Correction
TD 17
Polynômes

Exercice 7. Factoriser les polynômes suivants.

$x \mapsto x^5 - 3x^4 + 2x^3 - x^2 + 3x - 2$ $\quad$ $x \mapsto x^5 - 2x^4 - 5x^3 + 3x^2 + 8x + 3$ (1)

$x \mapsto x^5 - 3x^4 + 5x^3 - 7x^2 + 6x - 2$ $\quad$ $x \mapsto x^4 - 8x^2 + 15$ (2)

$x \mapsto x^4 + 1$ $\quad$ $x \mapsto x^4 - x^2 + 1$ (3)

Correction. Réponse :

$(x-1)^2 \times (x-2) \times (x^2+x+1)$ $\quad$ $(x+1)^2 \times (x-3) \times (x^2-x-1)$

$(x-1)^3 \times (x^2+2)$

Pour $x^4 - 8x^2 + 15$ on pose $y = x^2$ et on trouve $(y-3) \times (y-5)$. Donc

$$x^4 - 8x^2 + 15 = (x-\sqrt{3})(x+\sqrt{3})(x-\sqrt{5})(x+\sqrt{5})$$

Pour $x^4 + 1$ on écrit $x^4 + 1 = (x^2+1)^2 - 2x^2 = (x^2+1)^2 - (\sqrt{2}x)^2$ puis par identité remarquable

$$x^4 + 1 = (x^2 - \sqrt{2}x + 1) \times (x^2 + \sqrt{2}x + 1)$$

De même, $x^4 - x^2 + 1 = (x^2+1)^2 - 3x^2$ donc

$$x^4 - x^2 + 1 = (x^2 - \sqrt{3}x + 1) \times (x^2 + \sqrt{3}x + 1)$$

□

Exercice 12. Soit P une polynôme réel. Soit $\alpha \in \mathbb{C}$.

1. Montrer que si α est une racine de P, alors son conjugué $\bar{\alpha}$ est aussi une racine de P.
2. Montrer que si α est une racine de P et que $\alpha \notin \mathbb{R}$ alors P se factorise par $(x-\alpha)(x-\bar{\alpha})$.
3. Montrer que si $\alpha \notin \mathbb{R}$ alors $(x-\alpha)(x-\bar{\alpha})$ est toujours un polynôme de degré 2 réel avec discriminant strictement négatif.
4. On admet que tout polynôme non nul (même à coefficients complexes !) admet au moins une racine dans $\mathbb{C}$. Montrer que tout polynôme non nul réel se factorise comme un produit de termes $x - \alpha$ ($\alpha \in \mathbb{R}$) et de polynômes de degré 2 avec discriminant strictement négatif.
5. Factoriser $x \mapsto x^4 - 1$ en tant que polynôme réel.

Correction.

1. On écrit $P : x \mapsto \sum_{k=0}^{n} a_k x^k$. Si $\alpha \in \mathbb{C}$ est racine alors $\sum_{k=0}^{n} a_k \alpha^k$. En conjugant toute cette expression d'un coup on trouve $\sum_{k=0}^{n} \overline{a_k \alpha^k} = 0$ soit $\sum_{k=0}^{n} \bar{a}_k \bar{\alpha}^k = 0$. Mais les coefficients sont réels donc $\bar{a}_k = a_k$. Finalement l'égalité $\sum_{k=0}^{n} a_k \bar{\alpha}^k = 0$ montre que $\bar{\alpha}$ est racine de P.
2. Si $\alpha \notin \mathbb{R}$ alors α et $\bar{\alpha}$ sont bien deux racines **distinctes** de P et donc on peut factoriser deux fois de suite, c'est à dire écrire d'abord $P(x) = (x-\alpha)Q_1(x)$ puis $\bar{\alpha}$ sera alors racine de Q_1 et donc $Q_1(x) = (x-\bar{\alpha})Q_2(x)$, ainsi on a factorisé P par $(x-\alpha)(x-\bar{\alpha})$.
3. $(x-\alpha)(x-\bar{\alpha}) = x^2 - (\alpha+\bar{\alpha})x + \alpha\bar{\alpha}$. Mais $\alpha + \bar{\alpha} = 2\,\mathrm{Re}(\alpha)$ est bien un réel et $\alpha\bar{\alpha} = |\alpha|^2$ aussi. Le discriminant est donc $(2\,\mathrm{Re}(\alpha))^2 - 4|\alpha|^2$. L'inégalité $(\mathrm{Re}(\alpha))^2 \leqslant |\alpha|^2$ a été démontrée (au moment de l'inégalité triangulaire dans $\mathbb{C}$) et c'est une égalité si et seulement si α est réel, donc pour $\alpha \notin \mathbb{R}$ le discriminant est bien strictement négatif.
4. Il suffit de résumer : ou bien P de degré $n \geqslant 1$ admet une racine α réelle, et donc on factorise $P(x) = (x-\alpha)Q_1(x)$ et Q_1 est réel de degré $n-1$. Ou bien P admet une racine complexe non réelle et donc on factorise P par un polynôme réel P_1 de degré 2 avec $\Delta < 0$ et un quotient Q_1 qui est cette fois de degré $n-2$.
   Attention car la factorisation passe par $\mathbb{C}$ et donc les quotients successifs peuvent devenir complexes... Mais le même type d'argument que dans la première question montre que si P est à coefficients réels et si $P(x) = P_1(x) \times Q_1(x)$ alors Q_1 est nécessairement à coefficients réels : on conjugant cette relation on trouve $\overline{P(x)} = \overline{P_1(x)} \times \overline{Q_1(x)}$ soit $P(x) = P_1(x) \times \overline{Q_1(x)}$ vrai pour tout $x \in \mathbb{R}$, et de plus comme le discriminant de P_1 est strictement négatif on n'a jamais $P_1(x) = 0$. Donc pour tout $x \in \mathbb{R}$, $\overline{Q_1(x)} = Q_1(x)$. Écrivant $Q_1 : x \mapsto \sum_{k=0}^{m} b_k x^k$ alors pour $x \in \mathbb{R}$, $\overline{Q_1(x)} = \sum_{k=0}^{m} \bar{b}_k x^k$. Donc si $Q_1(x) = \overline{Q_1(x)}$ pour tout $x \in \mathbb{R}$, l'unicité des coefficients de Q_1 nous dit que $\bar{b}_k = b_k$ donc les coefficients sont réels.
   Puis on continue en factorisant Q_1 et ainsi de suite.
5. C'est le polynôme dont les racines sont $1, -1, i, -i$ donc $x^4 - 1 = (x-1)(x+1)(x-i)(x+i)$. Or $(x-i)(x+i) = x^2+1$. Donc $x^4 - 1 = (x-1)(x+1)(x^2+1)$. *On peut aussi s'y prendre autrement en posant $y = x^2$ et en factorisant $y^2 - 1$.*

□

Exercice 13. *Pour s'entrainer en Python*
On représente le polynôme $P : x \mapsto \sum_{k=0}^{n} a_k x^k$ par la liste `P` de longeur $n+1$ telle

que `P[k]` est exactement le coefficient a_k. Le fait que la liste soit de longueur $n+1$ n'implique pas que P est de degré $n+1$: les coefficients peuvent très bien être nuls. Le polynôme nul est donc aussi bien la liste vide `[]` qu'une liste comme `[0, 0, 0]`. Le polynôme x est `[0, 1]` mais on peut toujours étendre la liste par des zéros.
Écrire les fonctions suivantes :

1. `degre(P)` : donne le degré de P *(on pourra retourner -1 pour le polynôme nul)*
2. `somme(P, Q)`, `produit_constante(P, a)`, `produit(P, Q)` : comme leur nom l'indique
3. `puissance(P, m)` : renvoie P^m
4. `evalue(P, a)` : prend un réel a et calcule $P(a)$
5. `compose(P, Q)` : calcule $Q \circ P$
6. `derive(P)` (comme son nom l'indique), puis `derive_r(P, r)` : dérivée r-ième, par une boucle ou bien récursivement !

Application : les polynômes de Legendre L_n sont définis par $L_n(x) = \frac{1}{n!}\left[(x^2-1)^n\right]^{(n)}$.
Écrire un programme qui calcule L_n.
Les polynômes de Tchebychev T_n sont définis par $T_0 = 1$, $T_1 = x$ et par la relation de récurrence $\forall n \in \mathbb{N}$, $T_{n+2} = 2xT_{n+1} - T_n$. Écrire un programme qui calcule T_n.

Correction. On convient de la notation suivante : les variables `n`, `m` sont utilisées pour représenter non pas les longueurs des listes, mais leur longueur moins un. C'est à dire le degré du polynôme (s'il n'y a pas des zéros en trop). Le coefficient dominant de `P` est alors `P[n]` et pour parcourir toute la liste il faut une boucle avec `range(n+1)` ; rappelons que sinon, pour une liste `L` de longueur `n`, le dernier élément est `L[n-1]` et qu'elle se parcourt en entier avec `range(n)`.
On notera toujours `R` le polynôme résultat, de degré `r`.

1.
```
def degre(P):
    # le degré de P ...
    n = len(P) - 1
    # ... sauf s'il y a des zéros *à la fin* :
    # dans ce cas diminuer le degré de 1
    while n >= 0 and P[n] == 0:
        n = n - 1
    return n
```

   Remarques :
   — Dans la boucle il faut aussi tester `n >= 0` sinon à la fin on testera `P[-1]` ...
   — Le polynôme nul n'est pas traité comme un cas à part ; vérifiez que dans tous les cas (si `P = []` alors au départ `len(P)` vaut 0 ; ou bien si `P = [0, 0, 0]`) il est naturel que la fonction retourne -1.

2. La plus facile est en fait `produit_constante` : multiplier tous les coefficients par `a`

```
def produit_constante(P, a):
    n = len(P) - 1
    R = [0 for _ in range(n+1)]
    for i in range(n+1):
        R[i] = a * P[i]
    return R
```

Le deuxième plus facile est `produit` : le coefficient de degré i de P, multiplié par le coefficient de degré j de Q, **contribue au** coefficient de degré $i+j$ de PQ.

```
def produit(P, Q):
    n = len(P) - 1
    m = len(Q) - 1
    # le résultat, polynôme nul de la bonne taille
    r = n + m
    R = [0 for _ in range(r+1)]
    # double boucle
    for i in range(n+1):
        for j in range(m+1):
            R[i+j] = R[i+j] + P[i] * Q[j]
    return R
```

Enfin le plus compliqué est la somme car il faut nécessairement gérer le cas où les deux listes n'ont pas la même taille.

```
def somme(P, Q):
    n = len(P) - 1
    m = len(Q) - 1
    if n >= m:
        # le résultat est de degré n
        R = [0 for _ in range(n+1)]
        # somme là où P, Q ont des coefficients communs
        for i in range(m+1):
            R[i] = P[i] + Q[i]
        # *puis* rajouter les coefficients de P tout seul
        for i in range(m+1, n+1):
            R[i] = P[i]
        return R
    else:
        ...
```

On a alors les choix suivants dans le `else` :

— Recopier la même chose en échangeant les rôles de `P` et `Q`, et de `n` et `m`,
— Ou bien s'en passer, mais dès le départ échanger les variables `P` et `Q` si $n < m$,
— Ou bien (grosse astuce) `return somme(Q, P)` ce qui appelle la fonction récursivement (mais une seule fois) en échangeant les rôles de `P` et `Q`.

3. Sans commentaire, itératif ou récursif. P^0 est le polynôme constant égal à 1 donc la liste `[1]`.

```
def puissance(P, m):
    R = [1]
    for _ in range(m):
        R = produit(R, P)
    return R
```

ou

```
def puissance(P, m):
    if m == 0:
        return [1]
    else:
        Q = puissance(P, m-1)
        return produit(P, Q)
```

4. C'est un calcul de somme, la somme des `P[i] * a**i` :

```
def evalue(P, a):
    n = len(P) - 1
    s = 0
    for i in range(n+1):
        s = s + P[i] * a**i
    return s
```

5. C'est comme évalue, car c'est le polyôme $\sum b_i P^i$ (si $Q(x) = \sum b_i x^i$), mais donc en utilisant les fonctions `somme` et `puissance` :

```
def compose(P, Q):
    m = len(Q) - 1
    R = []
    for i in range(n+1):
        R = somme(R, produit(Q[i], puissance(P, i)))
    return R
```

6. Attention aux indices, vérifiez bien sur les premiers termes...

```
def derive(P):
    n = len(P) - 1
    # le résultat est de degré n-1
    r = n - 1
    R = [0 for _ in range(r+1)]
    for i in range(r+1):
        R[i] = (i+1) * P[i+1]
    return R
```

Remarque : pour un polynôme constant on aura $n = 0$ puis $r = -1$ et la syntaxe créera automatiquement pour `R` une liste vide.

Pour la dérivée r-ième le principe est exactement le même que pour les puissances, avec boucle ou récursivement. La dérivée 0-ième de P est P lui-même.

```
def derive_r(P, r):
    R = P
    for _ in range(r):
        R = derive(R)
    return R
```

ou

```
def derive_r(P, r):
    if r == 0:
        return P
    else:
        R = derive_r(P, r-1)
        return derive(R)
```

7. Enfin on termine (sans trop de commentaires)

```
def Legendre(n):
    # polynôme x^2 - 1
    P = [-1, 0, 1]
    Q = puissance(P, n)
    R = derive(Q, n)
    S = produit_constante(R, 1/factoriel(n))
    return S
```

(à supposer qu'on ait quelque part la fonction `factoriel`, ou bien `from math import factorial`), et pour Tchebychev on s'inspire de la méthode pour calculer des suites d'ordre 2. Un conseil est d'être très rigoureux sur les variables en indiquant clairement quel terme de la suite elles représentent en

fonction de l'indice `i` qui apparait dans la boucle, en début comme en fin de boucle :

```
def Tchebychev(n):
    # A représente T_(i) en début de boucle et T_(i+1) à la fin
    # avec T_(0) = 1
    A = [1]
    # B représente T_(i+1) en début de boucle et T_(i+2) à la fin
    # avec T_(1) = x
    B = [0, 1]
    for i in range(n):
        # la formule de récurrence
        C = somme(produit([0, 2], B), produit_constante(A, -1))
        # T_(i) devient T_(i+1)
        A = B
        # T_(i+1) devient T_(i+2)
        B = C
    # à la fin A représente T_(n) et B représente T_(n+1)
    return A
```

□