TP4 : Interpolation de Lagrange

Soit I un intervalle. Soit $f : I \to \mathbb{R}$.
On souhaite approcher la fonction f par des fonctions *simples*, faciles à évaluer en un point par exemple. Le théorème de Weierstrass (hors programme) affirme que toute fonction f continue sur un segment peut être aprochée uniformément sur ce segment par une suite de polynômes. De bonnes fonctions candidates seraient donc les fonctions polynomiales.

Objectif : Approcher la fonction f par une fonction polynomiale.

.1. Interpolation de Lagrange

Soit $n \in \mathbb{N}^*$. Soit $(x_0, x_1, \dots, x_n) \in I^{n+1}$ deux à deux distincts. On note :

$$y_0 = f(x_0), \qquad y_1 = f(x_1), \qquad \text{et} \qquad y_n = f(x_n)$$

- On choisit d'**interpoler** la fonction f. Autrement dit, on choisit d'approcher f par une fonction qui dont le graphe passe par les points $\big(x_0, y_0\big)$, ..., $\big(x_n, y_n\big)$.
- L'interpolation de Lagrange consiste à chercher une **fonction polynomiale** P_n qui approche f et l'interpole aux points (x_0, y_0), ..., (x_n, y_n).
  On souhaite donc construire un polynôme P_n de degré inférieur ou égal à n vérifiant :

$$\forall i \in [\![0, n]\!], \quad P_n(x_i) \ = \ y_i$$

- On peut démontrer, à l'aide du cours d'algèbre linéaire, le résultat suivant.

  Théorème 1.
  Il existe un unique polynôme P_n de degré inférieur ou égal à n, vérifiant :

$$\forall i \in [\![0, n]\!], \quad P_n(x_i) \ = \ y_i$$

  Il s'agit du polynôme suivant :

$$P_n(X) \ = \ \sum_{i=0}^{n} y_i \, L_i(X)$$

  où : $\forall i \in [\![0, n]\!]$, $L_i(X) = \prod\limits_{\substack{j=0 \\ j \neq i}}^{n} \dfrac{X - x_j}{x_i - x_j}$.
  Les polynômes L_0, ..., L_n sont appelés **polynômes de Lagrange** *associés aux* **noeuds** (x_j).

  Remarque
  La famille $(L_0, L_1, \dots, L_n)$ forme une base de $\mathbb{R}_n[X]$ appelée **base de Lagrange**.

I. Opérations sur les polynômes

Le but de cette partie est de définir un certain nombre d'outils spécifiques aux polynômes à coefficients réels, avant de s'intéresser plus spécifiquement à l'interpolation de Lagrange.

- Les polynômes seront représentés en **Python** par des listes. Plus précisément, le polynôme $P(X) = \sum\limits_{k=0}^{n} a_k X^k$ sera représenté par la liste $[a_n, a_{n-1}, \ldots, a_1, a_0]$ de ses coefficients, rangés par ordre des degrés décroissants.
- Par la suite, on identifiera un polynômé P et sa représentation par une liste `P`.

I.1. Normalisation

- Si $P \neq 0_{\mathbb{R}_n[X]}$, on dira que sa représentation sous forme de liste est *normalisée* lorsque `len(P)` est égale à $\deg(P) + 1$.
- La représentation normalisée du polynôme nul sera la liste vide.

▶ Définir une fonction **`normalise`** qui prend en argument une représentation quelconque d'un polynôme P et la normalise.

> On recherche le premier terme non nul de la liste `P`, et on supprime tous les 0 inutiles qui le précèdent. Ainsi, le premier coefficient de la liste renvoyée correspond au coefficient dominant du polynôme P.
>
> ```python
> def normalise(P) :
>     ''normalise(P : list) -> list''
>     if P == [] :
>         return []
>     else :
>         i = 0
>         while P[i] == 0 :
>             i += 1
>         return P[i:]
> ```

Désormais, on supposera que les représentations des polynômes qui seront passés en argument des fonctions à venir seront normalisées, et on exigera de ces dernières qu'elles retournent des représentations normalisées.

I.2. Évaluation par algorithme de Horner

Soit $x \in \mathbb{R}$. L'algorithme de Horner, consiste à effectuer le calcul de $P(x)$ en exploitant l'égalité suivante :

$$P(x) \;=\; \Big(\big((a_n\,x + a_{n-1})\,x + a_{n-2}\big)\,x + \cdots + a_1\Big)\,x + a_0$$

► Écrire, en donnant sa signature, une fonction `horner` qui prend en paramètre la liste `P` des coefficients d'un polynôme P et un réel `x`, et qui renvoie l'évaluation de P en `x` à l'aide de l'algorithme de Horner.

```
1  def horner(P, x) :
2      ''horner(P : list, x : float) -> y = float''
3      n = len(P) - 1
4      y = P[0]
5      for k in range(1, n+1) :
6          y = y * x + P[k]
7      return y
```

► Démontrer la terminaison de cet algorithme.

La variable `n-k` définit une suite strictement décroissante d'entiers positifs. En effet, la variable `n-k` prend successivement les valeurs $\mathtt{n} - 1$ jusqu'à 0 en décroissant de 1 à chaque tour de boucle (par construction de la boucle `for`). C'est donc un variant de la boucle `for`. La boucle se termine donc.

► Complexité.

- Déterminer le nombre total de multiplications et d'additions effectuées en fonction de n.

Soit `P` une liste de taille $n + 1$.
On considère l'addition et la multiplication comme opérations élémentaires.

- On commence par effectuer 1 opération élémentaire en ligne 3.
- On effectue ensuite n tours de boucle.
  De plus, pour chaque tour de boucle, on effectue 2 opérations élémentaires (1 addition et 1 multiplication).

Finalement, on effectue $1 + 2n$ opérations élémentaires.

- En déduire la complexité de cet algorithme.

On en déduit que l'algorithme de Horner est en $\underset{n \to +\infty}{\Theta}(n)$.

► Proposer une version récursive de l'algorithme de Horner. On nommera cette fonction `hornerrec`.

```
def hornerrec(P, x) :
    ''hornerrec(P : list, x : float) -> float''
    if len(P) == 1 :
        return P[0]
    else :
        L = P[: len(P) - 1]
        return hornerrec(L, x) * x + P[len(P) - 1]
```

I.3. Somme de polynômes

► Définir une fonction nommée `somme` prenant en arguments 2 listes `P` et `P` représentant 2 polynômes P et Q, et retournant la représentation normalisée de $P+Q$.

```
def somme(P, Q) :
    ''somme(P : list, Q : list) -> S : list''
    p = len(P)
    q = len(Q)
    if p < q :
        P = [0] * (q-p) + P
    else :
        Q = [0] * (q-p) + Q
    S = [P[k] + Q[k] for k in range(max(p, q))]
    return normalise(S)
```

I.4. Produit externe par un réel

► Définir une fonction nommée `mult` qui prend en argument un scalaire `a` et une liste `P` représentant un polynôme P, et qui renvoie la représentation normalisée du polynôme $\mathtt{a} \cdot P$.

```
def mult(a, P) :
    ''mult(a : float, P : list) -> list''
    if a == 0 :
        return []
    return [a * coef for coef in P]
```

I.5. Produit interne de polynômes

► Définir une fonction nommée `produit` qui retourne la représentation normalisée du produit de deux polynômes P et Q.

Soit $(P,Q) \in \left(\mathbb{R}_n[X]\right)^2$. Alors il existe $(a_0,\ldots,a_n,b_0,\ldots,b_n) \in \mathbb{R}^{n+2}$ tels que :

$$P(X) \ = \ \sum_{k=0}^{p} a_k\, X^k \qquad \text{et} \qquad Q(X) \ = \ \sum_{k=0}^{q} b_k\, X^k$$

On rappelle : $(P \times Q)(X) = \sum\limits_{k=0}^{p+q} \left(\sum\limits_{i=0}^{k} a_i\, b_{k-i} \right) X^k$.

```
def produit(P, Q) :
    p = len(P) - 1
    q = len(Q) - 1
    n = p + q
    R = [0] * (n + 1)
    P = [0] * (n - p) + P
    Q = [0] * (n - q) + Q
    for k in range(n + 1) :
        L = [ P[n-i] * Q[n - (k-i)] for i in range(k+1) ]
        R[n-k] = sum(L)
    return normalise(R)
```

II. Interpolation polynomiale

II.1. Polynômes de Lagrange

On rappelle qu'on considère x_0, x_1, ..., x_n des réels de l'intervalle I, deux à deux distincts. On rappelle également l'expression des polynômes de Lagrange associés aux noeuds (x_j) :

$$\forall i \in [\![0, n]\!], \qquad L_i(X) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{X - x_j}{x_i - x_j}$$

- Définir une fonction **lagrange** qui prend en argument la liste des noeuds (x_j) et un entier i, et qui retourne la représentation normalisée du polynôme L_i.

```python
def lagrange(noeuds, i) :
    ''lagrange(noeuds : list, i : int) -> L : list''
    L = [1]
    X = noeuds[:] # permet de copier la liste
    xi = X.pop(i)
    for xj in X :
        L = mult( 1/(xi-xj), produit(L, [1, -xj]) )
    return L
```

- Soit $(i, j) \in [\![0, n]\!]^2$.
  À quoi est égal $L_i(x_j)$? En déduire, pour tout $(y_0, y_1, \dots, y_n) \in \mathbb{R}^{n+1}$, l'existence d'un polynôme P_n de degré inférieur ou égal à n vérifiant :

$$\forall j \in [\![0, n]\!], \quad P_n(x_j) = y_j$$

Il est assez simple de prouver que ce polynôme P_n est l'unique polynôme de degré inférieur ou égal à n vérifiant ces conditions d'interpolations. Il sera désormais appelé le **polynômes d'interpolation de Lagrange** associé aux points (x_0, y_0), ..., (x_n, y_n).

- Soit $(i, j) \in [\![0, n]\!]^2$.

$$L_i(x_j) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

  Ainsi, le polynôme $\boxed{P_n(X) = \sum_{i=0}^{n} y_i\, L_i(X)}$ vérifie :

$$\deg(P_n) \leqslant n \qquad \text{et} \qquad \forall j \in [\![0, n]\!],\ P_n(x_j) = y_j$$

- Démontrons l'unicité de P_n en raisonnant par l'absurde.
  Supposons qu'il existe un polynôme Q_n distinct de P_n vérifiant :

$$\deg(Q_n) \leqslant n \qquad \text{et} \qquad \forall j \in [\![0, n]\!],\ Q_n(x_j) = y_j$$

  Alors, on démontre rapidement que :
  × le polynôme $P_n - Q_n$ est de degré au plus n,
  × le polynôme $P_n - Q_n$ admet au moins $n + 1$ racines (les réels x_0, x_1, ..., x_n).
  Le polynômes $P_n - Q_n$ est donc le polynôme nul. Ainsi : $P_n = Q_n$. Absurde !

- Définir alors une fonction nommée `interpole` qui prend en arguments les deux listes (x_j) et (y_j) et qui retourne ce polynôme P_n.

```
def interpole(X, Y) :
    ''interpole(X : list, Y : list) -> L : list''
    P = []
    n = len(X) - 1
    for i in range(n+1) :
        P = somme(P, mult(Y[i], lagrange(X, i)))
    return P
```

II.2. Interpolation polynomiale d'une fonction

Soit $(a, b) \in \mathbb{R}^2$ vérifiant : $a < b$. Soit $f : [a, b] \to \mathbb{R}$. Soit $(x_0, x_1, \ldots, x_n) \in [a, b]^{n+1}$ deux à deux distincts.

D'après la section précédente, un polynôme interpolateur de f est le polynôme d'interpolation de Lagrange associé aux points $\big(x_0, f(x_0)\big)$, $\big(x_1, f(x_1)\big)$, …, $\big(x_n, f(x_n)\big)$. Celui-ci sera désormais noté $P_n(f)$.

On s'intéresse à la question de la convergence uniforme du polynôme d'interpolation de Lagrange de f lorsque le nombre de noeuds n tend vers $+\infty$. Autrement dit, nous allons chercher à déterminer dans quelle mesure la quantité suivante tend vers 0 quand n tend vers $+\infty$.

$$M_n \;=\; \|f - P_n(f)\|_{\infty,[a,b]} \;=\; \sup\Big\{\big|f(t) - P_n(f)(t)\big| \;\Big|\; t \in [a, b]\Big\}$$

- Définir une fonction `norme` prenant en arguments une fonction `g` et deux réels `a` et `b`, et retournant la quantité :

$$\max\left\{\left|g\left(a + k\,\frac{b-a}{1000}\right)\right| \;\middle|\; 0 \leqslant k \leqslant 1000\right\}$$

Par la suite, cette fonction servira à évaluer la quantité M_n.

Il s'agit ici d'une recherche classique de maximum.

```
def norme(g, a, b) :
    ''norme(g : function, a : float, b : float) -> M : float''
    T = [a + k * (b-a) / 1000 for k in range(1001)]
    M = abs( g(T[0]) )
    for k in range(1, 1001) :
        x = abs( g(T[k]) )
        if x > M :
            M = x
    return M
```

II.2.a) Répartition uniforme des noeuds

Par construction du polynôme $P_n(f)$, ce polynôme interpolateur dépend de x_0, x_1, …, x_n, et donc de leur répartition sur le segment $[a, b]$.
On choisira dans la suite une répartition uniforme des noeuds en posant :

$$\forall j \in [\![0, n]\!], \qquad x_j \ = \ a + j\,\frac{b-a}{n}$$

On considère tout d'abord la fonction $f : t \mapsto \sin(\pi\, t)$ sur l'intervalle $[-1, 1]$.

- Écrire un script permettant de coder la fonction f.

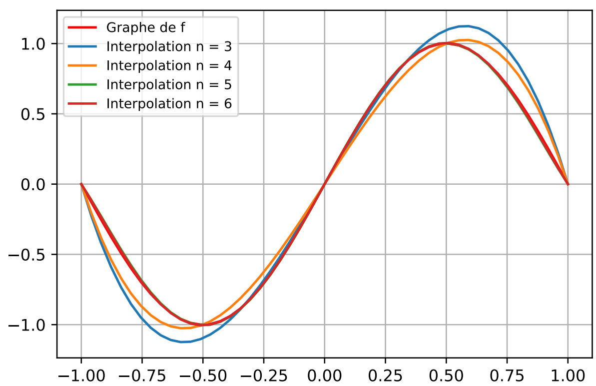
```python
import numpy as np
def f(t) :
    ''f(t : float) -> float''
    return np.sin(np.pi * t)
```

- Rédiger un script affichant, dans une même fenêtre graphique, le graphe de la fonction f et de ses polynômes interpolateurs de Lagrange $P_n(f)$ pour $n \in \{3, 4, 5, 6\}$.

```python
import matplotlib.pyplot as plt
a = -1
b = 1
X = np.linspace(a, b)
Y = [f(x) for x in X]
plt.clf()
plt.plot(X, Y color = 'red', label = 'Graphe de f')

for n in [3, 4, 5, 6] :
    X0 = list( np.linspace(a, b, n + 1) )
    Y0 = [f(x) for x in X0]
    P = interpole(X0, Y0)
    Z = [horner(P, x) for x in X]
    plt.plot(X, Z, label = 'Interpolation n = ' + str(n))

plt.grid()
plt.legend()
```



- Rédiger ensuite un script permettant de déterminer la plus petite valeur de n pour laquelle : $M_n \leqslant 10^{-6}$.

On calcule les polynômes d'interpolation $P_n(f)$ pour les valeurs successives de n tant que : $M_n > 10^{-6}$. Pour l'évaluation de M_n, on utilise la fonction `norme` définie plus haut.

```python
def pol_int(x) :
    return f(x) - horner(P, x)

n = 1
N = 2
while N > 10**(-6) :
    n += 1
    X0 = list( np.linspace(a, b, n + 1) )
    Y0 = [f(x) for x in X0]
    P = interpole(X0, Y0)
    N = norme(pol_int, a, b)
print(n)
```

II.2.b) Phénomène de Runge

On considère désormais la fonction $g : t \mapsto \dfrac{1}{1 + 8\,t^2}$, toujours sur l'intervalle $[-1, 1]$.

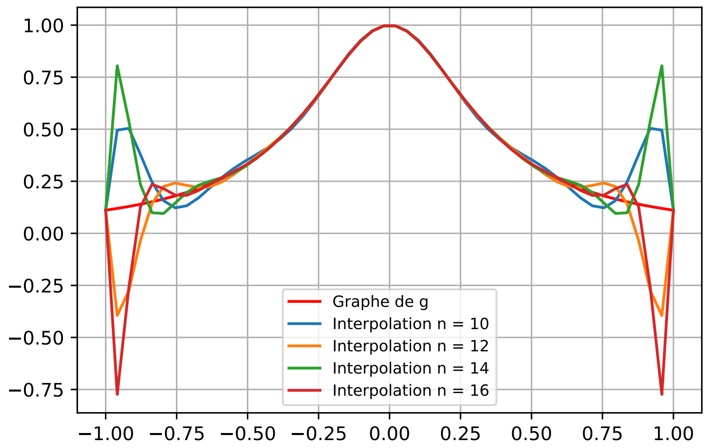
- Rédiger un script affichant, dans une même fenêtre graphique, le graphe de la fonction g et de ses polynômes interpolateurs de Lagrange $P_n(g)$ pour $n \in \{10, 12, 14, 16\}$.

```python
def g(t) :
    return 1 / (1 + 8 * t**2)

X = np.linspace(a, b)
Y = [g(x) for x in X]
plt.clf()
plt.plot(X, Y color = 'red', label = 'Graphe de g')

for n in [10, 12, 14, 16] :
    X0 = list( np.linspace(a, b, n + 1) )
    Y0 = [g(x) for x in X0]
    P = interpole(X0, Y0)
    Z = [horner(P, x) for x in X]
    plt.plot(X, Z, label = 'Interpolation n = ' + str(n))

plt.grid()
plt.legend()
```



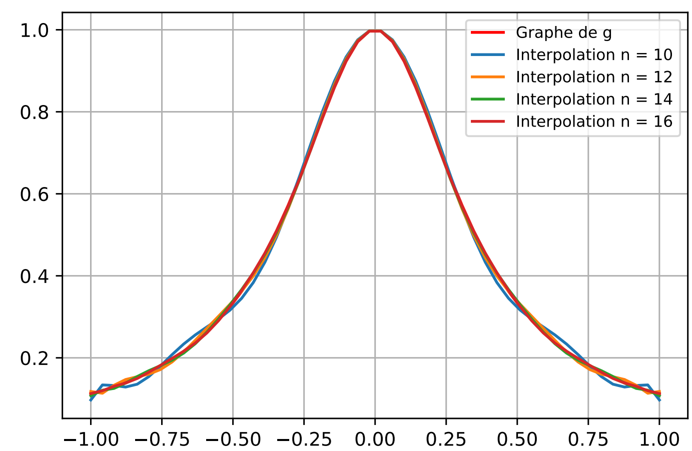
La divergence de l'interpolation que l'on observe au voisinage des extrémités de l'intervalle $[a, b]$ porte le nom de **phénomène de Runge**. Cet effet peut être évité en choisissant pour noeuds les **points de Tchebychev** :

$$\forall j \in [\![0, n]\!], \qquad x_j \;=\; \cos\left(\frac{(2j+1)\,\pi}{2\,(n+1)}\right)$$

► Reprendre la question précédente en choisissant pour noeuds les points de Tchebychev.

On modifie seulement la ligne 10 de la question précédente de la façon suivante.

```
X0 = [np.cos( (2*j + 1) * np.pi / (2 * (n + 1)) ) for j in range(n+1)]
```



► Avec ces points, pour quelle valeur minimale de n obtient-on : $M_n \leqslant 10^{-3}$?

On modifie légèrement le programme effectué en section précédente.

```
def pol_int2(x) :
    return g(x) - horner(P, x)

n = 1
N = 2
while N > 10**(-3) :
    n += 1
    X0 = [np.cos( (2*j + 1) * np.pi / (2 * (n + 1)) ) for j in range(n+1)]
    Y0 = [g(x) for x in X0]
    P = interpole(X0, Y0)
    N = norme(pol_int2, a, b)
print(n)
```