Sujet 0 CCP PSI Corrigé – Durée 3h

Erreurs dans le sujet initial, corrigées sur votre version :

- m_local.append(float(input("Masse de l'élement "+**str**(i)+' : '))) (**string** à remplacer par **str** plusieurs fois)
- L'amortissement est proportionnel à la vitesse → c_i est le coefficient de u_i' et pas de u_i dans les équations (1), (2), (3), confirmé par l'équation (4).
- Dans l'équation (2), c'est + $c_2 u_2'$ et pas – c_2
- L'aide en annexe définit les matrices en utilisant array. Pourquoi pas matrix qui respecte les multiplications matricielles ? ou donner la fonction dot qui effectue des multiplications matricielles sur les arrays à deux dimensions.
- *npts* au lieu de *ntps* dans la Q9, et on donne *ntps* = 10 000, valeur inutile et modifiée dans la Q12
- Dans la définition de la puissance dissipée pour la dernière question, c'est c_i au lieu de c.
- La fonction calcul_energie(X,c) doit aussi prendre dt en argument.

Q1

Sous la ligne ##FONCTIONS A APPELER ICI, on entre : L,n,m,k,c = geometrie(0)

Q2

L = 0.5, n = 3, m = [0.1, 0.2, 0.1], k = [20000.0, 20000.0,1000.0], c = [1000.0, 10.0, 10.0]
(Python rajoute .0 pour indiquer qu'il s'agit de flottants)

Q3

On conseille d'ouvrir les fichiers avec with, mais nous devons utiliser les fonctions données dans la documentation en annexe. Dans le même esprit, il est conseillé d'utiliser un bloc try ... except pour manipuler les fichiers, mais puisque cela n'est pas mentionné ici oublions-le (autant laisser le système afficher un message d'erreur plutôt que notre programme se poursuive avec des erreurs mal gérées).

```
def lire_fichier(nom_fic):
    f = open(nom_fic,'r')
    L_local = float(f.readline())
    n_local = int(f.readline())
    m_local,k_local,c_local = [],[],[]
    for i in range(n_local):
        m_local.append(float(f.readline()))
    for i in range(n_local):
        k_local.append(float(f.readline()))
    for i in range(n_local):
        c_local.append(float(f.readline()))
    f.close()
    return [L_local, n_local, m_local, k_local, c_local]
```

Q4

```
requete1 = "SELECT id, nom, date FROM CALCUL"
```

Q5

```
    requete2 = "SELECT L,n FROM CALCUL WHERE id = "+str(id_calcul)
    resultat=interroge_bdd(requete2)
    [L_local,n_local]=resultat[0]
    requete3 = "SELECT masse, raideur, amortissement FROM POUTRE JOIN LIAISON
```

```
        ON POUTRE.num_elem = LIAISON.num_liaison WHERE id_calcul = "+str(id_calcul)
    resultat=interroge_bdd(requete3)
    m_local = [resultat[i][0] for i in range(n_local)]
    k_local = [resultat[i][1] for i in range(n_local)]
    c_local = [resultat[i][2] for i in range(n_local)]
  return [L_local, n_local, m_local, k_local, c_local]
```

Remarque : m_local,k_local,c_local=resultat[:,0],resultat[:,1],resultat[:,2] peut aussi fonctionner, mais renvoie des tableaux colonnes et pas des listes.

Q6

Le système s'écrit :

$m_1u_1'' + (c_1 + c_2)u_1' - c_2u_2' + (k_1 + k_2)u_1 - k_2u_2 = 0$

$m_iu_i'' - c_iu_{i-1}' + (c_i + c_{i+1})u_i' - c_{i+1}u_{i+1}' - k_iu_{i-1} + (k_i + k_{i+1})u_i - k_{i+1}u_{i+1} = 0$

pour i allant de 2 à $n - 1$

$m_nu_n'' - c_nu_{n-1}' + c_nu_n' - k_nu_{n-1} + k_nu_n = f_n$

D'où les matrices :

$$M = \begin{pmatrix} m_1 & 0 & \dots & \dots & 0 \\ 0 & m_2 & 0 & & \vdots \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & m_n \end{pmatrix} \quad C = \begin{pmatrix} c_1 + c_2 & -c_2 & 0 & \dots & \dots & 0 \\ -c_2 & c_2 + c_3 & -c_3 & \ddots & & \vdots \\ 0 & -c_3 & c_3 + c_4 & -c_4 & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & -c_{n-1} & c_{n-1} + c_n & -c_n \\ 0 & \dots & \dots & 0 & -c_n & c_n \end{pmatrix}$$

$$K = \begin{pmatrix} k_1 + k_2 & -k_2 & 0 & \dots & \dots & 0 \\ -k_2 & k_2 + k_3 & -k_3 & \ddots & & \vdots \\ 0 & -k_3 & k_3 + k_4 & -k_4 & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & -k_{n-1} & k_{n-1} + k_n & -k_n \\ 0 & \dots & \dots & 0 & -k_n & k_n \end{pmatrix}$$

Q7

Rien ne dit ici que l'on doit utiliser les fonctions données en annexe. Nous allons donc nous autoriser la fonction diag, mais on aurait pu travailler avec la fonction zeros donnée en annexe.

La fonction *diag* est importée de numpy.

```
def creation_operateur(n,m,k,c):
    M_local=diag(m)        # crée une matrice diagonale avec les éléments de m sur la diagonale
    d_temp = [k[i]+k[i+1] for i in range(n-1)]+[k[n-1]] # diagonale de la matrice K
    K_local=diag(d_temp)
    for i in range(1,n):
        K_local[i,i-1] = K_local[i-1,i]=-k[i]
    d_temp = [c[i]+c[i+1] for i in range(n-1)]+[c[n-1]] # diagonale de la matrice C
    C_local=diag(d_temp)
    for i in range(1,n):
        C_local[i,i-1] = C_local[i-1,i]=-c[i]
    return [M_local,K_local,C_local]
```

Q8

Le schéma d'Euler explicite consiste à approcher y'_k par $\frac{y_{k+1}-y_k}{h}$, ce qui donne ensuite

$y_{k+1} = y_k + hy'_k = y_k + hf(x_k, y_k)$ avec $y' = f(x, y)$ et $h = x_{k+1} - x_k$

Le schéma d'Euler implicite consiste à approcher y'_{k+1} par $\frac{y_{k+1}-y_k}{h}$, ce qui donne ensuite

$y_{k+1} = y_k + hy'_{k+1} = y_k + hf(x_{k+1}, y_{k+1})$.

Pour $q \geq 2$, on a :

$V_q = \frac{X_q - X_{q-1}}{dt}$ et $A_q = \frac{V_q - V_{q-1}}{dt} = \frac{\frac{X_q - X_{q-1}}{dt} - \frac{X_{q-1} - X_{q-2}}{dt}}{dt}$ donc $A_q = \frac{X_q - 2X_{q-1} + X_{q-2}}{(dt)^2}$

L'équation (4) devient à l'instant t_q : $MA_q + CV_q + KX_q = F_q$ avec $F = {}^t\left(0 \quad 0 \quad \dots \quad 0 \quad f_{max}\sin(\omega t_q)\right)$

$$\frac{1}{(dt)^2}M\left(X_q - 2X_{q-1} + X_{q-2}\right) + \frac{1}{dt}C\left(X_q - X_{q-1}\right) + KX_q = F_q$$

$$M\left(X_q - 2X_{q-1} + X_{q-2}\right) + dt\ C\left(X_q - X_{q-1}\right) + (dt)^2 KX_q = (dt)^2 F_q$$

$$(M + dt\ C + (dt)^2 K)X_q = M\left(2X_{q-1} - X_{q-2}\right) + dt\ CX_{q-1} + (dt)^2 F_q$$

Soit $HX_q = G_q$

avec $H = M + dt\ C + (dt)^2 K$

et $G_q = M\left(2X_{q-1} - X_{q-2}\right) + dt\ CX_{q-1} + (dt)^2 F_q$

Q9

Petit problème : X est une matrice npts×n donc les X_q sont stockés en ligne a priori, ce qui semble logique, puisque chaque X_q nouvellement calculé est rajouté à la liste X comme un élément supplémentaire, mais en pratique, les X_q sont des vecteurs colonnes. On doit, ou bien décider que la fonction resoud renvoie un vecteur colonne X_q, et le convertir en ligne pour le stocker dans X, ou bien un vecteur ligne (ou une liste) et le convertir en vecteur colonne pour effectuer le calcul de Gq.

Les valeurs initiales ne sont pas données… au départ, les éléments sont à leur position d'équilibre, $X_0 = 0_n$ et à l'arrêt, $V_0 = 0_n$. Ainsi, $X_1 = dtV_1$ et $V_1 = dt\ A_1$.

$MA_1 + CV_1 + KX_1 = F_1$ donne alors $HX_1 = (dt)^2 F_1$.

On peut intégrer le calcul de X_1 dans la boucle principale en prenant comme valeurs initiales $X_0 = 0_n$ et $X_{-1} = 0_n$ (ce qu'on aurait pu deviner sans calcul !).

```
def calcul(n,M,K,C,npts,dt,fmax,omega):
    X = zeros((npts,n),float)              # construction de X avec X[0] = [0,...,0]
    Xqm1,Xqm2 = zeros((n,1),float),zeros((n,1),float)   # valeurs initiales de X_(-1) et X_(0)
    H = M+dt*C+dt*dt*K
    tq=0
    for q in range(1,npts):                # on commence à X[1]
        tq += dt
        Fq = zeros((n,1),float)            # j'aurais initialisé Fq avant la boucle, je suis la consigne
        Fq[n-1] = fmax*sin(omega*tq)
        Gq = dot(M,2*Xqm1-Xqm2)+dt*dot(C,Xqm1)+dt*dt*Fq
        X[q] = resoud(H,Gq)
        Xqm2,Xqm1 = Xqm1,X[q]
        Xqm1 = array(Xqm1)                 # convertit Xqm1 en array
        Xqm1.shape = (n,1)                 # puis en vecteur colonne, nécessaire pour le calcul de Gq
    return X
```

La fonction dot du module numpy effectue la multiplication matricielle entre deux arrays.

Attention à bien utiliser des array (créés avec zeros) pour les vecteurs colonnes et pas des listes.

*On aurait pu utiliser des matrix aussi, auquel cas on remplace dot(A,B) par A*B.*

Q10

Méthode : on met la matrice *H* sous cette forme $\begin{pmatrix} 1 & H'_{12} & 0 & \dots & 0 \\ 0 & 1 & H'_{23} & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & H'_{n-1,n} \\ 0 & \dots & \dots & 0 & 1 \end{pmatrix}$ en utilisant le pivot de Gauss.
On résout ensuite le système de proche en proche en commençant par le bas.

<u>Première version</u> : elle fonctionne mais je divise les lignes entières (complexité en O(n) au lieu de diviser seulement les coefficients non nuls… ou mieux, seulement les coefficients qui vont être utiles ensuite : il suffit de savoir qu'en théorie certains coefficient valent 0 et 1, inutile de les modifier en pratique).

```
def resoud(H,Gq):
    n = len(Gq)
    # créé une autre matrice pour ne pas modifier H
    N = zeros((n,n+1),float)
    N[:,:-1] = H
    N[:,-1] = transpose(Gq) # Gq est un vecteur colonne et N[:,-1] un vecteur ligne
    # On élimine tous les éléments sous la diagonale et on met les coefficients diagonaux à 1
    for i in range(n-1):
        N[i] = N[i]/N[i,i]
        N[i+1] = N[i+1] - N[i+1,i]*N[i]
    N[n-1]=N[n-1]/N[n-1,n-1]
    # On résout le système obtenu de proche en proche en commençant par le bas
    X=[0]*n
    X[n-1] = N[n-1,n]
    for i in range(n-2,-1,-1):
        X[i]=N[i,n]-N[i,i+1]*X[i+1]
    return X
```

<u>Deuxième version</u> : agit uniquement sur les coefficients nécessaires

```
def resoud(H,Gq):
    n = len(Gq)
    # créé une autre matrice pour ne pas modifier H
    N = zeros((n,n),float)
    N[:,:] = H
    # en théorie, on élimine tous les éléments sous la diagonale et on met les coefficients
    # diagonaux à 1 mais en pratique on modifie seulement les valeurs qui nous intéressent
    for i in range(n-1):
        pivot = N[i,i]
        N[i,i+1],Gq[i] = N[i,i+1]/pivot,Gq[i]/pivot
        N[i+1,i+1],Gq[i+1] = N[i+1,i+1] - N[i+1,i]*N[i,i+1],Gq[i+1]-N[i+1,i]*Gq[i]
    Gq[n-1] /= N[n-1,n-1]
    # On résout le système obtenu de proche en proche en commençant par le bas
    X=[0]*n
    X[n-1] = Gq[n-1]
    for i in range(n-2,-1,-1):
```

```
        X[i]=Gq[i]-N[i,i+1]*X[i+1]
    return X
```

<u>Troisième version</u> : on peut faire mieux en récupérant dans la matrice H seulement les coefficients utiles au lieu de recopier toute la matrice, mais ce travail supplémentaire risque d'être fastidieux et de ne pas rapporter davantage de points au concours !

Remarque : Cet algorithme est l'algorithme de Thomas.

Q11

La complexité de la fonction resoud est en O(n) (environ 3n multiplications par exemple), ce qui est bien plus performant qu'un pivot de Gauss traditionnel en O(n^3) (3 boucles imbriquées).

Il s'agit de la deuxième version. La première version est en O(n^2) puisqu'une opération sur une ligne revient à n opérations.

Q12

$T = 0{,}3$ et $dt = 2 \times 10^{-6}$ donc $npts = T/dt = 150\,000$
La matrice X est de dimension $npts \times n$, donc $150\,000 \times 200$, et contient donc 30 000 000 de flottants.
En double précision, un flottant est stocké sur 64 bits, soit 8 octets.
Il faut donc 240 000 000 octets, soit 0,2235 Go (1 Go = 1024^3 octets)

Remarque : d'après la commande sys.getsizeof, il semblerait que contrairement à la théorie, les flottants prennent 16 octets, auquel cas il faudra 0,447 Go.

Q13

On peut penser dans un premier temps à représenter les déplacements en y, mais vu qu'on représente la déformation de la poutre, les déplacements sont horizontaux.

```
def affiche_deplacement_pdt(q,X,L,n,ampl):
    X1 = linspace(0,L,n)          # position initiale
    X1 += X[q]*ampl               #X[q] est un array, donc il s'agit d'une multiplication vectorielle
    pl.axes()
    pl.xlim([-L/5,L*1.2])       # si on veut la même échelle que sur la figure 3
    pl.ylim([-1,1])             # si on veut la même échelle que sur la figure 3
    pl.plot(X1,[0]*n,color='blue',marker='D')
    pl.show()
```

Q14

```
def lieu_temps_depl_max(X):
    max = q0 = j0 = 0
    for q in range(len(X)):
        for j in range(len(X[0])):
            if abs(X[q,j]) > max:
                q0,j0,max = q,j,abs(X[q,j])
    return(j0,q0,max)     # dans l'ordre : numero du noeud, pas de temps, abs(deplacement max)
```

Q15

X contient les valeurs $u_i(t)$ à chaque instant $t = q.dt$, pour $q \in [[0, npts - 1[[$ et $i \in [[1,n]]$.

On effectue la même approximation que dans la question 8 : $V_q \approx \frac{X_q - X_{q-1}}{dt}$

A l'instant $t = q.dt$, $\dot{u}_i(t) \approx \frac{u_i(q.dt) - u_i((q-1)dt)}{dt} = \frac{X[q,i-1] - X[q-1,i-1]}{dt}$

pour i allant de 1 à n et q allant de 1 à npts – 1

$\dot{u}_i(t) - \dot{u}_{i-1}(t) \approx \frac{X[q,i-1] - X[q-1,i-1] - X[q,i-2] + X[q-1,i-2]}{dt}$

$P_{diss}(q.dt) \approx \frac{1}{(dt)^2} \sum_{i=2}^{n} c[i-1] * (X[q,i-1] - X[q-1,i-1] - X[q,i-2] + X[q-1,i-2])^2$

pour q allant de 1 à npts – 1

Avec $P_{diss}(0) = 0$, on connait les valeurs de P_{diss} pour q allant de 0 à npts – 1. Nous allons utiliser la méthode des rectangles à gauche pour calculer l'intégrale cherchée : $E_{diss}(T) = dt \sum_{q=0}^{ntps-1} P_{diss}(q.dt)$

$E_{diss}(T) = \sum_{q=0}^{ntps-1} \frac{1}{dt} \sum_{i=2}^{n} c[i-1] * (X[q,i-1] - X[q-1,i-1] - X[q,i-2] + X[q-1,i-2])^2$

$E_{diss}(T) = \sum_{q=0}^{ntps-1} \frac{1}{dt} \sum_{i=1}^{n-1} c[i] * (X[q,i] - X[q-1,i] - X[q,i-1] + X[q-1,i-1])^2$

```
def calcul_energie(X,c,dt):
    energie = 0
    t = 0
    X1 = arange(0,len(X)*dt,dt)     # vecteur des temps pour le tracé
    Y1 = [0]                        # vecteur des énergies avec E(0)
    for q in range(1,len(X)):
        pdiss = 0                   # la variable pdiss est égale à Pdiss(t)*dt
        for i in range(1,n):
            pdiss += c[i]*(X[q,i]-X[q-1,i]-X[q,i-1]+X[q-1,i-1])**2
        pdiss /= dt
        energie += pdiss
        Y1.append(energie)
    pl.plot(X1,Y1)
    pl.show()
    return energie
```

Remarque : la fonction a besoin de l'argument *dt* (pas mentionné dans l'énoncé).