teaching sciences for innovation

Proposition de corrigé

Concours : Banque PT

Année : 2019

Filière : PT

Épreuve : Informatique et Modélisation

Ceci est une proposition de corrigé des concours de CPGE, réalisée bénévolement par des enseignants de Sciences Industrielles de l'Ingénieur et d'Informatique, membres de l'UPSTI (Union des Professeurs de Sciences et Techniques Industrielles), et publiée sur le site de l'association :

https://www.upsti.fr/espace-etudiants/annales-de-concours

A l'attention des étudiants

Ce document vous apportera des éléments de corrections pour le sujet traité, mais n'est ni un corrigé officiel du concours, ni un corrigé détaillé ou exhaustif de l'épreuve en question.

L'UPSTI ne répondra pas directement aux questions que peuvent soulever ces corrigés : nous vous invitons à vous rapprocher de vos enseignants si vous souhaitez des compléments d'information, et à vous adresser à eux pour nous faire remonter vos éventuelles remarques.

Licence et Copyright

Informez-vous !

Retrouvez plus d'information sur les Sciences de l'Ingénieur, l'orientation, les Grandes Ecoles ainsi que sur les Olympiades de Sciences de l'Ingénieur et sur les Sciences de l'Ingénieur au Féminin sur notre site : www.upsti.fr

L'équipe UPSTI

Épreuve d'Informatique et Modélisation
Banque PT Session 2019

UPSTI

Corrigé

Corrigé Epreuve d'info banque PT 2019

I Figures de Chladni

I.1 Introduction du modèle physique (∼10%)

1°)

L'équation proposée est l'équation de d'Alembert. C'est une équation linéaire. Il existe une équation analogue en électromagnétisme et dans la modélisation de la corde de Melde.

2°)

Les pertes d'énergie se caractérisent par un terme du premier ordre dans les équations différentielles (amortissement). Ce n'est pas le cas ici, donc cette équation n'est pas le reflet d'un modèle physique autorisant des pertes d'énergie.

3°)

On a Δf qui est homogène à $\frac{\partial^2 f}{\partial x^2}$ donc des m^{-1}.

$\frac{\partial^2 f}{\partial t^2}$ étant homogène à des $m.s^{-2}$, c est donc homogène à une vitesse en $m.s^{-1}$. La célérité de l'onde ne peut pas dépendre des dimensions de la plaque, mais de ses paramètres matériaux que sont ρ et E.

E est en Pa soit $N.m^{-2}$ ou encore $kg.s^{-2}.m^{-1}$ et ρ est en $kg.m^{-3}$.

On peut donc proposer une expression de c :

$$c = \sqrt{\frac{E}{\rho}}$$

4°) Application numérique :

$$c = \sqrt{\frac{69 \cdot 10^9}{2{,}70 \cdot 10^3}} \simeq 5 \cdot 10^3 m.s^{-1}$$

5°)

a) $f_L(x, y, t)$ est solution de l'équation pour une plaque de côté L et $f_L(\alpha x, \alpha y, \beta t)$ est solution de l'équation pour une plaque de côté γL.

On en déduit $\alpha = \gamma$

b) $\tau' = \gamma\tau$

I.2 Modes de vibration (∼5%)

6°)

a) Avec $f(x,y,z) = u(x,y) \times h(t)$, l'équation différentielle devient :

$$h(t) \cdot \left(\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} \right) = \frac{1}{c^2} \cdot u(x,y) \cdot \frac{d^2 h(t)}{dt^2}$$

$$\Leftrightarrow \frac{\ddot{h}(t)}{h(t)} = \frac{c^2}{u(x,y)} \cdot \left(\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} \right)$$

Comme on a égalité entre une partie ne dépendant que de t et une partie ne dépendant que de (x,y), les deux termes doivent être constants :

$$\frac{\ddot{h}}{h} = \frac{c^2}{u(x,y)} \cdot \left(\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} \right) = K$$

$$\ddot{h} - Kh = 0 \qquad \text{et} \qquad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{Ku}{c^2}$$

b) La solution de $\ddot{h} - Kh = 0$ dépend du signe de K.

- $K > 0 \Rightarrow h(t) = A \cdot e^{\sqrt{K}t} + B \cdot e^{-\sqrt{K}t}$
  Solution impossible, le premier terme est divergent et le second tend vers 0
- $K = 0 \Rightarrow \ddot{h} = 0 \Leftrightarrow h(t) = A + Bt$ Solution impossible également car divergente ou constante.
- $K < 0 \Rightarrow h(t) = A\cos\omega t + B\sin\omega t$ avec $\omega^2 = -K$ Seules solutions acceptables.

c)

$$h(t) = h_0 \exp(i\omega t) \Rightarrow \frac{d^2 h(t)}{dt^2} = -\omega^2 h_0 \exp(i\omega t) = -\omega^2 h(t)$$

L'équation en $u(x,y)$ devient donc :

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = \frac{-\omega^2}{c^2} \cdot u(x,y)$$

Le résultat aurait été le même en posant $h(t) = A \cdot \cos\omega t + B \cdot \sin\omega t$.
On a $\frac{d^2 h(t)}{dt^2} = -\omega^2 \cdot (A \cdot \cos\omega t + B \cdot \sin\omega t) = -\omega^2 h(t)$

I.3 Simulation numérique temporelle (∼20%)

```
In [1]: # -*-coding:Utf-8 -*

        #Librairies utilisées et alias
        import numpy as np
        import matplotlib.pyplot as plt
        import numpy.random as rd
```

7°) Méthode d'Euler

a) Le principe de la méthode d'Euler pour résoudre numériquement une équation différentielle du premier ordre est le suivant :

On approxime la dérivée par le taux d'accroissement entre 2 points : $f'(t_n) = \dfrac{f(t_{n+1}) - f(t_n)}{t_{n+1} - t_n}$

Ainsi on écrit $f(t_{n+1}) = f(t_n) + h \cdot f'(t_n)$ avec $h = t_{n+1} - t_n$ et $f'(t_n)$ donné par l'équation différentielle en fonction de t_n, $f(t_n)$ et les paramètres du problème.

b)

$$X(t) = \begin{pmatrix} h(t) \\ h'(t) \end{pmatrix} \Rightarrow \frac{dX}{dt}(t) = \begin{pmatrix} h'(t) \\ h''(t) \end{pmatrix}$$

On peut donc écrire sous forme matricielle :

$$\begin{pmatrix} h'(t) \\ h''(t) \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{pmatrix} h(t) \\ h'(t) \end{pmatrix}$$

c) On pose $\tau = t_{n+1} - t_n$. On a donc

$$X[n+1] = X[n] + \tau \cdot AX[n] \Leftrightarrow X[n+1] = \begin{bmatrix} 1 & \tau \\ -\tau & 1 \end{bmatrix} \cdot X[n]$$

8°) Méthode d'Euler à droite

$$X(t_n + \tau) = X(t_n) + \int_{t_n}^{t_n+\tau} \frac{dX(t)}{dt} dt \approx X(t_n) + \tau \cdot \frac{dX(t_n + \tau)}{dt}$$

a)

$$X[n+1] = X[n] + \tau \cdot AX[n+1] \Leftrightarrow \begin{bmatrix} 1 & -\tau \\ \tau & 1 \end{bmatrix} \cdot X[n+1] = X[n] \Leftrightarrow X[n+1] = \frac{1}{1+\tau^2} \begin{bmatrix} 1 & \tau \\ -\tau & 1 \end{bmatrix} \cdot X[n]$$

b)

```
In [2]: # Code modifié pour répondre à la question
        # Constantes paramétriques
        npoints=2**10 # index maximal des tableaux
        tporte=5*2*np.pi # durée d'affichage
        X0=[1.,0] # conditions initiales

        #Initialisation des tableaux
        t=np.linspace(0,tporte,npoints+1) # temps
        tau=t[1]-t[0]
        X=np.zeros((2,npoints+1),dtype=float) # h et h'

        # Algorithme d'Euler
        X[:,0]=X0 # initialisation
        M=np.array([[1,tau],[-tau,1]]) # matrice de l'equation
        for i in range(npoints): # Calcul iteratif
```

```
        X[:,i+1]=1 / (1 + tau**2) * M.dot(X[:,i]) # ligne modifiee pour repondre à la questi
```

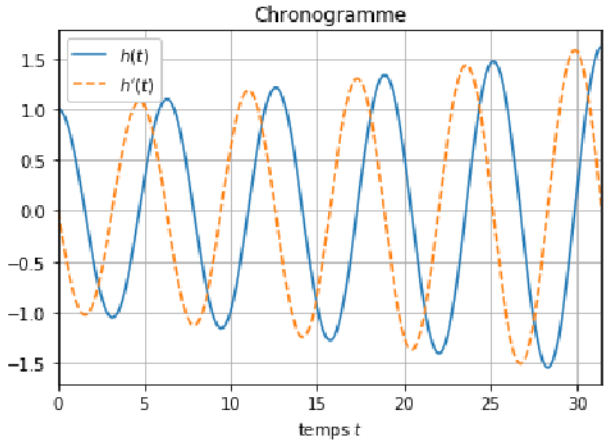
```
In [3]: # Code initial non modifié pour le tracé graphique
        # Constantes paramétriques
        npoints=2**10 # index maximal des tableaux
        tporte=5*2*np.pi # durée d'affichage
        X0=[1.,0] # conditions initiales

        #Initialisation des tableaux
        t=np.linspace(0,tporte,npoints+1) # temps
        tau=t[1]-t[0]
        X=np.zeros((2,npoints+1),dtype=float) # h et h'

        # Algorithme d'Euler
        X[:,0]=X0 # initialisation
        M=np.array([[1,tau],[-tau,1]]) # matrice de l'equation
        for i in range(npoints): # Calcul iteratif
            X[:,i+1]=M.dot(X[:,i])
```

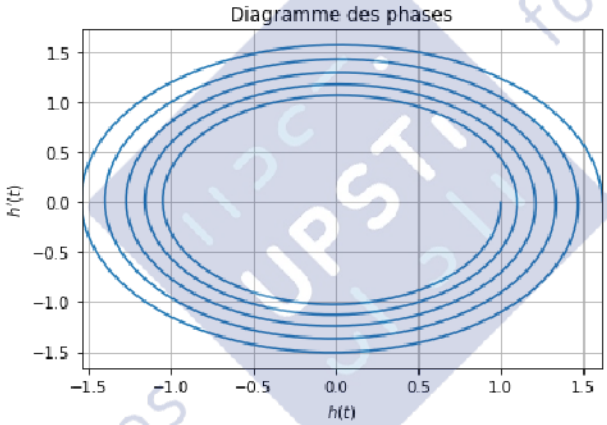
c) La méthode d'Euler à droite est du même ordre que la méthode d'Euler à gauche. Ici elle n'est pas plus pertinente, ni moins pertinente que la méthode d'Euler à gauche.

9°) Représentation graphique

a)

```
In [4]: plt.figure()
        plt.xlim(t[0], t[-1])
        plt.plot(t, X[0, :], label="$h(t)$")
        plt.plot(t, X[1, :], '--', label="$h'(t)$")
        plt.legend(loc='upper left', framealpha=1)
        plt.xlabel('temps $t$')
        plt.title('Chronogramme')
        plt.grid()

        plt.figure()
        plt.xlim(min(X[0, :]), max(X[0, :]))
        plt.plot(X[0, :], X[1, :])
        plt.xlabel('$h(t)$')
        plt.ylabel("$h'(t)$")
        plt.title('Diagramme des phases')
        plt.grid()

        plt.show()
```



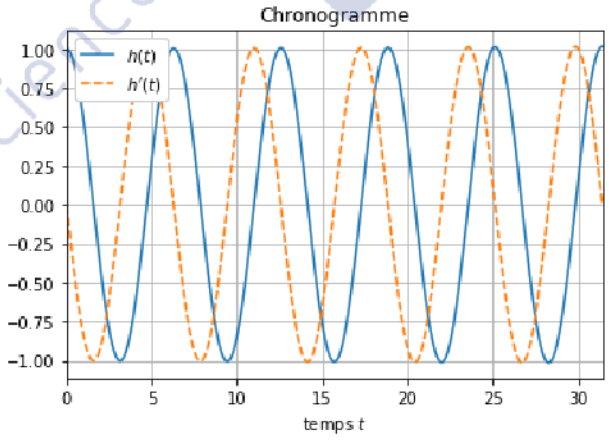


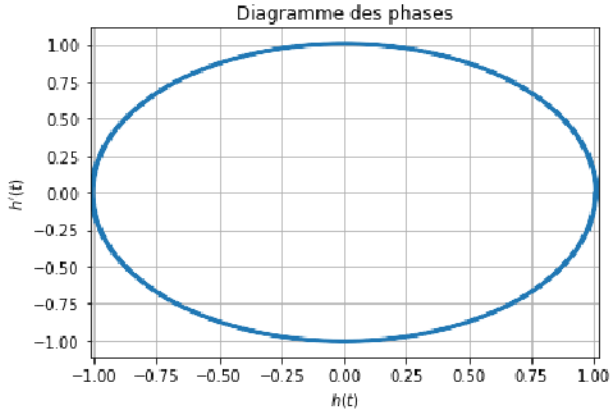
b) On a vu que les solutions devaient être sinusoïdales q6b), le chronogramme de h et h' devraient être sinusoïdaux et le diagramme de phase devrait être elliptique, ce qui n'est pas le cas ici.

c) On a augmenté le nombre de points pour diminuer le pas de discrétisation temporelle. L'inconvénient principal est l'explosion du temps de calcul.

```
In [5]: # Code initial non modifié pour le tracé graphique
        # Constantes paramétriques
        npoints=2**15 # index maximal des tableaux
        tporte=5*2*np.pi # durée d'affichage
        X0=[1.,0] # conditions initiales

        #Initialisation des tableaux
        t=np.linspace(0,tporte,npoints+1) # temps
```

```
tau=t[1]-t[0]
X=np.zeros((2,npoints+1),dtype=float) # h et h'

# Algorithme d'Euler
X[:,0]=X0 # initialisation
M=np.array([[1,tau],[-tau,1]]) # matrice de l'equation
for i in range(npoints): # Calcul iteratif
    X[:,i+1]=M.dot(X[:,i])

# Affichage non demandé dans l'énoncé
plt.figure()
plt.xlim(t[0], t[-1])
plt.plot(t, X[0, :], label="$h(t)$")
plt.plot(t, X[1, :], '--', label="$h'(t)$")
plt.legend(loc='upper left', framealpha=1)
plt.xlabel('temps $t$')
plt.title('Chronogramme')
plt.grid()

plt.figure()
plt.xlim(min(X[0, :]), max(X[0, :]))
plt.plot(X[0, :], X[1, :])
plt.xlabel('$h(t)$')
plt.ylabel("$h'(t)$")
plt.title('Diagramme des phases')
plt.grid()

plt.show()
```





10°) Méthode de Runge-Kutta d'ordre 2

$$X(t_n+\tau) = X(t_n) + \int_{t_n}^{t_n+\tau} \frac{dX(t)}{dt} dt$$

$$X(t_n+\tau) \approx X(t_n) + \frac{\tau}{2} \cdot \left(\frac{dX(t_n+\tau)}{dt} + \frac{dX(t_n)}{dt} \right)$$

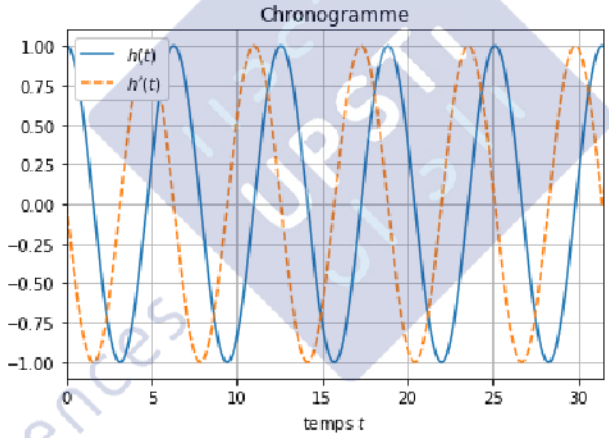
a) La méthode d'approximation est la méthode des trapèzes. Les méthodes précédentes sont les méthodes des rectangles à gauche et à droite.

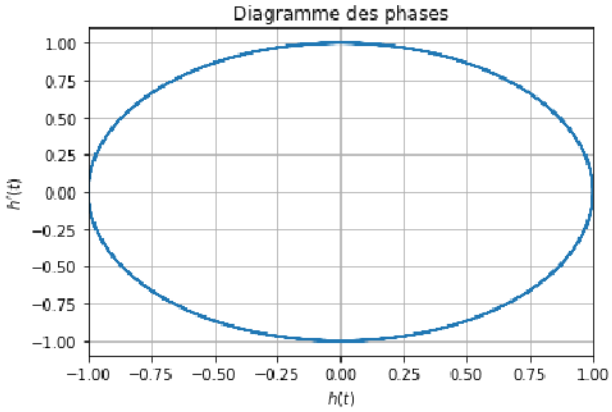
La méthode des trapèzes (ordre 2) a une erreur d'approximation plus faible que les méthodes des rectangles (ordre 1).

b)

```
In [6]: # Constantes paramétriques
        npoints=2**10 # index maximal des tableaux
        tporte=5*2*np.pi # durée d'affichage
        X0=[1.,0] # conditions initiales

        #Initialisation des tableaux
        t=np.linspace(0,tporte,npoints+1) # temps
        tau=t[1]-t[0]
        X=np.zeros((2,npoints+1),dtype=float) # h et h'

        # Algorithme de Range-Kutta d'ordre 2
        X[:,0]=X0 # initialisation
        A = np.array([[0, 1], [-1, 0]]) # matrice A
        for i in range(npoints): # Calcul iteratif
            K1 = A.dot(X[:,i])
            K2 = A.dot(X[:,i] + tau * K1)
            X[:,i+1] = X[:,i] + tau / 2 * (K1 + K2)
```

```
plt.figure()
plt.xlim(t[0], t[-1])
plt.plot(t, X[0, :], label="$h(t)$")
plt.plot(t, X[1, :], '--', label="$h'(t)$")
plt.legend(loc='upper left', framealpha=1)
plt.xlabel('temps $t$')
plt.title('Chronogramme')
plt.grid()

plt.figure()
plt.xlim(-1, 1)
plt.plot(X[0, :], X[1, :])
plt.xlabel('$h(t)$')
plt.ylabel("$h'(t)$")
plt.title('Diagramme des phases')
plt.grid()

plt.show()
```

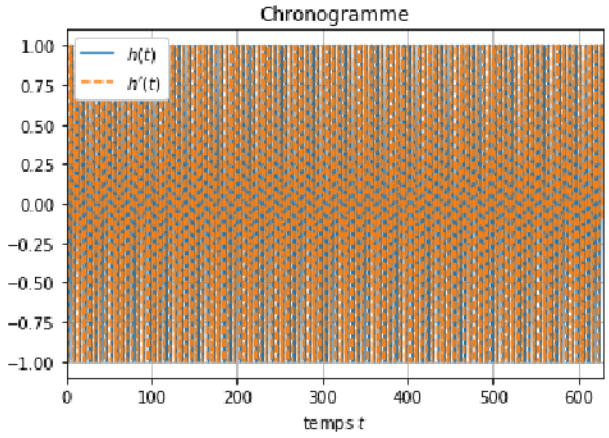


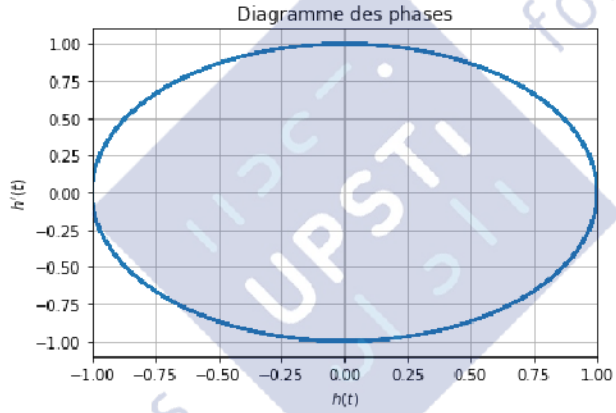


c) Visuellement les chronogrammes sont sinusoïdaux et le diagramme de phase elliptique. On peut donc conclure que la méthode de Range-Kutta est meilleure que la méthode d'Euler.

Nous n'avons pas assez d'information pour voir si cette méthode est vraiment stable. En affichant le chronogramme sur une durée beaucoup plus longue, on constate que la solution numérique reste sinusoïdale en gardant un pas de temps constant (voir ci-dessous). En revanche elle ne l'est pas en augmentant le pas de temps

```
In [7]: # Constantes paramétriques
        npoints=2**10 * 20 # index maximal des tableaux
        tporte=100*2*np.pi # durée d'affichage plus longue
        X0=[1.,0] # conditions initiales

        #Initialisation des tableaux
        t=np.linspace(0,tporte,npoints+1) # temps
        tau=t[1]-t[0]
        X=np.zeros((2,npoints+1),dtype=float) # h et h'

        # Algorithme de Range-Kutta d'ordre 2
        X[:,0]=X0 # initialisation
        A = np.array([[0, 1], [-1, 0]]) # matrice A
        for i in range(npoints): # Calcul iteratif
            K1 = A.dot(X[:,i])
            K2 = A.dot(X[:,i] + tau * K1)
            X[:,i+1] = X[:,i] + tau / 2 * (K1 + K2)

        plt.figure()
        plt.xlim(t[0], t[-1])
        plt.plot(t, X[0, :], label="$h(t)$")
        plt.plot(t, X[1, :], '--', label="$h'(t)$")
        plt.legend(loc='upper left', framealpha=1)
        plt.xlabel('temps $t$')
        plt.title('Chronogramme')
        plt.grid()

        plt.figure()
        plt.xlim(-1, 1)
        plt.plot(X[0, :], X[1, :])
        plt.xlabel('$h(t)$')
        plt.ylabel("$h'(t)$")
        plt.title('Diagramme des phases')
        plt.grid()

        plt.show()
```

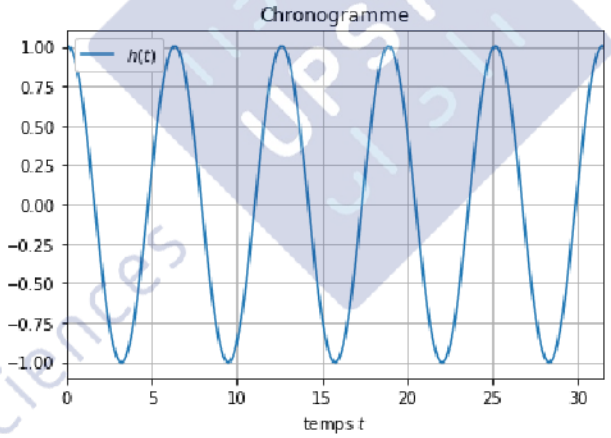


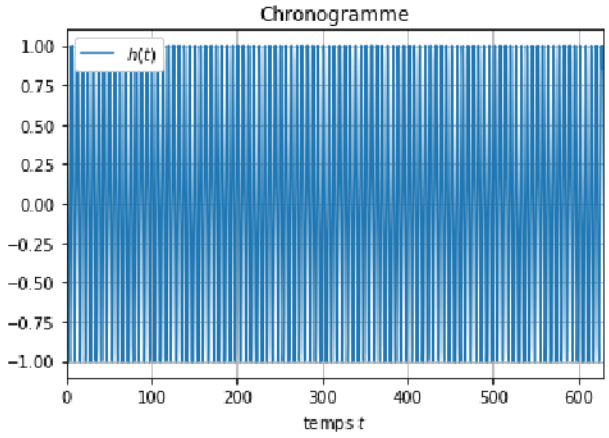


11°) Méthode stable

```
In [8]: # Constantes paramétriques
        npoints=2**12 # index maximal des tableaux modifié pour correspondre aux tracés
        tporte=100*2*np.pi # durée d'affichage
        X0=[1.,0] # conditions initiales

        #Initialisation des tableaux
        t=np.linspace(0,tporte,npoints+1) # temps
        tau=t[1]-t[0]
        X=np.zeros((2,npoints+1),dtype=float) # h et h'

        # Algorithme XXX
        X[:,0]=X0 # initialisation
        A=np.array([[0, 1], [-1, -tau]]) # matrice de couplage
        M=np.identity(2)+tau*A # matrice d'evolution
```

```
for i in range(npoints): # Calcul iteratif
    X[:,i+1] = M.dot(X[:,i])

plt.figure()
plt.xlim(t[0], t[205])
plt.plot(t[:206], X[0, :206], label="$h(t)$")
plt.legend(loc='upper left', framealpha=1)
plt.xlabel('temps $t$')
plt.title('Chronogramme')
plt.grid()

plt.figure()
plt.xlim(t[0], t[-1])
plt.plot(t, X[0, :], label="$h(t)$")
plt.legend(loc='upper left', framealpha=1)
plt.xlabel('temps $t$')
plt.title('Chronogramme')
plt.grid()

plt.show()
```





a)

$$A = \begin{bmatrix} 0 & 1 \\ -1 & -\tau \end{bmatrix}$$

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \tau \begin{bmatrix} 0 & 1 \\ -1 & -\tau \end{bmatrix} = \begin{bmatrix} 1 & \tau \\ -\tau & 1-\tau^2 \end{bmatrix}$$

$$X[n+1] = \begin{bmatrix} 1 & \tau \\ -\tau & 1-\tau^2 \end{bmatrix} \cdot X[n] = X[n] + \begin{bmatrix} 0 & \tau \\ -\tau & -\tau^2 \end{bmatrix} \cdot X[n]$$

b) Avec l'équation précédente on peut écrire :

$$X[n+1] = X[n] + \tau \begin{bmatrix} 0 & 1 \\ -1 & -\tau \end{bmatrix} \cdot X[n] = X[n] + \tau \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot X[n] - \tau^2 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \cdot X[n]$$

le terme en τ^2 ne correspond pas à une dérivée seconde de X, c'est un terme négligeable.

La méthode est donc d'ordre 1.

La méthode semble stable car pour une durée de simulation relativement longue, la solution reste sinusoïdale bornée.

I.4 Discrétisation spatiale du problème (∼15%)

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -\lambda \cdot u$$

avec $\lambda > 0$

On pose également le pas dimensionnel $d = \frac{L}{N}$ avec N le nombre de segments du maillage.

12°) Equation adimensionnée

$$X = \frac{x}{L} \times N, \qquad Y = \frac{y}{L} \times N, \qquad U(X,Y) = u(x,y)$$

$$\Delta U = \frac{\partial^2 U}{\partial X^2} + \frac{\partial^2 U}{\partial Y^2} = -\lambda' \cdot U$$

Expression de λ' :
On a écrit à la question 6 :

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = \frac{-\omega^2}{c^2} \cdot u(x,y) = -\lambda \cdot u$$

$$\frac{\partial^2 U}{\partial X^2} = \frac{\partial^2 u}{\partial x^2} \cdot \frac{L^2}{N^2} \; et \; \frac{\partial^2 U}{\partial Y^2} = \frac{\partial^2 u}{\partial y^2} \cdot \frac{L^2}{N^2} \Leftrightarrow \lambda' = \frac{\omega^2 L^2}{c^2 N^2} \Leftrightarrow \lambda' = \frac{\omega^2 d^2}{c^2}$$

Le nouveau pas de discrétisation est égal à 1.

13°) Discrétisation autour d'un point central

a)

$$U_{i-1,j} = U_{i,j} + (X_{i-1} - X_i) \cdot \frac{\partial U}{\partial X} + \frac{(X_{i-1} - X_i)^2}{2} \cdot \frac{\partial^2 U}{\partial X^2} + o\left(\frac{\partial^2 U}{\partial X^2}\right) \approx U_{i,j} - \frac{\partial U}{\partial X} + \frac{1}{2} \cdot \frac{\partial^2 U}{\partial X^2}$$

b)

$$\begin{cases} U_{i-1,j} \approx U_{i,j} - \dfrac{\partial U}{\partial X} + \dfrac{1}{2} \cdot \dfrac{\partial^2 U}{\partial X^2} \\ U_{i+1,j} \approx U_{i,j} + \dfrac{\partial U}{\partial X} + \dfrac{1}{2} \cdot \dfrac{\partial^2 U}{\partial X^2} \\ U_{i,j-1} \approx U_{i,j} - \dfrac{\partial U}{\partial Y} + \dfrac{1}{2} \cdot \dfrac{\partial^2 U}{\partial Y^2} \\ U_{i,j+1} \approx U_{i,j} + \dfrac{\partial U}{\partial Y} + \dfrac{1}{2} \cdot \dfrac{\partial^2 U}{\partial Y^2} \end{cases} \Leftrightarrow \begin{cases} U_{i-1,j} - U_{i,j} + \dfrac{\partial U}{\partial X} \approx \dfrac{1}{2} \cdot \dfrac{\partial^2 U}{\partial X^2} \\ U_{i+1,j} - U_{i,j} - \dfrac{\partial U}{\partial X} \approx \dfrac{1}{2} \cdot \dfrac{\partial^2 U}{\partial X^2} \\ U_{i,j-1} - U_{i,j} + \dfrac{\partial U}{\partial Y} \approx \dfrac{1}{2} \cdot \dfrac{\partial^2 U}{\partial Y^2} \\ U_{i,j+1} - U_{i,j} - \dfrac{\partial U}{\partial Y} \approx \dfrac{1}{2} \cdot \dfrac{\partial^2 U}{\partial Y^2} \end{cases}$$

En sommant les 4 équations on obtient :

$$\Delta U \approx U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4 \cdot U_{i,j} = -\lambda' \cdot U_{i,j}$$

14°) Discrétisation autour des bords

a) Côté vertical (X_0, Y_j)

$$\Delta U \approx U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4 \cdot U_{i,j}$$

b) Côté horizontal (X_i, Y_0)

$$\Delta U \approx U_{i-1,j} + U_{i+1,j} + U_{i,j+1} - 4 \cdot U_{i,j}$$

c) Coin (X_0, Y_0)

$$\Delta U \approx U_{i+1,j} + U_{i,j+1} - 4 \cdot U_{i,j}$$

d) Dans ce cas les bords ont été considérés comme libres.

15°) Détermination de l'équation matricielle équivalente

$$\forall i \in [0 \cdots N]\ F_i = \begin{pmatrix} U_{i,0} \\ \vdots \\ U_{i,N} \end{pmatrix} \text{ et } F = \begin{pmatrix} F_0 \\ \vdots \\ F_N \end{pmatrix}$$

$$MF = \lambda' F$$

a) Modes propres La matrice M est une matrice carrée de taille $(N+1)^2 \times (N+1)^2$, il y a donc au maximum $(N+1)^2$ modes propres.

b) Matrice M

$$M = \begin{pmatrix}
4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 4
\end{pmatrix}$$

c) Généralisation On voit que la matrice M s'écrit pour $N = 4$ comme dans l'énoncé sous la forme :

$$M = \begin{pmatrix} A & -I_N & 0_N & 0_N & 0_N \\ -I_N & A & -I_N & 0_N & 0_N \\ 0_N & -I_N & A & -I_N & 0_N \\ 0_N & 0_N & -I_N & A & -I_N \\ 0_N & 0_N & 0_N & -I_N & A \end{pmatrix}$$

avec $A = \begin{pmatrix} 4 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & -1 & 4 \end{pmatrix}$. La forme reste la même pour $N > 4$.

d)

```
In [9]: def genereM(A):
            """Fonction dont l'objectif est de renvoyer M
            à partir de A"""
            n, p = A.shape # pour obtenir la taille de A
            M = np.zeros(shape=(n**2, n**2)) # On crée une matrice de taille n^2
            In = np.eye(n) # Création de la matrice I_N
```

```
            for i in range(n):
                M[i*n:(i+1)*n, i*n:(i+1)*n] = A.copy() # On copie A sur les blocs diagonaux
                if i < n-1:
                    M[(i)*n:(i+1)*n, (i+1)*n:(i+2)*n] = In.copy() #On met les matrices identité
                    M[(i+1)*n:(i+2)*n, (i)*n:(i+1)*n] = In.copy()
            return M # renvoie la matrice M

        def genereA(N):
            """Fonction dont l'objectif est de renvoyer A
            à partir de N
            Fonction non demandée dans l'énoncé et supposée disponible"""
            A = np.zeros(shape=(N+1, N+1))
            for i in range(N+1):
                A[i,i] = 4
                if i < N:
                    A[i, i+1] = -1
                    A[i+1, i] = -1
            return A

        N = 5
        A = genereA(N)
        print(A)
        M = genereM(A)
        print(M)
```

```
[[ 4. -1.  0.  0.  0.  0.]
 [-1.  4. -1.  0.  0.  0.]
 [ 0. -1.  4. -1.  0.  0.]
 [ 0.  0. -1.  4. -1.  0.]
 [ 0.  0.  0. -1.  4. -1.]
 [ 0.  0.  0.  0. -1.  4.]]
[[ 4. -1.  0. ...  0.  0.  0.]
 [-1.  4. -1. ...  0.  0.  0.]
 [ 0. -1.  4. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ...  4. -1.  0.]
 [ 0.  0.  0. ... -1.  4. -1.]
 [ 0.  0.  0. ...  0. -1.  4.]]
```

I.5 Méthode de la puissance itérée ($\sim$ 10%)

16°)

$$\|M\| = \max |m_{ij}| \; \forall i,j \in [0 \cdots n-1, 0 \cdots p-1]$$

a)

```
In [10]: def normeM(M):
             """Fonction dont l'objectif est de renvoyer
             la norme de M à partir de M"""
             n, p = M.shape #dimensions de M
             r = 0
             for i in range(n):
                 for j in range(p):
                     if abs(M[i, j]) > r:
                         r = abs(M[i, j])
             return r # norme de M
```

b)

$$F^* = \frac{F}{\|F\|}$$

```
In [11]: def normevecteur(F):
             Fn = F * 1 / normeM(F)
             return Fn # vecteur normé
```

17°)

$$F_{p+1} = \frac{M \cdot F_p}{\|M \cdot F_p\|}$$

a)

```
In [12]: def puissanceiter(M,p):
             n, m = M.shape
             Fp = rd.rand(n,1)
             for i in range(1, p): # le pième vecteur est celui d'indice p-1
                 Fp = normevecteur(M.dot(Fp))
             return Fp #pième vecteur
```

b)

```
In [13]: def iterstabilise(M,er):
             n, p = M.shape
             F = rd.rand(n,1)
             Fp = normevecteur(M.dot(F))
             while normeM(Fp - F) > er:
                 F  = normevecteur(M.dot(F))
                 Fp = normevecteur(M.dot(F))
             F = normevecteur(M.dot(F))
             return F # vecteur stabilisé
```

c) Le variant de boucle est une expression dont la valeur varie à chaque itération. Ici c'est l'expression normeM(normevecteur(M.dot(F)) - F).

Un invariant de boucle est une propriété qui si elle est vraie avant l'exécution d'une itération, elle reste vraie après l'exécution de cette itération.

18°) Vecteur et valeur propre principaux

a)

```
In [14]: N = 4
         A = genereA(N)
         M = genereM(A)
         F = iterstabilise(M, 10**-5)
         lambdap = F.transpose().dot(M.dot(F))[0][0]
         print(lambdap)

         M = np.array([[1.36, 0.56],[0.14,0.94]])
         F = iterstabilise(M, 10**-5)
         print(F)
         lambdap = (M.dot(F))[0][0] / F[0][0]
         print(lambdap)
```

```
67.17691522821627
[[1.        ]
 [0.25000619]]
1.5000034647135063
```

b) La fréquence propre associée à la valeur λ' est :

$$\lambda' = \frac{\omega^2 L^2}{c^2 N^2} = \frac{4\pi^2 f^2 L^2}{c^2 N^2} \Leftrightarrow f = \frac{cN}{2\pi L}\sqrt{\lambda'}$$

I.6 Représentation graphique (∼ 5)

```
In [15]: #librairies complémentaires
         from mpl_toolkits.mplot3d import Axes3D
         from matplotlib import cm

         def graphe(Z):
             n=Z.shape[0]
             X=Y = np.arange(n)
             X, Y = np.meshgrid(X, Y)

             fig = plt.figure(0, figsize=(10,8))
             ax = fig.gca(projection='3d')
             surf = ax.plot_surface(X, Y, Z,rstride=1,cstride=1,cmap=cm.Spectral_r, linewidth=0,
             ax.set_zlim(-1, 1)
             fig.colorbar(surf, shrink=0.5, aspect=10)
             plt.tight_layout()

             fig1=plt.figure(1, figsize=(6,6))
             cont=plt.contour(Z,np.arange(21)/10-1,cmap=cm.Spectral_r)
             plt.clabel(cont, cont.levels, inline=True, fmt='%1.1f', fontsize=10)
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.title('Courbes de niveau')
plt.tight_layout()
plt.show()
```

19°)

$$Z = \begin{pmatrix} U_{00} & \cdots & U_{0j} & \cdots & U_{0n} \\ \vdots & \cdots & \vdots & \cdots & \vdots \\ U_{i0} & \cdots & U_{ij} & \cdots & U_{in} \\ \vdots & \cdots & \vdots & \cdots & \vdots \\ U_{n0} & \cdots & U_{nj} & \cdots & U_{nn} \end{pmatrix}$$

```
In [16]: def transforme(F):
             n = F.shape[0]
             n = int(np.sqrt(n))
             Z = F.reshape(n, n)
             return Z # matrice des valeurs U
```

20°) Tracé graphique

Commentaire sur les courbes On a choisi une discrétisation avec $n = 100$ d'où le tracé sur l'intervalle $(x, y) \in [0, 100]^2$. Par ailleurs on a normalisé les déplacements tels qu'ils restent inférieurs à 1 en valeur absolue.

1.7 Distribution des valeurs propres ($\sim$ 15%)

21°)

a) Une recherche de valeur dans un tableau non trié a une complexité asymptotique en $\mathcal{O}(n)$, il faut comparer chaque terme du tableau à la valeur recherchée.

b)

```
In [18]: def recherche(valpropre, v, er):
             i = 0
             r = false
             while not r and i < len(valpropre):
                 r = abs(v - valpropre[i]) < er
                 i += 1
             return r # booléen
```

c) Complexité dichotomie La complexité asymptotique d'une recherche dichotomique dans un tableau de taille n est $\mathcal{O}(\log n)$

d) Algorithme de dichotomie L'algorithme de dichotomie est le suivant :

1. on compare la valeur recherchée à la valeur médiane du tableau 1. on compare la cible à la valeur médiane du tableau

   `si elle est supérieure on refait la recherche sur le demi tableau supérieur`

   `si elle est inférieure on refait la recherche sur le demi tableau inférieur`

2. on itère ainsi jusqu'à ce que la valeur cible soit égale à la valeur médiane du sous-tableau considéré ou que le sous tableau ne soit plus que de taille 2.

e) Algorithme en python

```
In [19]: def recherchedicho(valpropre, v, er): # proposition avec prise en compte de l'erreur de
             """
             Recherche dichotomique de présence d'une valeur dans un tableau trié par ordre croi
             """
             a = 0
             b = len(valpropre) - 1
             r = False
             mil = int((a + b) / 2) # indice milieu entre a et b
             while (b - a) > 1:
                 if abs(v - valpropre[mil]) < er: # On regarde si on a trouvé la valeur
                     return True
                 elif v - valpropre[mil] > 0:
                     a = mil # on change la borne inférieure
                 else:
                     b = mil # on change la borne supérieure
                 mil = (a + b) // 2 # on calcule le nouvel indice milieu
             return r # booléen
```

22°)

a) Recherche de la présence de valeurs propres dans une plage donnée Difficile d'utiliser la fonction de recherche précédente qui ne renvoie qu'un booléen. On ne donne pas de précision numérique, il n'est donc pas possible de connaître l'ensemble des valeurs à tester par dichotomie. En revanche une recherche dichotomique qui nous donnerait l'indice de la première valeur supérieure à valmin et l'indice de la dernière valeur inférieure à valmax pourrait être utile. On propose ici la solution où l'on connaîtrait la précision prise arbitrairement égale à 10^{-5}.

```
In [20]: def rechercheplagedicho(valpropre, valmin, valmax):
             V = np.arange(valmin, valmax, 10**-5)
             valselect = []
             for v in V:
                 if recherchedicho(valpropre, v, 10**-5):
                     V.append(v)
             return np.array(V)
```

b) Utilisation de numpy

```
In [21]: def valsearch(valpropre, valmin, valmax):
             mask=(valpropre>valmin) * (valpropre<valmax)
             return valpropre[mask]
```

valpropre>valmin renvoie une liste de booléens de la longueur de valpropre dont les termes sont True si la condition est vrai et False sinon.

Il en est de même pour valpropre<valmax

Le produit des 2 termes est un produit de booléens termes à termes. On a donc mask qui est une liste de booléens de la taille de valpropre et dont les termes valent True lorsque les 2 conditions sont respectées et False sinon.

valpropre[mask] renvoie donc seulement les termes de valpropre correspondant aux indices de mask dont les valeurs sont True.

c) complexité La complexité asymptotique de ce code est en $\mathcal{O}(n)$ car on compare 1 fois chaque terme à valmin et valmax

d) comparaison L'algorithme proposé utilisant la dichotomie a une complexité en $\mathcal{O}(m \log n)$ où m est le nombre de valeurs testées entre valmin et valmax et n la taille de valpropres. Pour une précision correcte et si la plage entre valmin et valmax est grande, m devient très grand et

$$m \log n \gg n$$