluation( P, c )
        if Pc == 0 :
            return [ c, c ]                 # complété
        if evaluation(P, a) * Pc < 0 :
            b = c      # complété
        else :
            a = c                                  # complété
    return a, b
```

14. Chacun des couples fournis par **racinePossibles()** sera amélioré à l'aide de la fonction **racine()**, ce qui donne :

```python
def racines10(P, epsilon : float = 2**-3) -> list :
    """
>>> racines10([-5,12,32])
[[-0.625, -0.625], [0.25, 0.25]]
>>> racines10( produit([-5,12,32], [-2,-1,1]) )
[[-1, -1], [0.25, 0.25], [2, 2]]
    """
    intervalles = racinePossibles( P )
    racines = []
    for a,b in intervalles :
        racines += [ racine(P, a, b, epsilon) ]
    return racines
```

15. $P = 4^k Q \in \mathbb{Z}[X]$, ainsi on propose **[-1-4**k, 4**k]**.

16. $1 + 4^{-k} = +2^0(1 + 2^{-2k})$

Ainsi

- $s = 0$,
- $p = 0$, donc $e = 127 + p = \overline{0111\ 1111}^2$,
- $f = 2^{-2k} = \overline{\underbrace{000\ldots01}_{2k-1 \text{ zéros}}0\ldots0}^2$

d'où l'écriture $0'0111\ 1111'000\ldots010\ldots0$.

Ainsi puisque la partie fractionnaire comporte au plus 23 bits, tant que $k \leq 11$ le nombre $1+4^{-k}$ s'écrira de manière exacte en flottant 32 bits. Dès que $k > 11$, le 1 représentant 2^{-2k} dans la partie fractionnaire sera tronqué et le flottant $1 + 2^{-2k}$ sera égal à 1. Ce que l'on observe sur le tableau fourni.

17. On propose le code

```python
def evaluationOpti(P : list, x : float) -> float :
    """
>>> evaluationOpti( [1, -2, 1, 0], 1)
0
>>> evaluationOpti( [1, -2, 1, 0], 0)
1
>>> evaluationOpti( [1, -2, 1, 0], -1)
4
    """
    som = 0
    for i in range( len(P)-1, -1, -1 ) :
        som = x * som + P[i]
    return som
```

18. La division euclidienne de $P = \sum_{i=0}^{d} p_i X^i$ par $D = \sum_{i=0}^{e} d_i X^i$ se calcule par :

$R = P$
TANT QUE $deg(R) \geqslant e$
  ajouter $\frac{p_{deg(R)}}{d_e} X^{deg(R)-e}$ à Q
  $nQ = \frac{p_{deg(R)}}{d_e} X^{deg(R)-e} \times D$
  retrancher nQ à R
renvoyer Q, R

Ce qui donne :

```
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 ( [0], R )     # complété
7      cD = simplifier( D )         # copie de D sans 0 inutile
8      while d >= e :                   # complété
9          Q[ d-e ] = R[-1] / cD[-1]    # complété
10         nQ = produit( cD, ( d-e ) * [0] + [ -Q[ d-e ] ] )
11         R = somme( R, nQ )           # complété
12         d = degre(R)                 # complété
13     return Q, R                      # complété
```

Le résultat peut ne pas être exact. En effet, la division effectuée dans la boucle `while` peut induire des erreurs d'arrondis qui peuvent entraîner des inexactitudes.

Par exemple, `divisionEuclidienne([1, 3], [3])` renvoie `([0.3333333333333333, 1.0], [0])`.

Remarque : avec d'autres valeurs, la fonction peut ne pas s'arrêter (le degré du reste `R` devenant constant à cause des erreurs de calculs).