2
Sujet IPT — mars 2020

I

Mouvement d'une plateforme en mer — CCINP 2019 PC

I.1 Codage

Q 1. *Si N correspond au nombre de points, il y a $(N-1)$ intervalles, chacun d'eux représentant la durée Δt. Il faut donc $t_{\max} = \Delta t \cdot (N-1)$ et comme on exige que N soit entier, il faut convertir le quotient (flottant) en entier.*

```
tmax = 100.0 # secondes
dt   = 0.01  # secondes
N    = 1 + int(tmax/dt)  # entier (nb de points)
```

Remarque.— *Le rapport du jury mentionne que* l'ajout du 1 est très souvent oublié. *Il pourrait aussi mentionner que cette démarche est absurde : on ne peut pas imposer les valeurs de* $\mathrm{d}t$ *et de N simultanément, il y aura presque toujours des erreurs d'arrondi.*

`tmax`	1874	1875	1876
`dt`	1.874	1.875	1.876
`N`	1000	1001	1001

En pratique, il faut imposer N et choisir $\mathrm{d}t$ *en conséquence (ou imposer* $\mathrm{d}t$ *et calculer N en conséquence, mais personnellement, je privilégie la première manière).*

Q 2. *L'énoncé demande que l'instant $t_{\max}$ fasse partie de l'échantillon des instants. D'après l'annexe :*

```
T = np.linspace(0, tmax, N)
```

Q 3. *Réponse évidente.*

```
def force(f0, t, w):
    return f0*np.cos(w*t)
```

Remarque.— *Comme on utilise la fonction* cos *du module* `numpy`, *l'argument* `t` *peut être aussi bien un flottant qu'un tableau* `numpy` *de flottants (comme* `T`*).*

Q 4. *On définit une réponse nulle par défaut. Si le booléen (global)* `exc` *prend la valeur* `True`, *on modifie le tableau* `F` *à l'aide de la fonction* `force` *qui vient d'être définie et des* **variables globales** `F0`, `T` *et* `w`.

```
def force_exc():
    F = np.zeros(N)
    if exc:   # True en cas d'excitation
        F = force(F0, T, w)
    return F
```

Q 5. *D'après l'équation différentielle* (11),

$$x''(t_n) + 2\zeta\omega_0 x'(t_n) + \omega_0^2 x(t_n) = \frac{F_0}{m}\cos\omega_0 t_n$$

c'est-à-dire

$$v'(t_n) = \frac{F_0}{m}\cos\omega_0 t_n - 2\zeta\omega_0 \cdot v(t_n) - \omega_0^2 \cdot x(t_n) \approx \frac{v(t_{n+1}) - v(t_n)}{\mathrm{d}t}$$

Le schéma d'Euler est donc défini par la relation de récurrence suivante.

$$\forall\, 0 \leqslant n < N-1, \quad \begin{cases} x_{n+1} = & x_n + & \Delta t \cdot v_n \\ v_{n+1} = & -\omega_0^2 \Delta t \cdot x_n + & \left[1 - 2\zeta\omega_0 \Delta t\right] \cdot v_n + \frac{F_0}{m} \Delta t \cos \omega_0 t_n \end{cases}$$

Remarque.— *Prendre soin de répondre exactement à la question posée : on ne demande pas a_n, on demande la relation de récurrence entre (x_n, v_n) et (x_{n+1}, v_{n+1}).*

Remarque.— *Attention au quantificateur ! Les dernières valeurs calculées sont x_{N-1} et v_{N-1}, donc le dernier indice pour lequel on applique la relation de récurrence est $(N-2)$!*

Remarque.— *Le signe $\approx$ dans l'énoncé est absolument ridicule. Il faut choisir :*
– *ou bien on étudie un échantillon*

$$\big(x(t_0), x(t_1), \ldots, x(t_{N-1})\big)$$

de la fonction $x(t)$ (et dans ce cas, on a effectivement des valeurs approchées, qui découlent du Théorème des accroissements finis) ;
– *ou bien on définit un schéma numérique (Euler ou un schéma plus élaboré) et dans ce cas, on définit une suite au moyen d'une relation de récurrence et donc avec des égalités !*

Q 6. *Une bonne pratique (déjà préconisée en 1635) consiste à isoler les différents calculs pour que la structure de l'algorithme reste claire.*

Q 6.a *L'énergie mécanique totale $E(t)$ est la somme de l'énergie cinétique et de l'énergie potentielle.*

$$E_c(t) = \frac{1}{2} m v^2(t) \qquad E_p(t) = \frac{1}{2} k x^2(t)$$

La masse `m` *du système et la constante de raideur* `k` *sont des* **variables globales**.

```
def energie(x, v):
    return 0.5*(m*v**2 + k*x**2)
```

Q 6.b *On code également à part la relation de récurrence établie à la question précédente.*

```
def schema_euler(x, v, f):
    xx = (x + dt*v)
    vv = (v + dt*(f/m-2*zeta*om0*v-om0**2*x))
    return xx, vv
```

Dans ces fonctions, `dt`, `m`, `zeta` *et* `om0` *sont des* **variables globales**. *Le paramètre* `f` *donne la valeur de l'excitation F.*

Remarque.— *On n'ironisera pas sur le fait de noter* `w` *la pulsation ω et de noter* `om0` *la pulsation propre ω_0. (Il est bien connu que l'incohérence des notations augmente sensiblement le risque d'erreur de code.)*

Q 6.c *On peut maintenant écrire un schéma d'Euler clair.*
La longueur de l'excitation `F` *donne le nombre de points de discrétisation.*

```
def euler(F, x0, v0):
    N = len(F)
    # initialisation
    X, V, E = np.zeros(N), np.zeros(N), np.zeros(N)
    X[0], V[0], E[0] = x0, v0, energie(x0, v0)
    # itération
    for i in range(N-1):
        X[i+1], V[i+1] = schema_euler(X[i], V[i], F[i])
        E[i+1] = energie(X[i+1], V[i+1])
    return X, V, E
```

Q 7. *Un peu de mathématiques ! L'énoncé nous ressert ces $\approx$ qui sont absurdes...*

Q 7.a *D'après le Théorème des accroissements finis, le taux d'accroissement*

$$\frac{x\big((n+1)\Delta t\big) - x\big(n\Delta t\big)}{(n+1)\Delta t - n\Delta t}$$

est égal à la valeur de la dérivée x' à un certain instant compris entre $n\Delta t$ et $(n+1)\Delta t$. Il est donc raisonnable de considérer que

$$(\star) \qquad \frac{x\big((n+1)\Delta t\big) - x\big(n\Delta t\big)}{\Delta t} \approx x'\big((n+1/2)\Delta t\big)$$

et donc de **définir** *le schéma numérique en* posant *:*

$$(\ddagger) \qquad \forall\, 0 \leqslant n < N-1, \quad \frac{x_{n+1} - x_n}{\Delta t} = v_{n+1/2}$$

c'est-à-dire

$$\forall\, 0 \leqslant n < N-1, \quad x_{n+1} = x_n + \Delta t \cdot v_{n+1/2}.$$

Q 7.b *D'après la Formule de Taylor-Young,*

$$\frac{v\big((n+1/2)\Delta t\big) + v\big((n-1/2)\Delta t\big)}{2} = v(n\Delta t) + \frac{v'(n\Delta t)}{2} \cdot (\Delta t - \Delta t) + \mathcal{O}[(\Delta t)^2] = v(n\Delta t) + \mathcal{O}[(\Delta t)^2],$$

$$\frac{v\big((n+1/2)\Delta t\big) - v\big((n-1/2)\Delta t\big)}{\Delta t} = \frac{v'(n\Delta t)}{2\Delta t} \cdot (\Delta t + \Delta t) + \mathcal{O}[(\Delta t)^2] = v'(n\Delta t) + \mathcal{O}[(\Delta t)^2].$$

On déduit alors de (11) que

$$\frac{v\big((n+1/2)\Delta t\big) - v\big((n-1/2)\Delta t\big)}{\Delta t} \approx -\omega_0^2 \cdot x(n\Delta t) - 2\zeta\omega_0 \cdot \frac{v\big((n+1/2)\Delta t\big) + v\big((n-1/2)\Delta t\big)}{2} + \frac{F(n\Delta t)}{m}.$$

Il est donc raisonnable de fonder un schéma numérique sur l'égalité

$$\frac{v_{n+1/2} - v_{n-1/2}}{\Delta t} = -\omega_0^2 \cdot x_n - 2\zeta\omega_0 \cdot \frac{v_{n+1/2} + v_{n-1/2}}{2} + \frac{F_n}{m}$$

qu'on a tout intérêt à réécrire comme une relation de récurrence :

$$v_{n+1/2} = \frac{-\omega_0^2 \Delta t}{1 + \zeta\omega_0\Delta t} \cdot x_n + \frac{1 - \zeta\omega_0\Delta t}{1 + \zeta\omega_0\Delta t} \cdot v_{n-1/2} + \frac{F_n \cdot \Delta t}{m(1 + \zeta\omega_0\Delta t)}.$$

Remarque.— *Je ne mets pas de quantificateur ! L'énoncé nous dit d'une part que x_{n+1} est obtenu à partir de x_n et $v_{n+1/2}$ et que la dernière valeur calculée correspond à $n+1 = N-1$, c'est-à-dire $n + 1/2 = N - 1/2$, et d'autre part que la dernière valeur calculée de v_n est $v_{N-3/2}$. Il faudrait savoir !*

Q 7.c *Indiquons sans attendre la différence majeure entre ce schéma et le schéma d'Euler calculé plus haut : x_{n+1} est une fonction de x_n et de $v_{n+1/2}$ tandis que $v_{n+1/2}$ est une fonction de x_n et de $v_{n-1/2}$. Il nous faudra donc calculer $v_{n+1/2}$ avant de calculer x_{n+1} (alors que dans le schéma d'Euler, on calculait simultanément x_{n+1} et v_{n+1} en fonction de x_n et de v_n).*

Q 7.d *Enfin, pour faciliter la suite des opérations, rappelons la correspondance entre les valeurs réelles, les valeurs calculées par le schéma numérique et leur représentation informatique pour $0 \leqslant i < N$.*

	Valeur réelle	*Valeur calculée*	*Notation Python*
Position	$x(i\Delta t)$	x_i	`X[i]`
Vitesse	$v\big((i-1/2)\Delta t\big)$	$v_{i-1/2}$	`V[i]`
Énergie	$E(i\Delta t)$		`E[i]`

Q 8.a *Pour calculer une valeur approchée de*

$$E(i\Delta t) = \frac{1}{2}m\big(v(i\Delta t)\big)^2 + \frac{1}{2}k\big(x(i\Delta t)\big)^2,$$

il faut connaître des valeurs approchées de $x(i\Delta t)$ *et de* $v(i\Delta t)$*. On vient de le voir :*

$$v(i\Delta t) \approx \frac{v\big((i-1/2)\Delta t\big) + v\big((i+1/2)\Delta t\big)}{2}.$$

Il est donc raisonnable d'approcher $E(i\Delta t)$ *par le code suivant (la fonction* `energie(x, v)` *a été définie au* **[Q 6.a.]***).*

```
E[i] = energie(X[i], 0.5*(V[i]+V[i+1]))
```

Q 8.b *L'évaluation de* `E[i+1]` *pourrait poser un problème si je savais ce que valait l'indice i...*
On devine quelle difficulté pose le code précédent : pour calculer `E[i]`*, il faut connaître* `V[i+1]`*. Cela nous mène tout droit vers une* `IndexError` *lors du calcul de la dernière valeur de* `E[i]`*.*
En conséquence, le tableau `E` *sera plus petit que les deux tableaux* `X` *et* `V`*.*

Q 9. *Nous allons reprendre le code du schéma d'Euler* **[Q 6.]** *en ne changeant que ce qui mérite d'être changé.*
Il faut bien entendu remplacer le schéma d'Euler par le schéma Leapfrog. (Pas Frogleap, hein !)
À part ça, seul le calcul des valeurs de l'énergie est modifié : la liste `E` *est raccourcie et l'estimation de la vitesse* v_i *a changé (cf. question précédente).*

```
tmp = zeta*om0*dt
fac1, fac2 = 1-tmp, 1/(1+tmp)

def schema_leapfrog(x, v, f):
    vv = -om0**2*dt*fac2*x + fac1*fac2*v + f*dt*fac2/m
    xx = x + vv*dt
    return xx, vv

def leapfrog(F, x0, v0):
    N = len(F)
    # initialisation
    X, V, E = np.zeros(N), np.zeros(N), np.zeros(N-1)
    X[0], V[0] = x0, v0
    # itération
    for i in range(N-1):
        X[i+1], V[i+1] = schema_leapfrog(X[i], V[i], F[i])
        E[i] = energie(X[i], 0.5*(V[i]+V[i+1]))
    return X, V, E
```

Remarque.— *Il me paraît plus commode de définir* `fac1` *et* `fac2` *comme des variables globales plutôt que comme des variables* locales *(elles s'expriment en fonction de variables globales et on a déjà défini un assez grand nombre de variables globales pour ne pas se priver d'en définir d'autres).*

Remarque.— *Un code bien structuré est plus facile à modifier (à corriger, à mettre à jour, à développer...). Vu ?*

Q 10.a *On commence par définir les variables globales (qui sont des constantes)* en vérifiant que les unités sont cohérentes.

```
# Variables globales
m    = 110000      # masse en kg
k    =  67900      # Cte de raideur en N/m
om0  =      0.786  # pulsation en rad/s
zeta =      0.0501 # sans dimension

F0   =  10000      # amplitude de l'excitation en N
w    =      0.5    # pulsation de l'excitation en rad/s
```

Q 10.b *Il faut ensuite calculer un échantillon de l'excitation et définir les paramètres numériques de la simulation (cf* **[Q 1.]***). Par* **[Q 4.]***, il faut définir le tableau* $T = (i\Delta t)_{0\leqslant i<N}$ *des instants pour appeler la fonction* `force_exc()`.

```
# Discrétisation du temps
tmax = 10.0    # secondes
dt   =    0.01 # secondes
N    = 1+int(tmax/dt)
# Excitation
exc = True
T = np.linspace(0, tmax, N)
F = Force_exc()
```

Q 10.c *Enfin, si on a l'intention de faire varier les conditions initiales et de comparer les deux schémas numériques, il semble utile de définir une fonction* `integration(x0, v0, methode)` *où l'argument* `methode` *peut prendre les valeurs* `euler` *(fonction définie au* **[Q 6.c.]***) et* `leapfrog` *(fonction définie au* **[Q 9.]***).*

```
def integration(x0, v0, methode):
    X, V, E = methode(F, x0, v0)
    plt.plot(X, V)
    return X, V, E
```

Remarque.— *Pour Python, une fonction est une variable comme les autres, il est donc possible que l'argument d'une fonction soit une fonction. Pour cela, il est essentiel que toutes les fonctions qui peuvent être affectées à cet argument aient la même signature : même entrée et même sortie !*

Q 11. *Il s'agit de calculer la distance entre deux vecteurs au sens de la norme* $\|\cdot\|_\infty$.

Q 11.a *Version pythonienne — déconseillée aux concours...*

```
def ema(d, dref):
    return np.max(np.abs(d-dref))
```

Q 11.b *Version pédestre... pour les nostalgiques de la programmation à l'ancienne. Un grand classique à savoir écrire vite et sans réfléchir.*

```
def ema(d, dref):
    n = len(d)
    # initialisation
    max_courant = abs(d[0] - dref[0])
    # itération
    for i in range(1, n):
        delta = abs(d[i] - dref[i])
        if (delta>max_courant):
            max_courant = delta
    return max_courant
```

I.2 Analyse des résultats

Q 12. **Durée caractéristique**

Q 12.a *Les schémas étudiés reposent tous les deux sur la Formule de Taylor.*
Pour que les applications numériques aient au moins l'ombre une chance de donner une approximation correcte du système réel, il faut que les approximations effectuées en passant de la Formule de Taylor aux schémas numériques soient raisonnables, c'est-à-dire que le pas de temps Δt *soit assez petit. Toute la question est alors de savoir* petit devant quelle valeur de référence ?

Q 12.b *L'oscillateur du second ordre étudié ici possède une* **période propre** *:*

$$\tau = \frac{2\pi}{\omega_0}$$

qui est la période des oscillations en l'absence de frottements et d'excitation. C'est cette durée qu'on retient généralement comme valeur temporelle de référence.
Avec les données numériques de l'énoncé, $\tau \approx 8$s *et les valeurs du pas temporel* Δt *fournies, toutes inférieures au dixième de seconde, sont très sensiblement inférieures à la période propre* τ.
Les approximations effectuées par les schémas numériques sont donc raisonnables.

Q 12.c *Cela ne signifie pas pour autant que les résultats fournis par ces schémas soient forcément* précis *!*
On vient de vérifier que, à chaque étape *de la discrétisation, les approximations sont raisonnables. On ne peut en aucune manière en déduire que les résultats obtenus soient* globalement *fidèles à la réalité.*

Q 13. Ordre d'un schéma numérique
*Question à caractère culturel : l'***ordre** *d'un schéma numérique est une notion capitale mais étrangère au programme... ce qui n'empêche pas le rapport du jury de cingler les candidats à ce propos !*

Q 13.a Définition
*On dit qu'un schéma numérique est d'***ordre** α *lorsque le pas temporel* Δt *et l'erreur numérique absolue* EMA *sont reliés par*

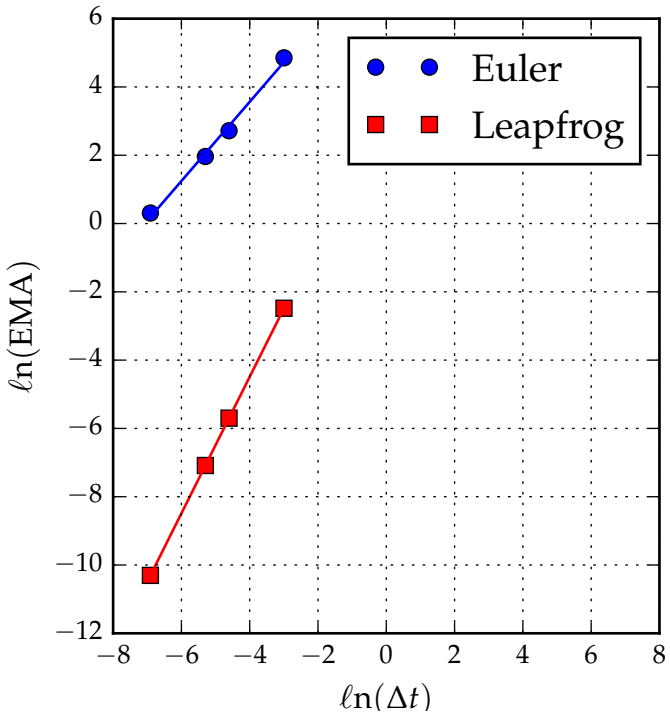
$$\mathrm{EMA} = \mathcal{O}\big((\Delta t)^\alpha\big).$$

Q 13.b *Pour comparer l'erreur moyenne absolue* EMA *au modèle* $A\cdot(\Delta t)^\alpha$*, nous allons procéder à un* ajustement linéaire *en expliquant* $\ell\mathrm{n}(\mathrm{EMA})$ *par* $\ell\mathrm{n}(\Delta t)$ *puisque*

$$\mathrm{EMA} = A\cdot(\Delta t)^\alpha \qquad \Longleftrightarrow \qquad \underbrace{\ell\mathrm{n}(\mathrm{EMA})}_{y} = \alpha\cdot\underbrace{\ell\mathrm{n}(\Delta t)}_{x} + \ell\mathrm{n}\,A.$$

Q 13.c *Il suffit de demander à Python (mais toute calculatrice sait faire aussi).*

```
from scipy import stats
X = np.log([0.05, 0.01, 0.005, 0.001])
Y = np.log([128.02, 15.14, 7.12, 1.36])
Z = np.log([0.0837, 0.00335, 0.000837, 0.0000335])
a1, b1, r1, p_val, sigma = stats.linregress(X, Y)
a2, b2, r2, p_val, sigma = stats.linregress(X, Z)
```



REMARQUE.— *On voit bien que c'est un matheux qui a fait les calculs :* aucun *souci des dimensions ! L'ajustement linéaire aurait dû être présenté sous la forme suivante :*

$$\ell\mathrm{n}\,\frac{\mathrm{EMA}}{\mathrm{EMA}_0} = \alpha\cdot\ell\mathrm{n}\,\frac{\Delta t}{\Delta t_0} + \ell\mathrm{n}\,\frac{A}{A_0}.$$

Cela ne change évidemment pas la valeur de α *— ce qui n'est pas une raison suffisante pour faire n'importe quoi !*

Q 13.d *On obtient ainsi $\alpha \approx 1,16$ et $r \approx 0,9973$ pour le schéma d'Euler (schéma d'ordre 1, erreur moyenne en $\mathcal{O}(\Delta t)$) et $\alpha \approx 2,00$ et $r \approx 0,9999$ pour le schéma Leapfrog (schéma d'ordre 2, erreur moyenne en $\mathcal{O}[(\Delta t)^2]$).*
En traçant les droites de régression, on constate l'alignement quasi-parfait des points.

Q 13.e Variante
On peut aussi estimer grossièrement l'ordre de chaque schéma par un calcul de proportions.

Schéma d'Euler					
Δt_1 (ms)	Δt_2 (ms)	*Proportion*	EMA_1 (J)	EMA_2 (J)	*Proportion*
10	5	2	15	7	2
5	1	5	7	1.5	5
Ordre probable : 1					

Schéma Leapfrog					
Δt_1 (ms)	Δt_2 (ms)	*Proportion*	EMA_1 (mJ)	EMA_2 (mJ)	*Proportion*
50	5	10	83	0,83	100
10	1	10	3	0,03	100
5	1	5	0,8	0,03	26
Ordre probable : 2					

Ces calculs peuvent se faire de tête, c'est leur principal mérite, cette méthode mérite d'être retenue.

Q 14. *Dans le schéma d'Euler sans amortissement ni excitation,*

$$\begin{cases} x_{n+1} = x_n + v_n \cdot \Delta t \\ v_{n+1} = v_n - x_n \cdot \omega_0^2 \Delta t \end{cases} \qquad et \qquad E_n = \frac{1}{2}\left(k \cdot x_n^2 + m \cdot v_n^2\right).$$

On en déduit que

$$\begin{aligned} E_{n+1} - E_n &= \frac{1}{2}k(x_{n+1}^2 - x_n^2) + \frac{1}{2}m(v_{n+1}^2 - v_n^2) \\ &= \underbrace{(k - m\omega_0^2)}_{=0} x_n v_n \cdot \Delta t + \frac{1}{2}(k v_n^2 + m\omega_0^4 x_n^2) \cdot (\Delta t)^2 = (\omega_0 \Delta t)^2 \cdot E_n \end{aligned}$$

puisque $\omega_0 = \sqrt{k/m}$.
La suite (E_n) est donc une suite géométrique : $E_n = (1 + \omega_0 \Delta t)^n E_0$ alors que la fonction $E(t)$ est constante (sans amortissement ni excitation, le système est conservatif). Autrement dit,

$$\begin{aligned} \text{EMA} &= \max_{0 \leqslant n < N} \left|(1 + \omega_0 \Delta t)^n - 1\right| \times E_0 \\ &\approx \left|(1 + \omega_0 \Delta t)^N - 1\right| \times E_0 \qquad (N \text{ ou } (N-1) \text{... who cares?}) \\ &\sim [\omega_0 (N \Delta t) \cdot E_0] \cdot \Delta t = \underbrace{(\omega_0 \cdot t_{\max} \cdot E_0)}_{Cte} \cdot \Delta t. \qquad (\textit{équivalent archi-classique}) \end{aligned}$$

On a ainsi vérifié que le schéma d'Euler était bien un schéma numérique du premier ordre, conformément aux observations numériques de la question précédente.

II

Intelligence artificielle en médecine — CCINP 2019 PSI

II.1 Analyse des données

Q 1. *Sélection simple.*

```
SELECT idpatient FROM MEDICAL
WHERE etat = 'hernie discale'
```

Q 2. *Sélection avec jointure : une sélection simple permet d'accéder à l'identifiant du patient, la jointure sert à récupérer son identité (prénom et nom).*

```
SELECT P.nom, P.prenom FROM PATIENT AS P
JOIN MEDICAL AS M
ON P.id = M.idpatient
WHERE M.etat = 'spondylolisthésis'
```

Remarque.— *C'est de la folie de coder l'état des malades par des chaînes de caractères ! Une simple faute de frappe lors de la saisie et on perd le patient...*

Q 3. *Une sélection avec fonction d'agrégation.*

```
SELECT etat, COUNT(idpatient) FROM MEDICAL
GROUP BY etat
```

Q 6. *On peut mettre en œuvre une méthode très pythonienne pour séparer les valeurs du tableau* `data` *en trois groupes en discutant sur les entrées du vecteur* `etat`.

```
def separationParGroupes(data, etat):
    T = [ [], [], [] ] # liste de listes
    N = len(data)      # nombre d'enregistrements à classer
    for e in range(3):  # Pour chaque état possible,
        # on extrait les données data[i] en discutant sur la valeur de etat[i].
        T[e] = [ data[i] for i in range(N) if etat[i]==e ]
    return T
```

*On peut aussi proposer une variante plus efficace, en ce qu'elle ne fait parcourir qu'*une seule fois *le tableau* `data` *et tire un parti maximal du codage des différents états possibles par les entiers* 0, 1 *et* 2.

```
def separationParGroupes(data, etat):
    T = [ [], [], [] ]
    N = len(data)
    for i in range(N):
        d, e = data[i], etat[i]
        T[e].append(d)             # futé, non ?
    return T
```

Cette variante pourrait être pythonisée facilement.

```
def separationParGroupes(data, etat):
    T = [ [], [], [] ]
    for (d, e) in zip(data, etat):  # épatant, non ?
        T[e].append(d)
    return T
```

II.2 Apprentissage et prédiction par la méthode KNN

Préparation des données

Q 9. *Pour normaliser x_k en fonction des valeurs extrêmes du vecteur X :*

$$\frac{x_k - \min(X)}{\max(X) - \min(X)}.$$

(C'est un calcul de proportion ! Rien d'autre à dire pour justifier sa réponse...)

Q 10. *On réinvente la roue : ce n'est pas palpitant, mais c'est la règle des concours !*

```
def min_max(X):
    mini, maxi = X[0], X[0] # valeurs initiales
    for x in X:             # Pas besoin d'indice ici !
        if x<mini:
            mini = x        # valeur actualisée du minimum
        elif x>maxi:        # on évite des tests inutiles
            maxi = x        # valeur actualisée du maximum
    return mini, maxi
```

La complexité de ce code est bien linéaire : on parcourt une seule fois le vecteur `X` *et à chaque étape, on effectue au plus deux tests et une affectation, ce qui nous donne au plus* $3N$ *opérations.*

Q 11.a *Deux manières : la première, pédestre, est probablement ce que le jury attend.*

```
def distance_vect(A, B):
    n = len(A)
    S = 0
    for i in range(n):
        S = S + (A[i]-B[i])**2
    return np.sqrt(S)

def distance2(z, data):
    N = len(data) # nombre de lignes = nombre d'enregistrements
    L = []
    for i in range(N):
        d = distance_vect(z, data[i])
        L.append(d)
    return L
```

Q 11.b *La seconde, très pythonienne, est peut-être trop moderne pour les concours... (Mais pourquoi continuer à coder à l'ancienne quand on peut développer les mêmes qualités d'analyse et d'imagination en se débarrassant des tâches pénibles ?)*

```
def distance_eucl(x, z):
    return np.sqrt(np.sum((x-z)**2))

def distance(z, data):
    return [ distance_eucl(z, x) for x in data ] # parcours du tableau ligne par ligne
```

Détermination des K plus proches voisins

Q 12.a *La fonction* `tri` *est récursive.*
Si son argument est un tableau avec un seul élément (c'est-à-dire un tableau trié), le tableau est retourné sans modification.
Dans le cas contraire, le tableau est scindé en deux parties à peu près égales, qui sont triées avant qu'on leur applique `fct`.
Il s'agit donc du tri `fct`.
Plus sûrement, on essaie ici de faire reconnaître le tri fusion. *Mais comme le code de* `fct` *est incomplet, comment être sûr ?*

Q 12.b *Le tri insertion est un algorithme de référence, qu'il faut savoir coder. Cette question doit donc être considérée comme une question de cours.*

```
def tri_insertion(T):
    n = len(T)
    # Si n=1, alors le tableau T est déjà trié !
    if n>=2 :
        for i in range(1, n):
            tmp = T[i] # nouvel élément à ranger
            k = i
            while ((k>0) and plus_grand_que(T[k-1], tmp)):
                T[k] = T[k-1] # décalage vers la droite
                k = k-1
            T[k] = tmp  # on range l'élément à sa place
```

Ici, les éléments du tableau `T` *sont des listes et on doit trier ces listes par ordre croissant du premier élément. On en déduit le code suivant pour la fonction auxiliaire.*

```
def plus_grand_que(liste, liste_ref):
    return (liste[0]>liste_ref[0])
```

Q 12.c *Le tri fusion a une complexité temporelle moyenne en $\mathcal{O}(n \ell\mathrm{n}\, n)$. Il est donc plus efficace que le tri insertion, dont la complexité temporelle est en $\mathcal{O}(n^2)$.*
Cependant le tri fusion a une complexité spatiale bien plus élevée que le tri insertion : quand le tri insertion se limite à enregistrer la valeur d'un élément à chaque itération, le tri fusion recopie des parties de tableaux.

Remarque.— *Compte-tenu de l'objectif visé (le classement de K plus grandes valeurs et non pas le classement de toutes les valeurs), ces deux méthodes sont également inappropriées. Mieux vaudrait utiliser une variante du tri insertion qui se borne à trier les K plus grandes valeurs de* `T` *et dont la complexité est en $\mathcal{O}(N \times K)$.*

Q 13. *Comme on le sait, le tri fusion repose sur l'algorithme de fusion de deux listes ordonnées T_1 et T_2 qu'on va présenter comme deux files d'attente.*

Q 13.a *Dans un premier temps, aucune de ces deux files n'est vide. Si on a déjà rangé (par ordre croissant) dans une liste T les éléments*

$$T_1[0], \ldots, T_1[i-1] \quad \textit{ainsi que} \quad T_2[0], \ldots, T_2[k-1],$$

alors il nous reste à comparer $T_1[i]$ et $T_2[k]$ (lignes 15 et 18). Si $T_1[i]$ est la plus petite valeur (ligne 15), c'est elle qu'on place dans T. Sinon (ligne 18), c'est $T_2[k]$ qu'il faut placer dans T.
Il suffit donc d'adapter les lignes 16 et 17.

```
T.append(T2[k])
k = k+1
```

Q 13.b *Dans un second temps, l'une des deux files est vide. Cette file correspond à la liste T_1 lorsque i prend la valeur n_1 =*`len(T1)` *(ligne 21) ou à la liste T_2 lorsque k prend la valeur n_2, c'est-à-dire tant que i n'est pas égal à n_1 (ligne 24).*
Il suffit cette fois encore d'adapter le code précédent (lignes 22 et 23).

```
for j in range(i, n1):
    T.append(T1[j])
```

Q 14.a Partie 1
Dans un premier temps, on calcule la distance de l'enregistrement `z` *aux enregistrements connus dans le tableau* `data`*. Ces distances sont placées dans la liste* `dist`*.*
On établit ensuite la liste T contenant les distances calculées et les indices des différents enregistrements du tableau `data` *(ces enregistrements ne sont donc pas dupliqués, c'est sage). Cette liste T est triée par ordre croissant de distance.*

Q 14.b Partie 2
On parcourt ensuite les K premiers éléments de la liste T. Pour le i-ème élément `T[i]`*, la valeur de* `T[i][1]` *est l'indice de l'enregistrement correspondant dans le tableau* `data` *et* `etat[T[i][1]]` *indique alors l'état de santé du patient. (Cf* **[Q 6.]***)*
À l'issue du parcours, on a dénombré les différents états parmi les K premiers éléments de la liste T. Les différents effectifs sont regroupés dans la liste `select`*, la valeur de* `select[e]` *donnant le nombre d'enregistrements avec l'état* `e` *parmi les K premiers éléments de T.*

Q 14.c Partie 3

Le paramètre `nb` *donne le nombre d'états possibles (pour nous :* `nb`$= 3$*). On reconnaît ici l'algorithme classique de localisation du maximum d'une liste : non seulement on cherche le maximum (= la valeur finale de* `res`*) mais on retient aussi avec* `ind` *l'indice où ce maximum est atteint. C'est cet indice qui est la valeur retournée par* `KNN` *: la fonction calcule donc l'état le plus fréquemment observé parmi les K plus proches voisins de l'enregistrement* `z`*.*

Validation de l'algorithme

Q 15. Matrice de confusion

Commençons par analyser la fonction `test_KNN`*.*

Dans un premier temps, on construit la liste `etatpredit` *en comparant à l'aide de la fonction* `KNN` *les différents enregistrements de la liste* `datatest` *au tableau complet* `data`*.*

Dans un second temps, on calcule la matrice de confusion*, que nous noterons B (comme Babel).*

Pour chaque enregistrement de la liste `datatest`*, on compare l'état réel du patient, connu avec la liste* `etattest`*, et l'état supposé du patient, calculé plus tôt avec* `KNN`*. On réaliste ici un* comptage *(initialisation à* 0*, incrémentation d'une unité pour chaque donnée).*

Q 15.a Coefficients diagonaux

Le coefficient diagonal $B_{i,i}$ *de la matrice de confusion donne donc le nombre d'enregistrements présents dans la liste* `datatest` *pour lesquels l'état réel et l'état supposé sont tous les deux égaux à i.*

L'algorithme `KNN` *a donc correctement évalué l'état de santé de chacun de ces patients.*

La trace de la matrice de confusion est donc le nombre *total d'évaluations correctes. Comme la liste* `datatest` *contient en tout* 100 *enregistrements, la trace est aussi égale au pourcentage d'évaluations correctes (ici 74%), appelé* taux de réussite *dans la suite de l'énoncé.*

Q 15.b *Sur la première ligne, les coefficients* $B_{0,1}$ *et* $B_{0,2}$ *sont les nombres de patients en bonne santé (*`etattest` *est égal à* 0*) qui ont été évalués comme malades (*`etatpredit` *égal à* 1 *ou* 2*).*

La somme $B_{0,1} + B_{0,2} = 11$ *donne donc le nombre total de* faux positifs*.*

Q 15.c *Sur la première colonne, les coefficients* $B_{1,0}$ *et* $B_{2,0}$ *sont les nombres de patients malades (*`etattest` *égal à* 1 *ou* 2*) qui n'ont pas été identifiés comme malades par l'algorithme (*`etatpredit` *égal à* 0*).*

La somme $B_{1,0} + B_{2,0} = 12$ *donc cette fois le nombre total de* faux négatifs*.*

Q 15.d *Suivant la même logique, la somme* $B_{1,2} + B_{2,1} = 3$ *des autres coefficients hors diagonale est le nombre de cas positifs avec une erreur de diagnostic (l'état prédit par l'algorithme n'est pas l'état réel du patient).*

La matrice de confusion permet d'évaluer l'efficacité de l'algorithme `KNN`*.*

Q 16. *Le taux de réussite de l'algorithme reste compris entre* 70 *et* 75%*, à peu près indépendamment de la valeur de K. Cela suggère d'utiliser l'algorithme avec une valeur moyenne de K (environ* 10*) pour obtenir un résultat rapide et assez fiable.*

En tout état de cause, l'algorithme doit rester au service du médecin — pas l'inverse. *Il ne devrait même pas y avoir matière à polémique à ce sujet.*

Complément
C'est propre à la logique des sujets que de fournir des codes un peu obscurs et pas bien ficelés. Je vous propose une réécriture de la fonction `test_KNN` inspirée par la lecture de *Coder proprement* de Robert C. Martin (Pearson, 2019).
Un bon code ne doit calculer qu'une chose à la fois. Alors scindons !
Les noms de variables et les noms de fonctions doivent permettre de lire le code comme un texte normal, sans avoir à réfléchir à la signification de tel ou tel identifiant. La complétion automatique, qui est en service sur tous les éditeurs de texte modernes, est très pratique pour atteindre ce but !
Les indices dans les boucles `for`... mais ce n'est plus possible ! Si on a vraiment besoin de la valeur de l'indice, utilisons un indice. Mais ici les indices ne servent à rien — sinon à masquer le véritable objet sur lequel on travaille.

```
def etats_predits(donnees_ref, donnees_globales, etat_global, K, nb_etats):
   return [KNN(donnees_globales, etat_global, rf, K, nb_etats) for rf in donnees_ref]

def matrice_confusion(etats_ref, etats_predits, nb_etats):
   B = np.zeros((nb_etats, nb_etats))
   for (i, j) in zip(etats_ref, etats_predits):
       B[i, j] += 1
   return B

def test_KNN(donnees_ref, etats_ref, donnees_globales, etat_global, K, nb_etats):
    EP = etats_predits(donnees_ref, donnees_globales, etat_global, L, nb_etats)
    return matrice_confusion(etats_ref, EP, nb_etats)
```

Toujours dans un but de lisibilité, le nombre d'arguments ne devrait pas dépasser 3...
– On pourrait définir `nb_etats` comme une variable globale (quel intérêt de mettre une constante en argument ?).
– Pourquoi isoler les `donnees_globales` de la liste `etat_global` ? Ce sont les mêmes patients ! On pourrait définir une structure regroupant toutes les données, à la manière d'une base de données. La programmation objet est pensée pour cet objectif.

Autrement dit, il faudrait ici reprendre toute la construction du code.
Produire un beau code est passionnant !