Problème 1. Manipulations de polynômes

1.1 Représentation dense des polynômes à une variable

La *représentation dense* d'un polynôme à une variable est la liste de ses coefficients : si $P = \sum_{i=0}^{d} c_i X^i$ où les c_i sont les coefficients du polynôme et d son degré (en particulier $c_d \neq 0$), sa représentation est la liste `[c_0, c_1, ..., c_d]`. On remarque que le premier élément de la liste est le coefficient constant de P et le dernier son coefficient dominant. *On impose dans toute la suite que la liste qui représente un polynôme P finisse par un coefficient non nul.*

Par exemple, le polynôme $X^4 - 2X^2 - X + 3$ est représenté par la liste `[3, -1, -2, 0, 1]`. Par convention, le polynôme nul est représenté par la liste vide.

Question 1.1. Manipulations élémentaires

(a) Écrire une fonction `degre` qui prend en entrée un polynôme et renvoie son degré. Le polynôme nul a par convention degré -1.

(b) Écrire une fonction `coefficient` qui prend entrée un polynôme et un entier $i \geq 0$ et renvoie le coefficient de degré i du polynôme. *Si i est plus grand que le degré, la fonction doit renvoyer* 0.

(c) Écrire une fonction `coefficient_constant` qui prend en entrée un polynôme et renvoie son coefficient constant. *La fonction renvoie une erreur pour le polynôme nul.*

(d) La *valuation* d'un polynôme P, notée $\mathrm{val}(P)$, est le degré de son plus petit monôme non nul. Par exemple $\mathrm{val}(X^4 - 2X^2 - X + 3) = 0$ et $\mathrm{val}(X^4 - 3X^3 + X^2) = 2$. On convient que $\mathrm{val}(0) = -1$. Écrire une fonction `valuation` qui prend en entrée un polynôme et renvoie sa valuation.

Question 1.2. Affichage basique On souhaite représenter nos polynômes sous une forme classique. Pour cela, on représente un monôme X^k par la chaîne de caractère `"X^k"`, un terme cX^k par `"c * X^k"` et un polynôme par la somme de ses termes, le terme de plus haut degré étant représenté en premier. Par exemple, le polynôme $X^4 - 2X^2 - X + 3$ est représenté par la chaîne `"1 * X^4 + 0 * X^3 + -2 * X^2 + -1 * X^1 + 3 * X^0"`. *On s'attachera à respecter les espaces prévues.*

(a) Écrire une fonction `repr_monome` qui prend en entrée un entier k et une chaîne `X` qui contient le nom de la variable du polynôme et renvoie la chaîne représentant le monôme de degré k en la variable X. Par exemple, `repr_monome(4, 'Y')` doit renvoyer la chaîne `Y^4`. *On rappelle que `str(k)` renvoie la chaîne de caractère représentant l'entier k en base* 10.

(b) Écrire une fonction `repr_terme` qui prend entrée un coefficient c, un entier k et une chaîne `X` et renvoie la représentation du terme cX^k sous la forme `c * X^k` (en respectant les espaces). *On suppose que `str(c)` renvoie la représentation du coefficient c sous forme de chaîne de caractères.*

(c) Écrire une fonction `repr_poly` qui prend en entrée un polynôme P (sous forme de liste) et une chaîne `X` et renvoie sa représentation sous forme de chaîne de caractères. Le polynôme nul est représenté par la chaîne `"0"`.

Question 1.3. Affichage amélioré On souhaite maintenant améliorer l'affichage des polynômes. On suppose que les polynômes sont à coefficients entiers.

(a) Modifier la fonction `repr_monome` pour qu'elle affiche `"1"` si $k = 0$ (au lieu de `"X^0"`) et `"X"` si $k = 1$ (au lieu de `"X^1"`).

(b) Modifier la fonction `repr_terme` pour qu'elle affiche `"X^k"` lorsque $c = 1$ (au lieu de `"1 * X^k"`) et qu'elle affiche simplement `"c"` lorsque $k = 0$.

(c) Modifier la fonction `repr_poly` pour d'une part qu'elle tienne compte des deux modifications précédentes, qu'elle n'affiche pas les termes nuls et qu'elle n'écrive pas `"+ -c"` mais simplement `"- c"` en présence d'un coefficient c négatif. *Par exemple, le polynôme $X^4 - X^2 - 3X + 2$ est représenté par la chaîne `"X^4 - X^2 - 3 * X + 2"`.*

Question 1.4. Additions et soustractions

(a) Écrire une fonction `addition` qui prend en entrée deux polynômes et renvoie leur somme. *On fera attention à ce que le résultat n'ait pas de* 0 *comme coefficient dominant. Par exemple la somme de* $X^2 + X + 1$ *et* $-X^2 + X + 1$ *est* $2X + 2$*, qui doit être représentée par la liste* `[2, 2]` *et non la liste* `[2, 2, 0]`*.*

(b) Quelle est sa complexité exprimée en nombre d'additions de coefficients, en fonction des degrés des deux polynômes d'entrée ?

(c) De même, écrire une fonction `soustraction` qui renvoie la différence de ses deux arguments.

Question 1.5. Multiplication On rappelle que le produit de deux polynômes $P = \sum_{i=0}^{m} p_i X^i$ et $Q = \sum_{j=0}^{n} q_j X^j$ est le polynôme $R = \sum_{i=0}^{m} \sum_{j=0}^{n} (p_i q_j X^{i+j})$.

(a) Écrire un algorithme `multiplication` qui calcule le produit de deux polynômes, en utilisant la formule précédente, et estimer sa complexité en fonction des degrés m et n des polynômes d'entrée.

On peut améliorer la complexité de l'opération de multiplication en utilisant par exemple l'algorithme de Karatsuba. Pour cela, on suppose que P et Q sont de même degré m et que $m+1$ est une puissance de 2. On note $k = \frac{m+1}{2}$. Soit

$$P = P_0 + X^k P_1 \quad \text{et} \quad Q = Q_0 + X^k Q_1$$

où P_0, P_1, Q_0 et Q_1 sont des polynômes de degré strictement inférieur à k. On définit $R_0 = P_0 \times Q_0$, $R_1 = P_1 \times Q_1$ et $R_2 = (P_0 + P_1) \times (Q_0 + Q_1)$.

(b) Exprimer $P \times Q$ en fonction de P_0, P_1, Q_0, Q_1 et k.

(c) En développant R_2, exprimer $P \times Q$ en fonction de R_0, R_1, R_2 et k, sans utiliser P_0, P_1, Q_0 et Q_1.

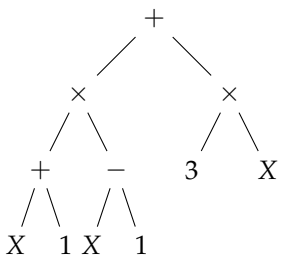
(d) Compléter la fonction `Karatsuba` qui utilise l'expression de la question précédente pour calculer le produit de deux polynômes avec un algorithme récursif.

```
def Karatsuba(P, Q):
    m = degre(P)
    if m == 0: # Cas de base
        return ...

    k = (m+1)//2
    P0, P1 = P[:k], P[k:]
    Q0, Q1 = Q[:k], Q[k:]

    R0 = Karatsuba(P0, Q0)
    R1 = Karatsuba(P1, Q1)
    R2 = Karatsuba(addition(P0,P1), addition(Q0,Q1))

    R = [0] * (2*m + 1) # P*Q est de degré 2m
    # Remplir R à partir de R0, R1 et R2
    ...

    return R
```

(e) On s'intéresse au nombre de multiplications $M(m)$ effectuées par l'algorithme de Karatsuba si les polynômes d'entrée sont de degré m. Donner l'équation de récurrence satisfaite par $M(m)$ et le cas de base $M(0)$. En déduire que $M(m) = O(m^{\log_2(3)})$. Justifier sans calcul que ce coût est sous-quadratique.

1.2 Les expressions polynomiales

Dans cette partie, on représente les polynômes par des arbres binaires : les nœuds (internes) d'un arbre sont étiquetés par l'une des trois opérations $+$, $-$, ou $\times$ et les feuilles par soit un nombre soit une variable. On autorise autant de variables que l'on souhaite. Ces arbres binaires correspondent de manière naturelle à l'écriture d'un polynôme sous forme d'une expression arithmétique. Un exemple est donné en figure 1.




```
plus(
    fois(
        plus(var("X"),cst(1)),
        moins(var("X"), cst(1))
    ),
    fois(cst(3),var("X"))
)
```

Figure 1: Représentation de l'expression $((X+1)\times(X-1))+(3\times X)$ sous forme d'arbre et construction de l'objet `PolExpr` correspondant.

La représentation en Python des expressions polynomiales est la classe `PolExpr` dont une implantation possible est fournie en figure 2.

La construction d'un objet de type `PolExpr` se fait à l'aide des fonctions suivantes :

- `cst(c)` où `c` est un nombre construit l'expression constante égale à c ;
- `var(v)` où `v` est une chaîne de caractère construit l'expression égale à la variable v ;
- `plus(e1,e2)`, `moins(e1,e2)` et `fois(e1,e2)` construisent respectivement la somme, la différence et le produit des deux expressions polynomiales `e1` et `e2`.

Pour manipuler une expression polynomiale, on dispose des méthodes suivantes :

- `e.fils_gauche()` et `e.fils_droit()` renvoient respectivement le fils gauche et le fils droit de la racine de `e` (avec une erreur si `e` est réduite à une feuille) ;
- `e.racine()` renvoie la valeur de la racine de `e`, c'est-à-dire le nombre c si `e` est la constante `cst(c)`, la chaîne `v` si `e` est la variable `var(v)`, et `'+'`, `'-'` ou `'*'` si `e` est une somme, une différence ou un produit, respectivement ;
- `e.est_cst()`, `e.est_var()`, `e.est_plus()`, `e.est_moins()` et `e.est_fois()` renvoient un booléen, qui vaut `True` si l'expression est respectivement une constante, une variable, une somme, une différence et un produit.

Question 1.6. Manipulations élémentaires

(a) Compléter la fonction `taille` suivante qui calcule la taille d'une expression, définie par son nombre d'opérations arithmétiques.

```
def taille(e):
    if e.est_var() or e.est_cst():
        return 0
    if e.est_plus() or e.est_moins() or e.est_fois():
        eg = e.fils_gauche()
        ed = e.fils_droit()
        return ...
    raise ValueError("L'expression est invalide.")
```

```python
class PolExpr:
    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right
    def __repr__(self):
        return "PolExpr({},{},{})".format(self.value, self.left, self.right)
    def fils_gauche(self):
        return self.left
    def fils_droit(self):
        return self.right
    def racine(self):
        return self.value
    def est_cst(self):
        return isinstance(self.value, int) or isinstance(self.value, float)
    def est_var(self):
        return isinstance(self.value, str) and self.value[0].isalpha() and       \
          self.value.isalnum()
    def est_plus(self):
        return self.value == "+"
    def est_moins(self):
        return self.value == "-"
    def est_fois(self):
        return self.value == "*"

def var(s):
    return PolExpr(s, None, None)
def cst(r):
    return PolExpr(r, None, None)
def plus(e, f):
    return PolExpr("+", e, f)
def moins(e, f):
    return PolExpr("-", e, f)
def fois(e, f):
    return PolExpr("*", e, f)
```

Figure 2: Une implantation possible de la classe `PolExpr`

(b) Écrire une fonction qui étant donné une expression e renvoie la liste de ses variables (dans un ordre quelconque). *Attention, la liste ne doit contenir qu'une seule fois chaque variable.*

(c) Écrire une fonction `derive` qui prend en entrée une expression e et une variable v (sous forme de chaîne de caractères) et renvoie l'expression dérivée de e par rapport à la variable v, en utilisant les règles habituelles de dérivation de la somme et du produit. *Par exemple, avec en entrée l'expression $(x+y)\times(2-x)$ et la variable x, `derive` renvoie l'expression $(1+0)\times(2-x)+(x+y)\times(0-1)$. On ne cherchera pas à simplifier le résultat.*

Question 1.7. Simplification d'expressions Comme on l'a vu précédemment, l'application de `derive` sur une expression peut déboucher sur une expression inutilement grosse. Le but de cette partie est d'implanter un algorithme de simplification des expressions polynomiales. L'architecture générale de l'algorithme est donné par la fonction `simplifie` ci-dessous :

```
def simplifie(e):
    if e.est_cst() or e.est_var():
        return e
    g = simplifie(e.fils_gauche()) # on simplifie récursivement
    d = simplifie(e.fils_droit())  #   les fils gauche et droit
    if e.est_plus():               # cas d'une somme
        return simplifie_plus(g, d)
    if e.est_moins():              # cas d'une différence
        return simplifie_moins(g, d)
    if e.est_fois():               # cas d'un produit
        return simplifie_fois(g, d)
    raise ValueError("L'expression n'est pas valide.")
```

Les fonctions `simplifie_plus`, `simplifie_moins` et `simplifie_fois` doivent implanter les simplifications suivantes :

- la somme, la différence et le produit entre deux nombres doivent être remplacées par la constante résultat (l'expression $2+3\times4$ devient donc la constante 14) ;
- les multiplications par 1 et 0, l'addition avec 0 ou la soustraction de 0 doivent être simplifiées (l'expression $1\times(x-0)$ devient x) ;
- l'expression $0-e$ doit être remplacée par $(-1)\times e$.

(a) Compléter la fonction `simplifie_plus` pour implanter les simplifications demandées.

```
def simplifie_plus(g, d):
    if g.est_cst() and d.est_cst():
        return ...
    if g.est_cst() and g.racine() == 0:
        return ...
    if d.est_cst() and d.racine() == 0:
        return ...
    return plus(g, d)
```

(b) Écrire la fonction `simplifie_moins` qui effectue les simplifications demandées.

(c) Écrire la fonction `simplifie_fois` qui effectue les simplifications demandées.

Question 1.8. Écriture d'une expression

(a) Écrire une fonction `developpe` qui prend en entrée une expression *e* (de type `PolExpr`) ne faisant intervenir qu'une unique variable et renvoie le polynôme correspondant sous forme dense (c'est-à-dire la liste de ses coefficients). *On pourra utiliser les fonctions `addition` et `multiplication` de la première partie.*

(b) Écrire une fonction `notation_infixe` qui étant donné une expression *e* sous forme de `PolExpr` renvoie *e* en notation infixe, c'est-à-dire la notation habituelle en mathématique, en entourant chaque opération de parenthèses. *Par exemple, si e est obtenue par `moins(fois(var("x"), cst(2)), plus(cst(3), var("y")))`, la fonction renvoie la chaîne `"((x * 2) - (3 + y))"`.*

(c) Écrire une fonction `notation_infixe_associative` qui effectue le même travail que la fonction précédente, mais en tenant compte des règles d'associativité habituelle. *Par exemple, avec la même expression qu'à la question précédente, la fonction renvoie la chaîne `"x * 2 - (3 + y)"`.*

Question 1.9. Représentation graphique des expressions L'objectif de cette question est de pouvoir représenter graphiquement une expression sous forme d'arbre. Pour cela, on utilise uniquement des caractères ASCII[1]. Par exemple, l'arbre de la figure 1 est représenté par le dessin de la figure 3.

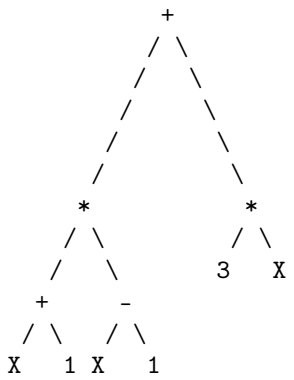


Figure 3: Représentation en *ASCII-art* de l'expression $((X+1)\times(X-1))+(3\times X)$.

L'algorithme de représentation consiste à calculer la représentation des fils gauche et droit de l'expression, à les juxtaposer horizontalement, puis d'ajouter la racine et les arêtes qui la relient aux racines de ses fils gauche et droit. On suppose pour simplifier que toutes les constantes et les variables peuvent être représentées par un seul caractère.

Formellement, la représentation d'une expression *e* est une liste de chaînes de caractères, toutes de même longueur. Chaque élément de la liste correspond à une ligne de la représentation. Par exemple, la représentation de l'expression $X+1$ est la liste `['␣␣+␣␣', '␣/␣\\␣', 'X␣␣␣1']`[2].

Pour juxtaposer deux représentations, il faut qu'elles soient de même taille, c'est-à-dire avoir le même nombre de lignes, et la largeur des lignes être la même des deux côtés. Pour cela, si l'une est plus petite que l'autre, on l'englobe dans un rectangle de la taille de la plus grande, en la centrant horizontalement et en l'alignant vers le haut, de telle sorte que la racine soit toujours au centre de la première ligne.

(a) Écrire une fonction `affiche_expr` qui prend en entrée la représentation d'une expression sous la forme d'une liste de chaînes de caractères et affiche l'expression sous forme d'arbre à l'écran. *La fonction ne doit rien renvoyer.*

(b) Écrire une fonction `centre_ligne` qui prend en entrée une chaîne de caractère `s` et une largeur ℓ, et renvoie une chaîne de largeur ℓ contenant `s` au milieu et des caractères *espace* (`" "`) autour. *On suppose que ℓ est un entier de même parité que la longueur de `s`, supérieur à celle-ci.*

[1] Ce type de représentation graphique est appelée *art ASCII*.

[2] Remarque : on a représenté les caractères *espace* de manière visible par `'␣'` ici, et le caractère *contre-oblique* \ est doublé car c'est un caractère d'échappement en Python.

(c) Écrire une fonction `englobe` qui prend en entrée une représentation d'expression (sous la forme d'une liste) et deux entiers ℓ et h et renvoie la représentation de la même expression, englobée dans un rectangle de largeur ℓ et de hauteur h, centrée horizontalement et alignée en haut. *On suppose que les dimensions ℓ et h sont suffisantes pour contenir la représentation de l'expression.*

(d) Écrire une fonction `juxtapose` qui prend en entrée les représentations de deux expressions et un entier i et renvoie leur juxtaposition horizontale, avec i colonnes remplies du caractère *espace* entre les deux représentations.

(e) Compléter la fonction `ascii_art` qui prend en entrée une expression e (de type `PolExpr`) et renvoie la représentation de e comme liste de chaînes de caractères. *Indication : le cas où les fils gauche et droit de e sont tous les deux soit une variable soit une constante est un cas particulier à traiter à part.*

```
def ascii_art(e):
    if e.est_cst(): return [str(e.racine())]
    if e.est_var(): return [e.racine()]

    Lg = ascii_art(e.fils_gauche())
    Ld = ascii_art(e.fils_droit())

    larg = max(len(Lg[0]), len(Ld[0]))
    haut = max(len(Lg), len(Ld))

    if larg == 1: # Cas particulier
        ...
        ...
        ...

    else:
        L = [" " * ... + e.racine() + " " * ...] # Racine
        for i in range(...):              # Dessin des arêtes
            ligne = " " * ... + "/" + " " * ... + "\\" + " " * ...
            L.append(ligne)
        L.extend(juxtapose(Lg, Ld, ...))

    return L
```