Chapitre (ALGO) 6 Polynômes

Résumé & Plan

L'objectif de ce TP est d'apprendre à représenter en machine un polynôme, et de coder en Python certaines opérations élémentaires simples.

- Les chapitres d'Informatique sont composés de cours et d'exercices intégrés. Le cours sera projeté au tableau.
- Il n'est pas attendu que toute la classe aborde tous les exercices. Traitez donc en priorité les exercices présents dans la liste donnée à chaque début de séance.
- Exercices 💣 / **Pour aller plus loin** : exercices plus difficiles, ou plus techniques. À ne regarder que si les autres sont bien compris.

Fichier externe ?

NON *pas de fichier externe dans ce TP*

1. GÉNÉRALITÉS

1.1. Représentation informatique d'un polynôme

Comment coder un polynôme en Python ? Puisqu'il est entièrement déterminé par la donnée de ses coefficients, il suffit par exemple de les ranger par ordre croissant de degré et considérer en convention que le polynôme nul correspond à la liste vide. Par exemple, le polynôme $P = X^2 + 2$ peut être codé par la liste `P = [2, 0, 1]`.

Un polynôme de degré n est donc représenté par une liste de taille $n+1$

Exercice 1 | **Quelques exemples** [Solution]

1. Écrire (sur le polycopié) la liste correspondant aux polynômes ci-après :
   - $P_1 = X^3 - X^4 + \pi X^2 - \ln(2)$,
   - $P_2 = (X-2)^3 + (5-X)^2$.

2. Écrire une fonction `calcul_P3(n)` (n étant un entier positif), et qui renvoie la liste correspondante au polynôme : $P_3 = \sum_{k=0}^{n} \frac{(-X)^k}{k!}$.
3. Écrire une fonction `calcul_P4(n)` (n étant un entier strictement positif), et qui renvoie la liste correspondante au polynôme : $P_4 = \sum_{k=1}^{n} \frac{(-X^2)^k}{\sqrt{k}}$.

1.2. Quelques fonctions utiles

Exercice 2 | **Fonction degré** [Solution] Écrire une fonction `degre(P)`, qui à un polynôme `P` (représenté sous forme de liste) renvoie son degré. Pour le cas du polynôme nul, on pourra coder l'infini par l'instruction suivante :

```
Inf = float('inf') # correspond à l'infini des maths
```

Testez votre fonction sur les polynômes P_1, P_2 et P_3, P_4 par exemple pour $n = 2$.

Exercice 3 | **Un coefficient** [Solution] Compléter la fonction `coeff(P, i)` qui renvoie le coefficient devant X^i du polynôme P :

```
def coeff(P, i):
    n = degre(P)
    if _______ <= i <= ________ :
        return ________
    else:
```

```
    return _________
```

Exercice 4 | **Deux opérations** [Solution]

1. Écrire une fonction `mult_X(P)` qui renvoie le polynôme XP sous forme de liste.
2. Écrire une fonction `derive(P)` qui renvoie le polynôme P′ sous forme de liste.

1.3. Évaluer un polynôme

Une opération plutôt attendue est celle de pouvoir l'évaluer en un flottant. Plusieurs techniques sont possibles, toutes détaillées dans l'exercice qui suit.

Exercice 5 | [Solution] Dans tout l'exercice, vous testerez vos fonctions sur $P = X^2 - X + 2$ et $x = 2$. On départagera les méthodes employées pour calculer $P(x)$ en comptant le nombre de multiplications effectuées (coûteuses). On supposera que la commande `x**k` effectue k multiplications dans son calcul.

1. Codez le polynôme P sur votre machine.
2. **[Méthode naïve]** Un(e) étudiant(e) propose la fonction suivante :

```
def eval_poly1(P, x):
    N = len(P)
    S = P[0]
    for k in range(1, N):
        S = P[k]*(x**k)
    return S
```

Cette fonction vous semble-t-elle correcte ? Fonctionne-t-elle pour le polynôme nul ? La reprendre sur votre machine en corrigeant les éventuel(s) problème(s) précédents. Combien il y a-t-il exactement de multiplications effectuées ? *On exprimera ce nombre en fonction de $n = len(P)$.*

3. **[Méthode plus élaborée]** Un(e) autre étudiant(e) remarque que lorsqu'on connaît x^{k-1}, une seule multiplication supplémentaire est nécessaire pour obtenir x^k. Proposer une autre version de la fonction précédente, nommée `eval_poly2`, tenant compte de cette amélioration. Vérifier que le nombre de multiplications est maintenant de l'ordre de $2n$.

4. **[Méthode encore plus élaborée 💣]** L'algorithme de HÖRNER repose sur le principe suivant. On note $P = \sum_{k=0}^{n} a_k X^k$,
   - si P est le polynôme nul (correspondant à la liste vide), le calcul de $P(x)$ est immédiat, on renvoie 0.
   - Sinon, il suffit de remarquer que :
   $$P(x) = \underbrace{((a_n x^{n-1} + \ldots + a_2) \times x + a_1)}_{\text{même factorisation « à la Hörner », mais sur } \sum_{k=1}^{n} a_k X^k} \times x + a_0.$$
   Sur un polynôme de degré 3, cela consiste à l'écrire de la manière suivante :
   $$ax^3 + bx^2 + cx + d = ((a \times x + b) \times x + c) \times x + d.$$

   Écrire une fonction récursive `eval_poly3(P, x)` permettant le calcul de $P(x)$ via l'algorithme de HORNER. *On peut montrer que le nombre de multiplications est sensiblement le même que dans la question précédente.*

2. APPLICATION À UNE FAMILLE DE POLYNÔMES

Exercice 6 | [Solution] On définit la suite de polynômes $(P_n)_{n\in\mathbb{N}^\star}$ par $P_1 = X$ et pour tout $n \in \mathbb{N}^\star$: $P_{n+1} = X^2 P_n' + 1$.

1. En utilisant des fonctions des exercices précédents, écrire une fonction `calcul_P(n)` qui renvoie P_n sous forme de liste, où $n > 0$ est un entier. Tester pour $n \in \{2, 3\}$ et vérifier les polynômes obtenus par le calcul.

2. Que peut-on conjecturer sur $\deg(P_n)$ et $\mathrm{dom}(P_n)$ pour tout $n \in \mathbb{N}^\star$?

3. Tracer le graphe de P_{10} sur l'intervalle $[-40, 40]$.

3. SOLUTIONS DES EXERCICES

Solution (exercice 1) [Énoncé]

1. En réordonnant et en développant, on a les expressions suivantes :

$$P_1 = -X^4 + X^3 + \pi X^2 + 0X - \ln(2), \quad P_2 = X^3 - 5X^2 + 2X + 17.$$

D'où les deux listes ci-après :

```
>>> P_1 = [-ma.log(2), 0, ma.pi, 1, -1]
>>> P_2 = [17, 2, -5, 1]
```

2. On a $P_3 = \sum_{k=0}^{n} \frac{(-1)^k}{k!} X^k$.

```
def calcul_P3(n):
    facto = 1 # stockage de la factorielle
    L = [1/facto]
    for k in range(1, n+1):
        facto *= k
        L.append((-1)**k/facto)
    return L
```

```
>>> P3_4 = calcul_P3(4)
>>> P3_4
[1.0, -1.0, 0.5, -0.16666666666666666, 0.041666666666666664]
```

3. On a $P_4 = \sum_{k=1}^{n} \frac{(-1)^k}{\sqrt{k}} X^{2k}$. La différence avec la question d'avant c'est qu'il faut distinguer les indices pairs des indices impairs. Voici une première version naïve :

```
def calcul_P4(n):
    L = [0] # pas de terme constant, on commence donc par zéro
    for k in range(1, 2*n+1):
        if k%2 == 0:
            L.append((-1)**(k//2)/ma.sqrt(k//2))
        else:
            L.append(0)
    return L
```

```
>>> P4_4 = calcul_P4(4)
>>> P4_4
[0, 0, -1.0, 0, 0.7071067811865475, 0, -0.5773502691896258, \
↪ 0, 0.5]
```

On peut se payer le luxe d'économiser les tests de parité, en partant d'une liste nulle de la bonne taille puis en complétant directement les bons coefficients.

```
def calcul_P4(n):
    L = [0]*(2*n+1)
    for k in range(2, 2*n+1, 2): # on avance de 2 en 2
        L[k] = (-1)**(k//2)/ma.sqrt(k//2)
    return L
```

```
>>> P4_4 = calcul_P4(4)
>>> P4_4
[0, 0, -1.0, 0, 0.7071067811865475, 0, -0.5773502691896258, \
↪ 0, 0.5]
```

Solution (exercice 2) [Énoncé]

```
def degre(P):
    if len(P) == 0:
        return -Inf
    else:
        return len(P) - 1
```

```
>>> degre(P_1)
4
>>> degre(P_2)
3
>>> degre(P4_4)
8
>>> degre(P4_4)
8
```

Solution (exercice 3) [Énoncé]

```
def coeff(P, i):
    n = degre(P)
    if 0 <= i <= n:
        return P[i]
    else:
        return 0
```

```
>>> coeff(P_1, 4)
-1
>>> coeff(P_1, 3)
1
```

```
>>> coeff(P_1, 5)
0
```

Solution (exercice 4) [Énoncé]

1. Multiplier par X un polynôme se traduit, dans la liste, par un décalage des coefficients vers la droite et l'ajout d'un zéro devant (il n'y a alors plus de terme constant).

```
def mult_X(P):
    return [0] + P
```

Par exemple,

```
>>> mult_X(P_1)
[0, -0.6931471805599453, 0, 3.141592653589793, 1, -1]
```

2.

```
def derive(P):
    n = degre(P)
    P_der = []
    for i in range(1, n+1):
        P_der.append(i*coeff(P, i))
    return P_der
```

Par exemple,

```
>>> derive(P_1)
[0, 6.283185307179586, 3, -4]
```

Solution (exercice 5) [Énoncé]

1.

```
>>> P = [2, -1, 1]
>>> 2**2-2+2 # valeur attendue
4
>>> x = 2
```

2. La fonction ci-dessous

```
def eval_poly1(P, x):
    N = len(P)
    S = 0
    for k in range(N):
        S = P[k]*(x**k)
    return S
```

est incorrecte ; en effet, la valeur de S est à chaque fois perdue puisqu'on utilise un symbole =. De plus, elle ne fonctionne pas pour le polynôme nul (c'est-à-dire la liste vide), la seconde affectation renverra une erreur. On peut corriger ainsi :

```
def eval_poly1(P, x):
    n = len(P)
    S = 0
    for k in range(n):
        S += P[k]*(x**k)
    return S
```

```
>>> eval_poly1(P, x)
4
>>> eval_poly1([], x)
0
```

On a à chaque étape de la boucle `for` $k+1$ multiplications, donc au total $\sum_{k=1}^{n-1}(k+1) = \sum_{k=2}^{n} k = \frac{n(n+1)}{2} - 1$.

3. On peut cette fois recycler les anciennes puissances à l'aide d'une nouvelle variable X.

```
def eval_poly2(P, x):
    N = len(P)
    S = 0
    X = 1 # stocke les puissances de x, au début x^0
    for k in range(N):
        S += P[k]*X
        X *= x
    return S
```

```
>>> eval_poly2(P, x)
4
>>> eval_poly2([], x)
0
```

Cette fois-ci, on a seulement 2 multiplications à chaque étape de la boucle `for`, donc au total $\sum_{k=1}^{n-1} 2 = 2(n-1)$.

4.

```
def eval_poly3(P, x):
    N = len(P)
    if len(P) == 0:
        return 0
    else:
        return eval_poly3(P[1:], x)*x + P[0]
```

```
>>> eval_poly3(P, x)
4
>>> eval_poly3([], x)
```

```
0
```

Solution (exercice 6) [Énoncé]

1.
```
def calcul_P(n):
    P = [0, 1] # le polynôme P_1 = X
    for k in range(2, n+1):
        P = mult_X(mult_X(derive(P)))[:]
        # [:] : méthode pour faire une copie de liste en dur
        P[0] += 1 # dû au +1 dans la relation de récurrence
    return P
```

```
>>> calcul_P(2)
[1, 0, 1]
>>> calcul_P(3)
[1, 0, 0, 2]
>>> calcul_P(4)
[1, 0, 0, 0, 6]
```

2. On peut afficher plusieurs valeurs :

```
>>> for n in range(1, 10):
...     P_n = calcul_P(n)
...     print("Pour n = "+str(n)+"—deg :", degre(P_n), "dom :
", P_n[-1])
...
Pour n = 1—deg : 1 dom : 1
Pour n = 2—deg : 2 dom : 1
Pour n = 3—deg : 3 dom : 2
Pour n = 4—deg : 4 dom : 6
Pour n = 5—deg : 5 dom : 24
Pour n = 6—deg : 6 dom : 120
Pour n = 7—deg : 7 dom : 720
Pour n = 8—deg : 8 dom : 5040
Pour n = 9—deg : 9 dom : 40320
```

On peut donc raisonnablement conjecturer que :

$$\forall n \in \mathbb{N}^\star, \quad \boxed{\deg(P_n) = n, \quad \mathrm{dom}(P_n) = (n-1)!}.$$

3.
```
P10 = calcul_P(10)
X = np.linspace(-40, 40, 10**3)
Y = [eval_poly2(P10, x) for x in X]
plt.plot(X, Y, label="n=10")
plt.legend()
```