PSI*

Informatique

Devoir (auto)surveillé N°1

Le barème est fourni à titre indicatif.

Dans ce sujet, on suppose que toutes les fonctions Python demandées sont écrites dans un script comportant la ligne :

```
import numpy as np
```

Si vous êtes amenés à utiliser tout ou parties de certains modules Python, penser à ajouter les instructions d'importation appropriées.

Par ailleurs, vos fonctions ne chercheront pas à valider la correction des arguments passés.

Si vous êtes amenés à introduire des variables, utilisez des noms explicites et/ou ajouter des commentaires permettant de les identifier facilement. De façon générale, pensez à limiter le nombre de variables utilisées dans vos codes.

Exercice I : suites récurrentes linéaires d'entiers (15 points)

On s'intéresse à la suite $(u_n) \in \mathbb{Z}^{\mathbb{N}}$ définie par ses p ($p \in \mathbb{N}^*$) premiers termes $u_0, u_1, ..., u_{p-1}$ avec $\forall i \in [\![0\,;\,p-1]\!], u_i \in \mathbb{Z}$ et la récurrence linéaire d'ordre p suivante :

$$\forall n \geq p-1, u_{n+1} = \alpha_0 u_n + \alpha_1 u_{n-1} + ... + \alpha_{p-1} u_{n-p+1}$$

où les p coefficients α_i sont entiers.

Dans ces conditions, on montre facilement que l'on a : $\forall n \in \mathbb{N}, u_n \in \mathbb{Z}$.

Partie I : écriture matricielle et construction d'un tableau `numpy`

Pour tout entier naturel $n \geq p-1$, on peut écrire :

$$\begin{pmatrix} u_{n+1} \\ u_n \\ u_{n-1} \\ \vdots \\ \vdots \\ u_{n-p+3} \\ u_{n-p+2} \end{pmatrix} = \underbrace{\begin{pmatrix} \alpha_0 & \alpha_1 & \alpha_1 & \cdots & \cdots & \cdots & \alpha_{p-1} \\ 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & 0 & & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & \ddots & \ddots & 0 & 0 & 0 \\ \vdots & & & \ddots & 1 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 \end{pmatrix}}_{\substack{=A \\ (\text{matrice carrée d'ordre } p)}} \begin{pmatrix} u_n \\ u_{n-1} \\ u_{n-2} \\ \vdots \\ \vdots \\ u_{n-p+2} \\ u_{n-p+1} \end{pmatrix}$$

Dans un premier temps, le codage des entiers de numpy sur 64 bits suffit à nos calculs.

[Q.1 / 4 points] Ecrire une fonction Python `A_build` qui reçoit en argument une liste `ALPHA` (dont les éléments sont, dans cet ordre, les coefficients $\alpha_0, \alpha_1, ..., \alpha_{p-1}$) et qui renvoie un tableau `numpy` correspondant à la matrice A.

Pour cette première question, on rappelle que la fonction `zeros` de `numpy` permet de créer un tableau rempli de zéros. Elle reçoit en argument un tuple comportant autant de d'éléments que le tableau comporte de dimensions. Ainsi, la matrice nulle de dimension 5×8 sera obtenue grâce à l'appel :

```
np.zeros((5,8))
```

Partie II : utilisation d'une classe d'objets et calcul de u_n

A partir de la récurrence précédente, on a, pour tout entier naturel $n \geq p-2$:

$$\begin{pmatrix} u_{n+1} \\ u_n \\ \vdots \\ u_{n-p+2} \end{pmatrix} = A^{n-p+2} \begin{pmatrix} u_{p-1} \\ u_{p-2} \\ \vdots \\ u_0 \end{pmatrix}$$

On s'est ainsi classiquement ramené au calcul d'une puissance d'une matrice carrée.

[Q.2 / 3 points] Rappeler rapidement (mais précisément !) le principe général de l'exponentiation rapide pour l'exponentiation d'un scalaire ou d'une matrice.

On suppose, à partir de maintenant, que le codage des entiers de `numpy` sur 64 bits ne suffit pas pour nos calculs. Pour pouvoir « aller plus loin » et tirer parti des capacités de Python à manipuler de grands entiers, on suppose également que l'on a créé un nouveau type, appelé `MatInt`, d'objets correspondant à des matrices d'entiers.

Ces objets ont trois attributs :

- `coeff` : correspondant à une liste de listes, chaque liste de la liste principale correspondant à une ligne de la matrice considérée.
- `nrows` : correspondant à la longueur de `coeff` (et donc aux nombres de lignes de la matrice).
- `ncol` : correspondant à la longueur de chaque élément de la liste `coeff` (et donc au nombre de colonnes de la matrice).

Ainsi, pour créer l'objet `MatInt` correspondant à la matrice

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

on utilisera la syntaxe :

```
M = MatInt([[1,2],[3,4],[5,6]])
```

et `M.coeff` vaudra `[[1,2],[3,4],[5,6]]`, `M.nrows` vaudra 3 et `M.ncol` vaudra 2.

[Q.3 / 3 points] Ecrire une fonction Python `A_build2` qui reçoit en argument une liste `ALPHA` (dont les éléments sont, dans cet ordre, les coefficients $\alpha_0, \alpha_1, ..., \alpha_{p-1}$) et qui renvoie un objet `MatInt` correspondant à la matrice A.

On suppose que dans la classe `MatInt`, on a surchargé les opérateurs de multiplication (`*`) et d'exponentiation (`**`).

[Q.4 / 5 points] Ecrire une fonction Python `SRL` qui reçoit en argument une liste `ALPHA` (dont les éléments sont, dans cet ordre, les coefficients $\alpha_0, \alpha_1, ..., \alpha_{p-1}$), une liste `INIT` (dont les éléments sont, dans cet ordre, les *p* premiers termes de la suite (u_n) : $u_0, u_1, ..., u_{p-1}$) et un entier naturel *n*. La fonction `SRL` devra renvoyer l'entier u_n.

Exercice II : factorielles et autres bidules binomiaux (25 points)

Pour tout entier naturel n et tout entier naturel $p \leq n$, on rappelle que l'on a :

$$\binom{n}{p} = \frac{n!}{p! \times (n-p)!}$$

Ce nombre est un entier naturel.

Partie I : un calcul efficace

L'évaluation de $\binom{n}{p}$ à l'aide de l'expression précédente conduit à l'évaluation de trois factorielles qui peuvent atteindre des valeurs élevées (alors que $\binom{n}{p}$ est petit) et conduire à des erreurs d'arrondi sur la division. Ainsi, on préfère utiliser, pour p non nul :

$$\binom{n}{p} = \frac{n \times (n-1) \times (n-2) \times ... \times (n-p+1)}{p \times (p-1) \times (p-2) \times ... \times 2 \times 1}$$

[Q.1 / 2 points] Evaluer, en fonction de p, le nombre total de multiplications requises par ce calcul (on ne comptabilisera pas la division).

Rappelons également que l'on a :

$$\binom{n}{p} = \binom{n}{n-p} = \binom{n}{p'}$$

[Q.2 / 3 points] Ecrire une fonction Python `binomial_base` recevant deux entiers n et p (avec $p \leq n$) et renvoyant l'entier $\binom{n}{p}$ en effectuant un minimum de multiplications.

Partie II : deux codes récursifs

Pour tous entiers naturels *n* et *p* non nuls, on a la formule de Pascal :

$$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$$

[Q.3 / 3 points] Ecrire une fonction Python récursive `binomial_rec1` recevant deux entiers *n* et *p* (avec $p \leq n$) et renvoyant l'entier $\binom{n}{p}$ en utilisant la formule de Pascal ci-dessus (les cas de base sont $p = n$ et $p = 0$).

[Q.4 / 2 points] Le code précédent n'est pas du tout efficace. En considérant le calcul de $\binom{7}{3}$, sauriez-vous expliquer pourquoi ?

On peut également considérer la relation suivante, valable pour tous entiers naturels *n* et *p* non nuls :

$$\binom{n}{p} = \frac{n}{p} \times \binom{n-1}{p-1}$$

[Q.5 / 4 points] Ecrire une fonction Python récursive `binomial_rec2` recevant deux entiers *n* et *p* (avec $p \leq n$) et renvoyant l'entier $\binom{n}{p}$ en utilisant la formule précédente (Attention ! Assurez-vous que la fonction renvoie bien un entier !).

Partie III : Des calculs avec les flottants

Pour tout réel *x* strictement positif, rappelons que l'on a (fonction Gamma) :

$$\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} \mathrm{d}t$$

Pour tout entier naturel *n*, on a : $n! = \Gamma(n+1)$.

Pour de grandes valeurs de *n*, on utilise classiquement une valeur approchée de $\Gamma(n+1)$ pour obtenir une valeur approchée de $n!$ En 1964, le mathématicien hongrois Cornelius LANCZOS a proposé l'approximation suivante :

$$\Gamma(x+1) \simeq (x+5,5)^{x+0,5} \times e^{-(x+5,5)} \times \sqrt{2\pi} \times \left(c_0 + \frac{c_1}{x+1} + \frac{c_2}{x+2} + ... + \frac{c_6}{x+6} \right)$$

où les coefficients c_i sont des réels avec c_0 très proche de 1.

[Q.6 / 4 points] Ecrire une fonction Python `Gamma_LN` recevant un flottant strictement positif *x* et renvoyant le logarithme népérien de $\Gamma(x)$ (on utilisera $\Gamma(x) = \frac{\Gamma(x+1)}{x}$ et on supposera que la fonction `Gamma_LN` comporte la ligne `L=[…]` où `L` est une liste contenant, dans cet ordre, les coefficients $c_0, c_1, ..., c_6$.).

Il n'est pas utile d'utiliser la fonction Gamma pour des valeurs de *n* « raisonnables » (disons jusqu'à 30). On propose la fonction Python suivante qui renvoie un flottant comme résultat de $n!$ La fonction comporte des valeurs de $n!$ pour $n \in [\![0\,;4]\!]$, effectue un calcul classique pour $n \in [\![5\,;30]\!]$ et utilise la fonction `Gamma_LN` pour $n \geq 31$:

```
def factorial_wg(n):
    from math import exp
    F = [1.,1.,2.,6.,24.]
    if n <= 4:
        return(…)        A COMPLETER
    elif n <= 30:
        res = 24.
        for k in range(5,n+1):
            res *= k
        return(…)        A COMPLETER
    else:
        return(…)        A COMPLETER
```

[Q.7 / 2 points] Sur votre copie, recopier et compléter les trois « return » de la fonction `factorial_wg`.

En écrivant $\ln\left(\binom{n}{p}\right) = \ln\left(\frac{n!}{p!\times(n-p)!}\right) = \ln(n!) - \ln(p!) - \ln((n-p)!)$, on peut utiliser le calcul suivant pour coder le calcul de $\binom{n}{p}$:

$$\mathrm{E}\left(0.5 + \exp\left(\ln(n!) - \ln(p!) - \ln((n-p)!)\right)\right) \qquad (E)$$

[Q.8 / 3 points] En vous inspirant de la fonction `factorial_wg`, coder la fonction `factorial_LN_wg` qui reçoit en argument un entier *n* et renvoie un flottant correspondant à $\ln(n!)$.

[Q.9 / 2 points] En utilisant l'expression (E) ci-dessus, écrire une fonction `binomial_wg`, qui reçoit deux entiers *n* et *p* ($p \leq n$) et renvoie un flottant correspondant à $\binom{n}{p}$.

Fin du sujet