Devoir à la maison n°1

Distribué le jeudi 17 octobre 2024, ***à rendre le jeudi 7 novembre 2024***

Exercice 1 : Ecrire une fonction récursive, prenant en argument deux entiers naturels a et b (b étant supposé non nul), qui renvoie le reste dans la division euclidienne de a par b. Les seules opérations arithmétiques autorisées dans votre programme sont l'addition et la soustraction.

Exercice 2 : Une façon de représenter en mémoire un polynôme $P(x) = \sum_{i=0}^{n} a_i x^i$ consiste à stocker ses coefficients a_i dans une liste que l'on nommera `liste_coeff`.

Si P est le polynôme nul on conviendra que cette liste est vide, et sinon cette liste aura une longueur égale au degré du polynôme P augmenté de 1, et `liste_coeff[i]` contiendra la valeur de a_i.

Ainsi par exemple le polynôme $P(x) = 1 + 4x + 2x^3$ admet pour liste associée $[1, 4, 0, 2]$. Dans toute la suite on supposera que l'on travaille avec des polynômes différents du polynôme nul.

1. Algorithme naïf
   (a) Définir une fonction Python `Naif`, prenant comme paramètre d'entrée une liste associée à un polynôme P, et un flottant x, retournant la valeur de $P(x)$. Cette fonction utilisera des additions, des multiplications, et la fonction exponentiation de Python, codée par a**k pour le calcul de a^k.
   (b) En Python, le coût (=complexité temporelle) d'une exponentiation à la puissance k (k entier naturel) est $\alpha \log(k)$ (où α est une constante), sauf pour $k = 0$ où le coût vaut 1 .
   Montrer que le coût $C_1(n)$ de l'algorithme `Naif` défini à la question précédente, appliqué à un polynôme de degré n, est $C_1(n) = \alpha \log(n!) + \beta n + \gamma$, où β et γ sont des constantes.

2. Algorithme de Horner
   L'évaluation d'un polynôme en un réel x consiste à écrire le polynôme $P(x)$ de la façon suivante :

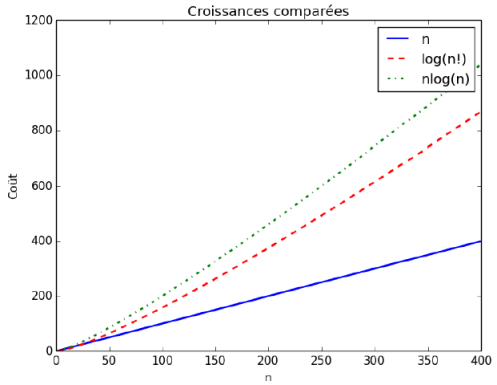
$$P(x) = \sum_{i=0}^{n} a_i x^i = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots)))$$

   Exemple : Pour évaluer $P(x) = 1 + 4x + 2x^3$, on écrit P(x) = 1 + x(4 + x(0 + x(2))) Le calcul se déroule donc suivant les étapes suivantes :

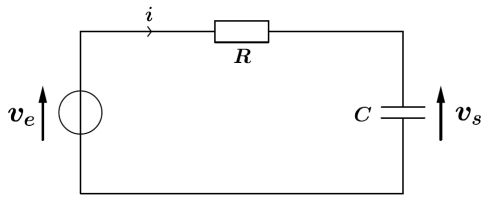
$$2 \rightarrow 0 + x \times 2 \rightarrow 4 + x \times (0 + x \times 2) \rightarrow 1 + x \times (4 + x \times (0 + x \times 2))$$

   (a) Définir une fonction Python `Horner`, prenant comme paramètre d'entrée une liste associée à un polynôme P, et un flottant x, retournant la valeur de $P(x)$. Cette fonction utilisera la méthode de Horner décrite ci-dessus.
   (b) Déterminer la complexité $C_2(n)$ de cet algorithme pour un polynôme de degré n . Quel nom porte ce type de complexité ?

(c) On donne sur la figure ci-dessous le graphe de trois fonctions.
Entre l'algorithme naïf et l'algorithme de Horner, quel est l'algorithme le plus efficace ? Au niveau des calculs, d'où provient le gain en efficacité ?



Exercice 3 : On souhaite réaliser un filtrage numérique avec un filtre passe-bas du premier ordre. Pour cela, on considère un circuit RC série pour lequel on note v_e la tension d'entrée et v_s la tension aux bornes du condensateur.



On peut établir l'équation différentielle

$$v_e = v_s + \frac{1}{\omega_c}\frac{dv_s}{dt} \quad \text{avec } \omega_c = \frac{1}{RC}.$$

Le signal d'entrée v_e est échantillonné avec une période d'échantillonnage T_e. Pour tout entier k allant de 0 à N, on note $t_k = kT_e$.

L'échantillonnage du signal d'entrée est stocké dans une liste `Ve` composée de flottants, i.e. :

$$\texttt{Ve=[}v_e(t_0), v_e(t_1), ..., v_e(t_N)\texttt{]}$$

1. Montrer que pour tout entier $k \in [\![0; N-1]\!]$, on a
$$v_s(t_{k+1}) = v_s(t_k) + \omega_c \int_{t_k}^{t_{k+1}} v_e(t) - v_s(t)dt.$$
2. Pour tout entier $k \in [\![0; N-1]\!]$, on approche l'intégrale $\int_{t_k}^{t_{k+1}} v_e(t) - v_s(t)dt$ par l'aire du trapèze formé par les points de coordonnées $(t_k, 0)$, $(t_{k+1}, 0)$, $(t_{k+1}, v_e(t_{k+1}) - v_s(t_{k+1}))$ et $(t_k, v_e(t_k) - v_s(t_k))$. Avec cette approximation, montrer qu'il existe deux réels A et B tels que, pour tout $k \in [\![0; N-1]\!]$, on ait
$$v_s(t_{k+1}) = Av_s(t_k) + B(v_e(t_{k+1}) + v_e(t_k)).$$
Exprimer A et B en fonction de T_e et ω_c.
3. On part de $v_s(0) = 1$. Écrire une fonction `signsortie` qui reçoit en argument le tableau `Ve` et retourne un tableau de même longueur contenant les valeurs du signal de sortie calculées aux instants t_k en utilisant la relation de récurrence précédente.